Rich booleans, version 2.2.2

An advanced portable framework for C++ that provides extra information to the user if a condition fails

Q-Mentum [http://www.q-mentum.com]

<info@q-mentum.com>

Abstract

The Rich Booleans framework is a set of macros that hold extra information if the condition in them fails, so a developer better understands what goes wrong. They allow modularization of assertions and other constructs that check conditions, so on the one hand we can have different types of assertion macros (e.g. in the ModAssert [http://sourceforge.net/projects/modassert/] package we have `MOD_ASSERT`, `MOD_VERIFY`, `MOD_ASSERT_P`, ...), and on the other hand macros that replace booleans in the assertion macros to provide extra information. So one could write `MOD_ASSERT(rbEQUAL(a,b))`, `MOD_VERIFY(rbLESS(foo(), 10))`, etc. These Rich Booleans could be reused in other situations, like an assertion macro in a unit testing framework, or in contracts. The macros just have to know how to handle a rich boolean.

This is a big improvement over the traditional assert function in C and C++. In different libraries, macros were written to extend the assert function, like `ASSERT_EQUAL(a,b)` to give a short explanation like "lefthand is <9>, righthand is <10>". However, such a macros functionality can't be reused in any way if we want to have a similar `VERIFY_EQUAL` (which should still evaluate its arguments in non-debug mode), or an `ASSERT_EQUAL_MSG` (to add a message). The Rich Booleans library allows to write modular assertions (like in the ModAssert library), that allow such reuse.

Over 90 interesting rich booleans are in this library, like `rbSTRING` that shows where two strings are different (taking care of mismatches and insertions in an intelligent way), and similar ones for ranges of iterators and STL containers, or `rbFILE` that checks whether a file has certain properties. They can be combined to make powerful checks, like checking whether the integer values in one vector are smaller than those in another vector, comparing two vectors of vectors, comparing a vector of objects to a vector of pointers to objects, or combining rich booleans with logical operations. Most Rich Booleans also allow binding of parameters, so one could e.g. check if all the integers in a vector are smaller than a given number.

The Rich Boolean macros are carefully constructed so they evaluate their arguments only once, and add parentheses around arguments where necessary.

It is released under the wxWindows Licence [http://opensource.org/licenses/wxwindows.php], so it can be used in both open source and commercial applications, without requiring provision of source, or runtime acknowledgements.

Support is available through the mailinglist [https://lists.sourceforge.net/lists/listinfo/modassert-users].

Table of Contents

Change log 3
    New in version 2.2.1 3
    New in version 2.2 3
    New in version 2.1 4
    New in version 2.0.1 4
    New in version 2.0 5
    New in version 1.4.1 5
    New in version 1.4 5
    New in version 1.3.1 6
    New in version 1.3 6
    New in version 1.2.1 6
    New in version 1.2 6
    New in version 1.1 6
    New in version 1.0 6
    New in version 0.9 7
    New in version 0.8 7
    New in version 0.7 7
    New in version 0.6 8
    New in version 0.5 8
    New in version 0.4 8
    New in version 0.3 9
    New in version 0.2 9
What is a Rich Boolean? 10
License 11
Support 11
Projects that use Rich Booleans 11
    ModAssert 11
    UquoniTest 11
Requirements 12
Installation 12
    Build the library 12
    Adjusting your development environment to use Rich Booleans 13
Adding Rich Booleans to individual projects 14
Linking to the Rich Booleans library 14
Using Rich Booleans 14
Possible problems when upgrading 14
Choose your strings 14
    Escaping 15
Known problems 15
The available rich booleans 15
    Comparing values 16
    Comparing strings 27
    Filesystem 32
    Type checking with RTTI 36
    Type checking with wxWidgets 37
    Logical expressions with Rich Booleans 39
    Exceptions 41
    Checking a range 42
    Available Rich Boolean functor classes for checking a range 43
    Multi-pass and single-pass ranges 51
    Comparing two ranges 51
    Available classes for performing checks on two ranges 52
    Multi-pass and single-pass ranges 63
    Dynamic matching 63

Containers with methods `begin()` and `end()` 64
Creating an inline container 66
Containers with overloaded funtions `begin()` and `end()` 67
Deprecated Rich Booleans 69
Using Rich Boolean functors in Rich Boolean macros 70
Getters 70
Combining Rich Boolean functors in logical operations 76
Expression objects 77
Binding parameters on a Rich Boolean functor 77
Selecting arguments 78
Removing the text from analyses 79
Making your own rich booleans (advanced) 79
Writing Rich Boolean functor classes using the wrappers 79
Writing a Rich Boolean Functor without the wrappers 86
Creating a Rich Boolean without the macros rbn_RB 89
Creating an analysis 90
Writing macros that use rich booleans (advanced) 93
Processing Rich Booleans (advanced) 93
Considerations related to macros 94
Thread safety 94
Exception safety 95
Warning levels 95
Converting objects to a string 95
STL containers 96
Containers with overloaded non-member functions begin and end 97
When converting containers to a string is needed 98
Windows types that can be converted to a string 98
The package RichBoolTest (advanced) 98
Index 99

Change log

New in version 2.2.1

- The function GetFileLength is added.
- Overloads for the function Values for const char* and const wchar_t* are added, to allow the usage of string literals of different lengths.
- The typedefs StringsEqualCS, WStringsEqualCS, StringsEqualCI and WStringsEqualCI are added to make string comparisons in containers or arrays easier.
- Some minor changes for usage in the next version of UquoniTest.

New in version 2.2

- The Rich Booleans rbREGEXP and rbREGEXP_F use a C++0x regular expression to check strings.
- The Rich Booleans rbIN_VALUES, rbIN_VALUES_CONTAINER, rbIN_VALUES_RANGE, rbIN_VALUES_ARRAY, rbIN_CONTAINER_VALUES, rbIN_RANGE_VALUES, rbIN_ARRAY_VALUES and rbIN_VALUES_VALUES use a C++0x initializer list to more easily provide values.

- The overloaded function template Values is added to provide values inline in a Rich Boolean.
- The Rich Boolean rbBITS_ARE is a generalization of rbBITS_ON and rbBITS_OFF, which is at the same time also more readable.
- The Rich Booleans rbFILE, rbDIRECTORY, rbFILE_EXISTS, rbDIRECTORY_EXISTS and rbDOES_NOT_EXIST now can have wide character arguments on Windows (except with the Cygwin compiler).
- The Rich Booleans rbSTRING_CS, rbSTRING_CI, rbSTRING_BEGINS_WITH_CS, rbSTRING_BEGINS_WITH_CI, rbSTRING_ENDS_WITH_CS, rbSTRING_ENDS_WITH_CI, rbSTRING_CONTAINS_CS and rbSTRING_CONTAINS_CI are case sensitive and case insensitive shorthands for rbSTRING, rbSTRING_BEGINS_WITH, rbSTRING_ENDS and rbSTRING_CONTAINS.
- Project files for Visual C++ 2010 are added.
- The project files for Code::Blocks now have a target for the Cygwin compiler, and they set a compiler flag to use C++0x (ignored if not available).

New in version 2.1

- There are now Rich Boolean objects IsSubsetOf, IsMultiSubsetOf and IsOrderedSubsetOf that can be used in Rich Boolean macros and functors that work on two ranges, and check whether the first range is a subset of the second range. They have optimized versions IsSubsetOfSorted, IsMultiSubsetOfSorted and IsOrderedSubsetOfSorted that can be used if the second range is sorted. The Rich Boolean factories MatchesSubsetOf, MatchesMultiSubsetOf, MatchesOrderedSubsetOf, IsSubsetOfCustomSorted, IsMultiSubsetOfCustomSorted and IsOrderedSubsetOfCustomSorted are generalizations of these.
- The Rich Boolean AllUnique checks whether all elements in a range are unique. The Rich Boolean factory AllPairs is a generalization of this.
- The macros rbIN_ARRAY_RANGE, rbIN_ARRAY_CONTAINER, rbIN_RANGE_CONTAINER, rbIN_CONTAINER_RANGE, rbIN_ARRAY_XCONTAINER, rbIN_RANGE_XCONTAINER and rbIN_XCONTAINER_RANGE are added
- The macros rbLESS_REL_NEAR, rbMORE_REL_NEAR and rbDIFF_REL_NEAR are added
- The Rich Booleans rbDOESNT_EXIST, rbFILE_EXISTS and rbDIRECTORY_EXISTS check whether a file or directory exists or not. The Rich Booleans rbFILE and rbDIRECTORY check whether a file or directory exists and has certain properties.
- Rich Boolean objects can be negated with the unary operator !
- Macros for making Rich Boolean functor classes from Rich Boolean functor core classes of which the constructor takes an argument, have been added.
- A dash is added before the state of an analysis when it is streamed out, e.g. '- ok' instead of 'ok'

New in version 2.0.1

- The documentation is updated to inform about a common compile error if you use Unicode with WxWidgets (which is not yet supported), and a comment is added where the compile error occurs.

New in version 2.0

- Writing custom Rich Booleans is now a lot easier by using classes that wrap a custom Rich Boolean Functorcore, which is a lot simpler than writing a Rich Boolean Functor.
- Writing custom Rich Booleans without a wrapper has changed a little to solve some problems, you now need to add 4 methods if your Rich Boolean takes 2 arguments.
- It is now possible to combine Rich Boolean functors in logical operations
- Most Rich Boolean macros now have a variation that can be used in assertion macros that return a value
- Comparing of two ranges and of two strings now uses points, so partial matching is used when you compare two ranges of containers or strings
- Pointers can now also be converted to a string
- The output of Rich Booleans that work on ranges is made more consistent with the output of other Rich Booleans (e.g. "range is sorted: `array'-`array+4' - nok" instead of "range is not sorted: `array'-`array+4'")
- A NULL char pointer that is converted to a string now yields the string "\NULL"
- The Rich Boolean Functor classes wxIsKindOf, wxHasType and wxEqualTypes are renamed to WxIsKindOf, WxHasType and WxEqualTypes because of a collision with a macro in the wxWidgets library.

New in version 1.4.1

- The conversion of single characters and character pointers to a string is now escaped.
- The warning that no assignment operator could be made for GetMember is resolved.

New in version 1.4

- The GetValue template parameters of Rich Boolean classes now can have a state
- Two templates classes are added to facilitate using a member or method of an object in Rich Boolean objects
- All macros now evaluate their arguments only once
- You can now eliminate the literal text in Rich Booleans to reduce the executable size
- The template class Pointer can now also be used in compilers that can't do partial template specialization, but can't be nested.
- The template classes PointerToValue and PointerLikeToValue are added that return a value instead of a const reference, to avoid problems when they are nested.
- Project files for Code::Blocks are added
- A bug in the Compare class using input interators, that caused it to walk too far, is solved
- The way Expression objects are used, is optimized. The class EmptyExpression is removed; where it was used, it is replaced by NULL pointers
- Endl no longer adds a carriage return on Windows, since ModAssert no longer needs this

- On Windows, LARGE_INTEGER and ULARGE_INTEGER can now be converted to a string with RichBool::ToString, so they can be used in Rich Booleans

New in version 1.3.1

- Warning levels are increased where it is feasible to clean up the remaining warnings
- Autolinking is now used for Visual C++.

New in version 1.3

- The license has changed to the wxWindows licence
- The Rich Booleans `rbSTRING_BEGINS_WITH`, `rbSTRING_ENDS_WITH` and `rbSTRING_CONTAINS` are added.
- The Rich Boolean rbEXCEPTION is added that gives information about an exception
- Rich Booleans that work on one or two ranges, can now also specify the range by giving the begin of the range and the number of elements to check on
- Undocumented classes and functions are put in the namespace detail inside the namespace RichBool

New in version 1.2.1

Solution and project files specific for Microsoft Visual Studio 2005 are added.

New in version 1.2

- The Rich Booleans `rbIN_ARRAY` and `rbIN_ARRAYS` have been added to perform a check on an entire array without having to give its size.
- Some performance improvements.
- The Rich Booleans ending in _ARG are deprecated. Instead, you should use All, Has, Unique and Compare as Rich Boolean factories in the equivalent Rich Booleans without the suffix _ARG.
- The Rich Boolean factory `CompareUnordered` and the Rich Boolean `AllEqualUnordered` are added.
- STL and WxWidgets containers can now be converted to a string.

New in version 1.1

- The Rich Booleans `rbORDER2`, `rbORDER3` and `rbORDER4` have been added to check relations on 2, 3 or 4 expressions in a flexible way.
- The Rich Booleans `rbWX_EQUAL_TYPES`, `rbWX_HAS_TYPE` and `rbWX_IS_KIND_OF` have been added to do runtime typechecking with the wxWidgets mechanism.
- The Rich Boolean `rbPRED4` is added to use a predicate that takes four arguments.

New in version 1.0

- Makefiles with a configure script are added, so compilation should work on most UNIX-like systems.

- The include directory is now called `richbool`
- The Rich Boolean rbSTRING can now work with widecharacter strings. Therefore, the template classes BasicStrcmp, BasicStrcmpToUpper, BasicStrcmpToLower, BasicCollate, BasicCollateToUpper and BasicCollateToLower are added, with typedefs for both char and wchar_t. They can also work with basic_string objects that don't have the default template arguments.
- Rich booleans that start with rbIN_STD_CONTAINER are renamed to start with rbIN_CONTAINER.
- Rich booleans that start with rbIN_WX_CONTAINER are renamed to start with rbIN_XCONTAINER.
- The Rich boolean `rbBOUNDARY` is renamed `rbIN_INTERVAL`.
- The Rich boolean `rbEQUAL_TYPES` now has pointers as arguments instead of references.
- The Rich booleans `rbXOR`, `rbIN_OPEN_INTERVAL`, `rbEQUAL_DATA` and `rbEQUAL_DATA_BITWISE` are added.
- Custom value getters now can have a prefix or a suffix added to their expression.
- Comparison of strings now takes care of char pointers that are null.
- Comparison of strings with collation now takes better care of characters that are ignored.
- Comparison of MFC CStrings can now be done.
- Only output of strings is now escaped, for efficiency.

New in version 0.9

- The Rich Booleans rbEQUAL_STRING, rbSTRCMP_EQUAL and rbSTRICMP_EQUAL are replaced by the more generic rbSTRING, which also allows to check if a string is lexicographically less than another string, and can use locales.
- Three Rich booleans are added to do runtime typechecking: rbEQUAL_TYPES, rbHAS_TYPE and rbDYNAMIC_CASTABLE.

New in version 0.8

- The Rich Booleans that work on ranges and containers have been drastically rewritten, to maximize simplicity and genericity. It is now possible to check whether a predicate applies to all elements, at least one element or exactly one element in a range or container. Analysis of such checks can give the data of all the elements in the range, or only the ones for which the predicate doesn't apply.
- Analysis of most Rich booleans now contains the textual representation of the arguments; if they're used in a range or container, the index of the element is shown instead.

New in version 0.7

- It is now possible to use partial matches in custom rich booleans, which is useful when two ranges of objects are compared (this is also used in the rich booleans `rbSTD_MAPS_EQUAL` and `rbWX_MAPS_EQUAL`)
- Code that uses rich booleans can now tell a rich boolean to only evaluate the arguments, if required

- Code that uses rich booleans can now tell a rich boolean to always create an analysis, even if the condition succeeds

New in version 0.6

- The rich booleans `rbRANGE_SORTED`, `rbRANGE_SORTED_STRICTLY` are added that check if a range is sorted, and the rich booleans `rbRANGE_ADJACENT_PRED` and `rbRANGE_ADJACENT_RB` are added that check a condition on adjacent elements in a range
- The rich booleans `rbSTD_MAPS_EQUAL` and `rbWX_MAPS_EQUAL` are added that check if two maps are equal
- Project files for wxWidgets are adjusted to wxWidgets 2.6.1 (include and library directories were adjusted)
- The class `Info` is renamed to `Analysis`, to avoid name confusion with the level `Info` in ModAssert, if both namespaces would be used with using; derived classes are also renamed accordingly, the method `GetInfo` of Rich Boolean classes is renamed `Analyse`, and some more minor name changes
- Rich Boolean classes now have `typedef bool Points;` added, to take future changes into account
- The output of Rich Booleans now show valid values between parentheses instead of curly braces.

New in version 0.5

- The rich booleans `rbRANGE_PRED`, `rbRANGE_RB`, `rbRANGE_SP_PRED` and `rbRANGE_SP_RB` have been added to check if all the elements in a range fullfil a given condition. Similar rich booleans have been added for STL containers and wxWidgets containers.
- You can now bind the arguments of rich booleans with two or three arguments.
- The macros `rbEQUAL_BITWISE`, `rbBITS_ON` and `rbBITS_OFF` are added to do bitwise checks.
- The extension of header files is changed to .hpp, and uppercase symbols in filenames were made lowercase.

New in version 0.4

- You can now compare a container of pointers to objects with a container of objects, two containers of pointers to objects or even more complex combinations. You can do the same with iterators, smart pointers and other pointerlike objects.
- The Rich Boolean `rbDELTA` is renamed `rbNEAR`.
- The Rich Boolean `rbREL_NEAR` has been added that checks whether the absolute value of the difference between two numbers is less than a given percentage of the biggest.
- If a bad pointer is dereferenced in a Rich Boolean, its address will be shown (if it is recognized as a bad pointer)
- All `Info` objects now have a flag that indicates whether the condition was ok, not ok or not evaluated
- Objects of the class `TextInfo` now can have other `Info` objects embedded in them
- `InfoWithStrings` now has a template constructor to customize the comparing of characters
- `rbPRED1`, `rbPRED2` and `rbPRED3` are renamed `rb1_PRED`, `rb2_PRED` and `rb3_PRED`

- Rich Boolean macros `rb1_RB`, `rb2_RB` and `rb3_RB` are added, that take a Rich Boolean object as their last argument
- `Pred2` and `Pred3` now have different stringize template parameters, so their arguments can be serialized in different ways

New in version 0.3

- Rich Booleans `rbEQUAL_RANGES_SP2`, `rbEQUAL_RANGES_SP2_PRED`, `rbEQUAL_RANGES_SP2_RB`, `rbEQUAL_ARRAY_RANGE_SP2`, `rbEQUAL_ARRAY_RANGE_SP2_PRED` and `rbEQUAL_ARRAY_RANGE_SP2_RB` are added to compare two ranges where the second range can be traversed only once, so a single pass algorithm is used for the second range
- Rich Booleans `rbEQUAL_RANGES_SP12`, `rbEQUAL_RANGES_SP12_PRED` and `rbEQUAL_RANGES_SP12_RB` are added to compare two ranges where both ranges can be traversed only once, so a single pass algorithm is used for both ranges
- The class `RichBool::Info` and its derived classes no longer have a method `GetText`. Instead they now have `operator<<` to stream their data into a stream, which is more efficient.
- For wxWidgets, streaming is now done to a `wxTextOutputStream` object instead of a `wxString`, which is more efficient when the output is written to a `wxTextOutputStream` object.
- The Rich Booleans that compare ranges or containers have a new, more consistent name, e.g. `rbEQUAL_RANGES_PRED` is now `rbRANGES_PRED` (because they're not necessarily equal), and `rbEQUAL_RANGES` is now `rbRANGES_EQUAL` to be consistent with `rbRANGES_PRED`
- Rich Booleans that compare containers have been renamed. STD_CONT becomes STD_CONTAINER or STD_CONTAINERS (depending on whether one or two containers are involved), and WX_CONT becomes WX_CONTAINER or WX_CONTAINERS (idem). Furthermore the same renaming changes as in the previous item were applied.
- The order of the arguments for the Rich Boolean `rbBOUNDARY` has changed.
- The package RichBoolTest is now in the same download.
- The text "ranges have different content" in the output of Rich Booleans that compare ranges, has been changed to "predicate doesn't apply on those ranges" if a predicate or rich boolean is used to compare the ranges instead of equality.
- A bug where comparison of ranges is not done if `richbool_check` is false, was solved.
- The method `BadPtr` now has an extra argument, the size of the object. Some platforms can check for this, like Microsoft Visual C++. If that compiler is used, `BadPtr` will use the function `IsBadReadPtr`.

New in version 0.2

- Rich Booleans `rbEQUAL_RANGES_RB` and `rbEQUAL_ARRAY_RANGE_RB` to use another Rich Boolean to pairwise compare the elements of two ranges
- Rich Booleans `rbEQUAL_ARRAY_STD_CONT`, `rbEQUAL_ARRAY_STD_CONT_PRED`, `rbEQUAL_ARRAY_WX_CONT` and `rbEQUAL_ARRAY_WX_CONT_PRED` to compare a container to an array

- Rich Booleans `rbEQUAL_STD_CONT_RB`, `rbEQUAL_ARRAY_STD_CONT_RB`, `rbEQUAL_WX_CONT_RB` and `rbEQUAL_ARRAY_WX_CONT_RB` to to use another Rich Boolean to pairwise compare the elements of two ranges

- Most Rich Booleans now have a corresponding class with methods `bool operator(...)` and `RichBool::Info* GetInfo(...)`, that the Rich Boolean macros call; the classes can be reused in the Rich Booleans mentioned above ending in _RB

- Rich Booleans `rbOR` and `rbAND` to combine Rich Booleans (also `rbOR_DE`, `rbOR_BE`, `rbAND_DE` and `rbAND_BE` to allow nesting)

- Rich Boolean `rbBOUNDARY` to check if a value is between two others

- Rich Booleans `rbPRED1`, `rbPRED2` and `rbPRED3` to easily use predicates

- Text returned by many Rich Booleans is made shorter, using operators instead of words; e.g. "{5} == {6} - nok" instead of "expected {5} to be equal to {6}". This way they are also independent of language.

- Valid values are now shown between { and } instead of < and >, to distinguish easily between values and operators; e.g. "{5} < {4} - nok" instead of "<5> < <4> failed"

- The class `RichBool::SharedInfo` is added, a shared pointer class for objects of the type `RichBool::Info`. Therefore the method `clone()` in `RichBool::Info` and derived classes is no longer necessary, and is removed.

- The class `RichBool::InfoWithExpressions` is renamed `RichBool::TextInfo`

- The text `NULL pointer` is replaced by the more accurate `bad pointer`

- A new package called RichBoolTest was made, that contains tests for every Rich Boolean (before the tests were in the package ModAssert).

- The preprocessor definition RICHBOOL_USE_WX_STRING has been deprecated, use RICHBOOL_USE_WX instead. RICHBOOL_USE_WX_STRING will disappear in the next version.

- Filenames have been shortened so they fit the 8.3 format, to improve portability.

- The function `bool BadPtr()` has been improved to detect the usage of uninitialized memory with Microsoft Visual C++ (in debug mode).

What is a Rich Boolean?

A Rich Boolean is a macro, that is a bit like a boolean expression, e.g. `rbEQUAL(a,b)` instead of `a==b`. The main difference is that it holds an analysis of the boolean expression, if the condition fails. The analysis can be streamed out, and e.g. output "`a':<1> == `b':<2> - nok". Between ` and ' are the expressions, between < and > are their values. "nok" indicates that the condition failed (in some circumstances it is shown when it succeeded, then it would end in "ok").

A Rich Boolean can be assigned to an object of the class `RichBool::Bool`. This class has a method `bool operator()`, which tells if the condition succeeded or not, and `const RichBool::SharedAnalysis& GetAnalysis() const` that returns a shared pointer to the analysis (it may be NULL). Furthermore some objects need to be declared, so usually this is all taken care of with a macro (see the section called “Writing macros that use rich booleans (advanced)”). See the section called “Projects that use Rich Booleans” for some libraries that have such macros.

A Rich Boolean doesn't always create an analysis, to save time and memory. You can specify (at compiletime or runtime) to do one of the following:

- only evaluate the arguments of the Rich Boolean macro
- only evaluate the condition without creating an analysis
- only create an analysis if the condition fails
- always create an analysis

Most Rich Boolean macros have a corresponding class, e.g. `RichBool::Equal`, which are called Rich Boolean functor classes. Objects of these classes are called Rich Boolean functors, and can be used in more complex Rich Boolean macros and even in certain Rich Boolean functor classes; they can also act as regular functors (function objects) because they have a method `operator()(...)`.

License

The Rich Booleans package is released under the wxWindows Licence. This is basically the LGPL license, but with an exception notice that states that you can use, copy, link, modify and distribute the code in binary form under your own terms. So you can use it in e.g. commercial applications without having to reveal the source code; you don't even have to mention that you use the Rich Booleans package.

See the files `COPYING.LIB` and `LICENCE.txt` in the main directory for the complete license, or online at http://opensource.org/licenses/wxwindows.php

Support

To get help with Rich Booleans, or to discuss suggestions, subscribe to the mailinglist [https://lists.sourceforge.net/lists/listinfo/modassert-users]. This list is for both ModAssert and Rich Booleans, because they usually are used together.

Projects that use Rich Booleans

ModAssert

ModAssert [http://sourceforge.net/projects/modassert/] is an open source project that allows to use Rich Booleans in 128 different assertion macros. The assertions can optionally have expressions that are shown in case of a failure, a level, a group, and an action to be taken. Failing assertions are passed to loggers and a displayer object that asks the user what to do. The loggers and the displayer object can be set at runtime by the application. You can associate filters with loggers and the displayer, so you can filter out assertions on several criteria. Two dialogs are provided to show the Rich Boolean and the expressions, one for Win32 and one for wxWidgets. There is also code to display the Rich Boolean and the expressions in command line applications. Several loggers are provided that log to a stream, or to the trace window in Win32 applications. For Rich Booleans version 2.2.2, you need at least ModAssert version 1.3. Note: the examples in this document use the macro `MOD_ASSERT` from the ModAssert package.

UquoniTest

UquoniTest [http://www.q-mentum.com/] is a unit testing package for C++ developed by Q-Mentum. Its assertions can use Rich Booleans. The assertions can optionally have expressions that are shown in case of a failure. They stop the test after a failure, but there are variations that continue after a failure, so assertions after it are still evaluated. If the evaluation of the condition in an assertion causes problems, the assertion can be modified so it is ignored, which causes a warning to remind the developer of the ignored assertion.

These eight variations of the assertion macro are made possible by using Rich Booleans, because they allow the assertion macros to vary in their behaviour, and leave the analysis to the Rich Boolean (just as with the assertions in ModAssert).

UquoniTest has other features like: automatic registering of tests, easy fixture tests, parameterized and template tests, integration with assertions in domain code, easy abstract tests (including multiple abstraction levels and multiple inheritance), create test directories (optionally prefilled with files and directories), an extensive mock library, static and dynamic testsuites, custom testlisteners and testwrappers, orthodoxy testing, timeouts on tests, test time measured up to microseconds, and more. Unlike some other unit testing packages for C++, UquoniTest allows you to place breakpoints in test code.

Requirements

- A C++ compiler that can compile namespaces and templates. There are two features that require partial template specialization, but they can be circumvented if your compiler can't do this. If your compiler can't do this, adjust the header file `richbool/config.hpp`, so that the symbol `RICHBOOL_NO_PARTIAL_TEMPLATE_SPECIALIZATION` is defined, so other headerfiles exclude that functionality. This is already done for Microsoft Visual C++ 6.0.
- STL or wxWidgets

Installation

Download the file and decompress its contents to a directory.

Build the library

Below are instructions to build the Rich Booleans library in the configurations that you need. You only need to build the RichBool library. The other projects whose name contains 'test' are only provided for experienced users who wish to test their own Rich Booleans, or test the existing Rich Booleans in different ways than was already done (although it is unlikely that this is necessary).

Using the Visual Studio project files

Please use the workspace, solution and project files that are specific to the version of your compiler (`*.dsw` and `*.dsp` for 6.0, `*7.1.sln` and `*7.1.vcproj` for .NET 2003, `*8.0.sln` and `*8.0.vcproj` for .NET 2005, `*9.0.sln` and `*9.0.vcproj` for .NET 2008). Converting these files to another version may require adjustments.

Note that these project files are not tested with Visual Studio Express. For Visual Studio Express you probably just need to exclude the files that begin with 'mfc'.

The included Visual Studio project files have many configurations. The configurations that have ST in their name (Debug ST, Release ST) are for applications built with the Single-Threaded option, the ones that have MT in their name (Debug MT, ...) are for applications built with the Multithreaded option, and the ones that have MTD in their name (Debug MTD, Debug Wx MTD, ...) are for applications built with the Multithreaded DLL option.

The configurations that have Wx in their name (Debug Wx MTD, Release Wx MTD) are for use with wxWidgets; these have only been tested without Unicode support, and with the static library version. For wxWidgets, there are only configuations with MTD in their name, because the wxWidgets libraries are built with the Multithreaded DLL option by default, and the Multithreaded configuration of wxWidgets is not well supported.

The libraries are built in the directory `lib`.

Using the GNU makefiles

To use the GNU makefiles, enter the command `./configure`, followed by `make`. You can also build the wxWidgets version by entering the command `./configure --with-wx` instead of `./configure`, this defines the symbol `RICHBOOL_USE_WX` during compilation.

Using the Windows makefiles for Mingw gcc

The makefiles `Makefile.win` in several directories can be used to build the Rich Booleans library and the tests, but you will probably have to adjust the directories for the include files and library files. They have only one configuration, the one that doesn't use wxWidgets.

Using the Code::Blocks project files

The files with the extension `cbp` and `workspace` in several directories are respectively Code::Blocks project files and workspaces. They have only one configuration, the one that doesn't use wxWidgets.

Other compilers

Create a library project (or makefile), add the `include` directory to the include path, and add all the .cpp files, except the ones that start with 'mfc'.

Adjusting your development environment to use Rich Booleans

To use Rich Booleans in your application or library, you need to specify the include and library directory. There are two options to do this: one option makes the Rich Booleans available to all your projects, the other option is to specify it only for the projects in which you want to use Rich Booleans. The first option is preferred, especially if you plan to use Rich Booleans in many projects. There is no danger for confusion with include files from other libraries with the same name, as long as you keep the include files in the `richbool` directory and only add its parent directory to the include path.

This section describes how to make Rich Booleans available to *all* your projects.

Microsoft Visual C++

Add the directory called `include` in the RichBool directory to the VC++ include directories (Tools -> Options -> Projects and solutions -> VC++ Directories, and select Include files in the combo box on the right), and add the directory called `lib` in the RichBool directory to the VC++ include directories (Tools -> Options -> Projects and solutions -> VC++ Directories, and select Library files in the combo box on the right)

gcc

Install the library in `/usr/local/lib/` and the header files (the directory `richbool`, not only the headerfiles in it) in `/usr/local/include` (or in other directories that are respectively in LIBRARY_PATH and CPLUS_INCLUDE_PATH). You can do this with the command `make install` as a superuser (if you already ran `./configure` or `./configure --with-wx`), this builds the library and installs it in `/usr/local/lib/` and puts the header files in `/usr/local/include`. An alternative is to add the directories to LIBRARY_PATH and CPLUS_INCLUDE_PATH.

Other compilers

For other compilers, consult the documentation of your compiler.

Adding Rich Booleans to individual projects

If you chose to not make the Rich Booleans available to all your projects (as explained in the previous section), you have to setup every project that uses it correctly as explained in this section. Make sure that the headerfiles can be included, and that the libraries can be found.

- For Microsoft Visual C++, it is best to make an environment variable called RICHBOOL, that contains the directory where the Rich Booleans are, to make upgrading easier. Then add `$(RICHBOOL)/include` to your include path (Project Settings -> C/C++ -> Preprocessor). Add `$(RICHBOOL)/lib` to your additional library path (Project Settings -> Link -> Input).
- For gcc, use the -I and -L command line options to specify the include and library directories.
- For other compilers, consult the documentation of your compiler.

Linking to the Rich Booleans library

Link the Rich Booleans library with your executable (unless you use Visual C++, then the necessary libraries are linked automatically; this probably also works with Borland C++ builder, but this has not been tested).

Note: if you use Visual C++ and link statically to the MFC library, you might have to include `afxwin.h` in at least one source file, to force the linker to link with the correct libraries, to avoid link errors.

Using Rich Booleans

Rich booleans can be used on their own, but it's not the easiest way to use them. Usually you need them when you use another library that uses them (see the section called “Projects that use Rich Booleans”). If that library includes the headerfiles for the rich booleans you need, you don't need to include them in your sourcecode, but you might have to add others. Otherwise, you just include `richbool/richbool.hpp` (or others for specific Rich Booleans).

Possible problems when upgrading

If you use Microsoft Visual C++ or Borland C++, the autolinking mechanism may cause link errors if some `.obj` files still contain the instruction to link to the old version. In that case you should rebuild your application entirely.

If you use ModAssert as well, you may have to rebuild these libraries as well.

Choose your strings

Most macros in this package suppose that their arguments can be converted to a string, so that their values can be displayed. If you provide an argument of a type that can't be converted to a string, a compile error will be given.

At present, the rich booleans can use two types of strings to show values, `std::string` and `wxString` of the wxWidgets library. By default, `std::string` is used. If you want to use `wxString`, you should add the symbol `RICHBOOL_USE_WX` in your makefile or project, and link with the richbool library that is also built with this symbol. See the section called “Converting objects to a string” for more information.

Wide characters strings cannot be used yet to stream objects into, not even with `wxString`. If you use `wxString` and get a compile error in `portability.hpp`, you probably use unicode characters, so you should switch to Ansi strings or multi-byte strings. By default VC++ 2005 uses unicode characters; you can change this in the project properties, more specifically the entry Character Set under General. But wide characters strings can be streamed into ordinary characters strings, where necessary escaping is added. On some compilers a single wide character will be considered an unsigned short, so it will be shown as a number. If your compiler can distinguish a single wide character from an unsigned short, adjust the file `richbool/config.hpp` so that `RICHBOOL_WCHAR_T_IS_USHORT` is `0`. With Visual C++ 6.0 this is not possible. With Visual C++ 2003 and later it is possible if you set a switch in the project settings: C/C++ -> Language -> Treat w_char_t as Built-in charater. With (recent versions of) gcc it is always possible, so you don't have to adjust this.

Escaping

All output of string objects and characters by Rich Booleans is escaped. So non-printable characters are converted to their hexadecimal value, preceded by a backslash and a character 'x'. Some values are converted to the way they would appear in source code, like `'\n'` and `'\t'`. A backslash is converted to `'\\'`.

For efficiency, escaping is not done for objects of other classes and primitive types. If you define a stream operator for your own objects, make sure only printable characters are outputted to keep the output readable; otherwise you may overload the method `RichBool::ToString` (see the section called “Converting objects to a string”), which by default uses the stream operator.

Known problems

If you use Microsoft Visual C++ 6.0, and optimizations are not disabled (e.g. in Release mode), rbDYNAMIC_CASTABLE_REF will crash the application if the casting can't be done.

If you use Microsoft Visual C++ 6.0, the content of a `std::string` object will not be escaped with Rich Booleans like `rbEQUAL`, `rbLESS`, `rbMORE` and similar Rich Booleans (`std::wstring` and `wxString` objects don't have this problem). But it is recommended to use `rbSTRING` instead anyway, where this problem doesn't occur.

The available rich booleans

All the classes mentioned in this section are in the namespace RichBool, which is omitted in the text for brevity.

Note: Many Rich Booleans have two versions, one that starts with `rb` followed by uppercase letters or digits, and another that starts with `rbv` followed by uppercase letters or digits. The Rich Booleans of the first kind are meant for assertion macros that don't return a value, the Rich Booleans of the second kind are meant for assertion macros that do return a value. The difference has to do with implementation issues. An assertion macro that returns a value will usually return the first argument of the Rich Boolean, but there are some exceptions, which are mentionned in the documentation of the Rich Booleans in question below.

If you use Visual C++ 6 and use a literal string in a Rich Boolean that returns a value, you should cast it to `const char*`, due to a limitation of this compiler.

```
// does not work with Visual C++ 6:
```

```
const char *name = MOD_VERIFY_V(rbvFILE_EXISTS("file.txt"));

// ok:
const char *name = MOD_VERIFY_V(rbvFILE_EXISTS((const char*)"file.txt"));
```

Comparing values

To use the Rich Booleans in this section, include `richbool/richbool.hpp` Note: the parameters of these Rich Booleans can have different types, as long as the required operators and function are overloaded for them. E.g. for `rbEQUAL(a,b)` you can give a `std::string` object and a character literal, because `operator==` is defined on these.

rbEQUAL(a,b), rbvEQUAL(a,b)

This macro checks whether its arguments are equal (with `operator==`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<1> == `b':<2> - nok
```

Corresponding class: `Equal<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbEQUAL_PTR(a,b), rbvEQUAL_PTR(a,b)

This macro checks whether its arguments, which should be pointers, are equal (with `operator==`). Upon failure, it converts its arguments to strings, and if they are considered valid pointers, the values that they point to will also be converted to a string, which will all be shown in the output.

Example output:

```
(0x0012feb0 -> 1) == (0x0012feac -> 1) - nok
```

Example output:

```
(0x0012feb0 -> 1) == (0x00000000) - nok
```

Corresponding class: `EqualPtr<class GetValue1=Value, class GetValue2=Value >`

rbLESS(a,b), rbvLESS(a,b)

This macro checks whether `a` is less than `b` (with `operator<`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<3> < `b':<2> - nok
```

Corresponding class: `Less<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbLESS_OR_EQUAL(a,b), rbvLESS_OR_EQUAL(a,b)

This macro checks whether `a` is less than or equal to `b` (with `operator<=`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<3> <= `b':<2> - nok
```

Corresponding class: `LessOrEqual<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbMORE(a,b), rbvMORE(a,b)

This macro checks whether `a` is bigger than `b` (with `operator>`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<2> > `b':<3> - nok
```

Corresponding class: `More<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbMORE_OR_EQUAL(a,b), rbvMORE_OR_EQUAL(a,b)

This macro checks whether `a` is more than or equal to `b` (with `operator>=`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<2> >= `b':<3> - nok
```

Corresponding class: `MoreOrEqual<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbDIFF(a,b), rbvDIFF(a,b)

This macro checks whether its arguments are different (with `operator!=`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<2> != `b':<2> - nok
```

Corresponding class: `Diff<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbEQUAL_USING_LESS(a,b), rbvEQUAL_USING_LESS(a,b)

This macro checks whether its arguments are equal, but using `operator<` instead of `operator==` (i.e. by evaluating `!(a<b) && !(b<a)`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<1> == `b':<2> - nok
```

Corresponding class: `EqualUsingLess<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbEQUAL_USING_MORE(a,b), rbvEQUAL_USING_MORE(a,b)

This macro checks whether its arguments are equal, but using `operator>` instead of `operator==` (i.e. by evaluating `!(a>b) && !(b>a)`). Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<1> == `b':<2> - nok
```

Corresponding class: `EqualUsingMore<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbIN_INTERVAL(val, low, high), rbvIN_INTERVAL(val, low, high)

This macro checks whether `val` is in the closed interval `low` and `high`. It only uses `operator<` on the arguments. Upon failure, it converts its arguments to strings, which will be shown with the message that `val` was not in the interval of `low` and `high`.

It is recommended to use the Rich Boolean `rbORDER3` instead, which is more flexible, unless the concept of an interval (or range) is important in your code and you want to express that by using this Rich Boolean.

Example output:

```
`val':<6> in [`low':<2>, `high':<4>] - nok
```

Corresponding class: `InInterval<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbIN_OPEN_INTERVAL(val, low, high), rbvIN_OPEN_INTERVAL(val, low, high)

This macro checks whether `val` is in the open interval (`low`, `high`). It only uses `operator<` on the arguments. Upon failure, it converts its arguments to strings, which will be shown with the message that `val` was not in the open interval of `low` and `high`.

It is recommended to use the Rich Boolean `rbORDER3` instead, which is more flexible, unless the concept of an interval (or range) is important in your code and you want to express that by using this Rich Boolean.

Example output:

```
`val':<4> in (`low':<2>, `high':<4>) - nok
```

Corresponding class: `InInterval<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbORDER2(a, op, b), rbvORDER2(a, op, b)

This macro checks whether its arguments `a` and `b` satisfy the relation `op`. `op` can be `==`, `<`, `<=`, `>`, `>=` or `!=`. Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<2> < `b':<1> - nok
```

Corresponding class: `Order2<class Relation, class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>` `Relation` can be any of `EqualRelation`, `LessRelation`, `LessOrEqualRelation`, `MoreRelation`, `MoreOrEqualRelation` and `DifferentRelation`.

rbORDER3(a, op1, b, op2, c), rbvORDER3(a, op1, b, op2, c)

This macro checks whether its arguments `a`, `b` and `c` satisfy the relations `a op1 b` and `b op2 c`. `op1` and `op2` can be `==`, `<`, `<=`, `>`, `>=` or `!=`. Upon failure, it converts its arguments to strings, which will be shown in the output.

Note: Assertion macros that return a value, and have `rbvORDER3` as the condition, will return the third argument (i.e. the second value), not the first value.

Example output:

```
`a':<2> < `b':<1> < `c':<3> - nok
```

Corresponding class: `Order3<class Relation1, class Relation2, class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>` `Relation1` and `Relation2` can be any of `EqualRelation`, `LessRelation`, `LessOrEqualRelation`, `MoreRelation`, `MoreOrEqualRelation` and `DifferentRelation`.

rbORDER4(a, op1, b, op2, c, op3, d)

This macro checks whether its arguments `a`, `b`, `c` and `d` satisfy the relations `a op1 b`, `b op2 c` and `c op3 d`. `op1`, `op2` and `op3` can be `==`, `<`, `<=`, `>`, `>=` or `!=`. Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<1> < `b':<2> <= `c':<2> < `d':<0> - nok
```

Corresponding class: `Order4<class Relation1, class Relation2, class Relation3, class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class GetValue4=Value, class Stringize=MakeString>` `Relation1`, `Relation2` and `Relation3` can be any of `EqualRelation`, `LessRelation`, `LessOrEqualRelation`, `MoreRelation`, `MoreOrEqualRelation` and `DifferentRelation`.

rbNEAR(a,b,diff), rbvNEAR(a,b,diff)

This macro checks whether its two first arguments are close enough, i.e. if their difference isn't more than the third argument. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Example output:

```
fabs(`a':<1.0>-`b':<2.0>) = 1 <= `diff':<0.1> - nok
```

Corresponding class: `Near<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbLESS_NEAR(a,b,diff), rbvLESS_NEAR(a,b,diff)

This macro checks whether first argument is less than the second, but allowing a tolerance of `diff`, i.e. if the first is between `b` and `b+diff`, it is still accepted. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Note that this is *less* strict than `rbLESS`.

Example output:

```
`a':<1.0> <~`b':<2.0> (difference is <1>, `diff':<0.1> allowed) - nok
```

Corresponding class: `LessNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbMORE_NEAR(a,b,diff), rbvMORE_NEAR(a,b,diff)

This macro checks whether the first argument is more than the second, but allowing a tolerance of `diff`, i.e. if the first is between `b` and `b-diff`, it is still accepted. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Note that this is *less* strict than `rbMORE`.

Example output:

```
`a':<1.0> >~`b':<2.0> (difference is <1>, `diff':<0.1> allowed) - nok
```

Corresponding class: `MoreNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbDIFF_NEAR(a,b,diff), rbvIFF_NEAR(a,b,diff)

This macro checks whether the first argument is different from the second, requiring a tolerance of `diff`, i.e. if the first is between `b-diff` and `b+diff`, it is neither accepted. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Note that this is *more* strict than `rbDIFF`.

Example output:

```
`a':<1.0> >~`b':<2.0> (difference is <1>, `diff':<0.1> allowed) - nok
```

Corresponding class: `MoreNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbIN_INTERVAL_NEAR(val, low, high, diff), rbvIN_INTERVAL_NEAR(val, low, high, diff)

This macro checks whether `val` is in the closed interval `low` and `high`. It only uses `operator<` on the arguments. Upon failure, it converts its arguments to strings, which will be shown with the message that `val` was not in the interval of `low` and `high`.

It is recommended to use the Rich Boolean `rbORDER3` instead, which is more flexible, unless the concept of an interval (or range) is important in your code and you want to express that by using this Rich Boolean.

Example output:

```
`val':<6> in [`low':<2>, `high':<4>] - nok
```

Corresponding class: `InIntervalNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbORDER2_NEAR(a, op, b, diff), rbvORDER2_NEAR(a, op, b, diff)

This macro checks whether its arguments `a` and `b` satisfy the relation `op`, allowing a tolerance of `diff`. `op` can be ==, <, <=, >, >= or `!=`. Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<2> <~ `b':<1> - nok
```

Corresponding class: `Order2Near<class Relation, class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>` `Relation` can be any of `EqualNearRelation`, `LessNearRelation`, `MoreNearRelation` and `DifferentNearRelation`.

rbORDER3_NEAR(a, op1, b, op2, c, diff), rbvORDER3_NEAR(a, op1, b, op2, c, diff)

This macro checks whether its arguments `a`, `b` and `c` satisfy the relations `a op1 b` and `b op2 c`, allowing a tolerance of `diff`. `op1` and `op2` can be ==, <, <=, >, >= or `!=`. Upon failure, it converts its arguments to strings, which will be shown in the output.

Note: Assertion macros that return a value, and have `rbvORDER3` as the condition, will return the third argument (i.e. the second value), not the first value.

Example output:

```
`a':<2> < `b':<1> < `c':<3> - nok
```

Corresponding class: `Order3Near<class Relation1, class Relation2, class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>` `Relation1` and `Relation2` can be any of `EqualNearRelation`, `LessNearRelation`, `MoreNearRelation`, and `DifferentNearRelation`.

rbORDER4_NEAR(a, op1, b, op2, c, op3, d, diff)

This macro checks whether its arguments `a`, `b`, `c` and `d` satisfy the relations `a op1 b`, `b op2 c` and `c op3 d`. `op1`, `op2` and `op3` can be ==, <, <=, >, >= or `!=`. Upon failure, it converts its arguments to strings, which will be shown in the output.

Example output:

```
`a':<1> < `b':<2> <= `c':<2> < `d':<0> - nok
```

Corresponding class: `Order4<class Relation1, class Relation2, class Relation3, class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class GetValue4=Value, class Stringize=MakeString>` `Relation1`, `Relation2` and `Relation3` can be any of `EqualRelation`, `LessRelation`, `LessOrEqualRelation`, `MoreRelation`, `MoreOrEqualRelation` and `DifferentRelation`.

rbREL_NEAR(a,b,diff), rbvREL_NEAR(a,b,diff)

This macro checks whether its two first arguments are close enough, i.e. if their difference isn't more than the third argument multiplied by the biggest of the absolute values of the two numbers. Upon failure, it converts its arguments to strings, which will be shown in the output. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

This Rich Boolean is to be preferred over `rbNEAR(a,b,diff)` if you expect both small and big values for `a` and `b`, with about the same proportional error.

Example output:

```
relative difference of `a':<10> and `b':<8.99> = 0.101 <= `diff':<0.1> - nok
```

Corresponding class: `RelNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbLESS_REL_NEAR(a,b,diff), rbvLESS_REL_NEAR(a,b,diff)

This macro checks whether the first argument is less than the second, but allowing a tolerance of `diff*max(a,b)`, i.e. if the first is between `b` and `b+diff*max(a,b)`, it is still accepted. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Note that this is *less* strict than `rbLESS`.

Example output:

```
`a':<1.0> <~ `b':<2.0> (relative difference is <0.5>, `diff':<0.1> allowed) - nok
```

Corresponding class: `LessRelativeNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbMORE_REL_NEAR(a,b,diff), rbvMORE_REL_NEAR(a,b,diff)

This macro checks whether the first argument is more than the second, but allowing a tolerance of `diff*max(a,b)`, i.e. if the first is between `b` and `b-diff*max(a,b)`, it is still accepted. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Note that this is *less* strict than `rbMORE`.

Example output:

```
`a':<1.0> >~ `b':<2.0> (relative difference is <0.5>, `diff':<0.1> allowed) - nok
```

Corresponding class: `MoreRelativeNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbDIFF_REL_NEAR(a,b,diff), rbvIFF_REL_NEAR(a,b,diff)

This macro checks whether the first argument is different from the second, requiring a tolerance of `diff*max(a,b)`, i.e. if the first is between `b-diff` and `b+diff`, it is neither accepted. Upon failure, it converts its arguments to strings, which will be shown in the output. The actual difference is also shown. This can only be used with floating point numbers, and objects on which `operator-` is defined and for which the function `fabs` is overloaded.

Note that this is *more* strict than `rbDIFF`.

Example output:

```
`a':<1.0> !=~`b':<2.0> (relative difference is <1>, should be at least`diff':<0.1>
```

Corresponding class: `DiffRelativeNear<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbEQUAL_BITWISE(bits1, bits2), rbvEQUAL_BITWISE(bits1, bits2)

This macro checks whether `bits1` and `bits2` are equal bitwise. They can be of different types, but they must have the same size. Upon failure, it will show which bits are different.

Example output:

```
1: 00001111 00000000 00000000 00000000
C:   XXXX
2: 00110011 00000000 00000000 00000000
```

Note: to display the output of this rich boolean, it is best to use a non-proportional font, so the comparison is easily understood.

Note: the different bytes are shown in the same order as they appear in memory, so the output is different on little endian and big endian machines. In a byte, the most significant bit is shown first, the least significant bit last.

Corresponding class: `EqualBitwise<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbEQUAL_DATA(buf1, buf2, size)

This macro checks whether the buffers `buf1` and `buf2` of length `size` bytes have equal contents. Upon failure, it will show which bytes are different.

Example output:

```
1: 00 15 ff 78
C:       XX
2: 00 15 fe 78
```

Note: to display the output of this rich boolean, it is best to use a non-proportional font, so the comparison is easily understood.

Corresponding class: `EqualData<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbEQUAL_DATA_BITWISE(buf1, buf2, size)

This macro checks whether the buffers `buf1` and `buf2` of length `size` bytes have equal contents. Upon failure, it will show which bits are different.

Example output:

```
1: 00001111 00000000 11111111 00000000
C:   XXXX
2: 00110011 00000000 11111111 00000000
```

Note: to display the output of this rich boolean, it is best to use a non-proportional font, so the comparison is easily understood.

Corresponding class: `EqualDataBitwise<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize=MakeString>`

rbBITS_ON(bits, mask), rbvBITS_ON(bits, mask)

This macro checks whether the bits that are on in `mask`, are also on in `bits`. `bits` and `mask` can be of different types, but they must have the same size. Upon failure, it will show which bits in `bits` are not on, but should be on.

Example output:

```
bits: 00111100 00000000 00000000 00000000
err :       XX
mask: 00001111 00000000 00000000 00000000
```

Note: to display the output of this rich boolean, it is best to use a non-proportional font, so the comparison is easily understood.

Note: the different bytes are shown in the same order as they appear in memory, so the output is different on little endian and big endian machines. In a byte, the most significant bit is shown first, the least significant bit last.

Corresponding class: `BitsOn<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbBITS_OFF(bits, mask), rbvBITS_OFF(bits, mask)

This macro checks whether the bits that are on in `mask`, are off in `bits`. `bits` and `mask` can be of different types, but they must have the same size. Upon failure, it will show which bits in `bits` are not off, but should be off.

Example output:

```
bits: 00001111 00000000 00000000 00000000
err :       XX
mask: 00110011 00000000 00000000 00000000
```

Note: to display the output of this rich boolean, it is best to use a non-proportional font, so the comparison is easily understood.

Note: the different bytes are shown in the same order as they appear in memory, so the output is different on little endian and big endian machines. In a byte, the most significant bit is shown first, the least significant bit last.

Corresponding class: `BitsOff<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rbBITS_ARE(bits, mask), rbvBITS_ARE(bits, mask)

This macro checks whether the bits in `bits` are what is specified in the string `mask`. The string `mask` contains a character `0` where a bit should be off and a character `1` where a bit should be on; a character # means the bit can be both 0 and 1. Bits in the string `mask` can be separated by a space, dot, colon, dash and comma. The number of bits in `bits` and the number of bits specified in `mask` must be the same. Upon failure, it will show which bits in `bits` are not what is expected.

Example:

```
rbBITS_ARE(n, "0##0:1111 0100:#100 0000:0000 0000:0000")
```

Example output:

```
bits: 0011:1100 0100:1000 0000:0000 0000:0000
err :        XX       X
mask: 0##0:1111 0100:#100 0000:0000 0000:0000
```

Note: to display the output of this rich boolean, it is best to use a non-proportional font, so the comparison is easily understood.

Note: the different bytes are shown in the same order as they appear in memory, so the output is different on little endian and big endian machines. In a byte, the most significant bit is shown first, the least significant bit last.

Corresponding class: `BitsAre<class GetValue1=Value, class GetValue2=Value, class Stringize=MakeString>`

rb1_PRED(a,pred), rbv1_PRED(a,pred), rb2_PRED(a,b,pred), rbv2_PRED(a,b,pred), rb3_PRED(a,b,c,pred), rbv3_PRED(a,b,c,pred), rb4_PRED(a,b,c,d,pred), rbv4_PRED(a,b,c,d,pred)

These macros check whether the predicate `pred` returns true if the arguments are given to its method `bool operator()(...)`. If not, it converts the arguments to a string, which will be shown in the output.

`pred` could be a C++0X lambda if your compiler supports them (gcc 4.5 and Visual C++ 2010).

Example output:

```
predicate(`a':<5>) - nok
```

Corresponding classes:

`Pred1<class GetValue1=Value, class Stringize=MakeString>`

`Pred2<class GetValue1=Value, class GetValue2=Value, class Stringize1=MakeString, class Stringize2=MakeString>`

`Pred3<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class Stringize1=MakeString, class Stringize2=MakeString, class Stringize3=MakeString>`

`Pred4<class GetValue1=Value, class GetValue2=Value, class GetValue3=Value, class GetValue4=Value, class Stringize1=MakeString, class Stringize2=MakeString, class Stringize3=MakeString, class Stringize4=MakeString>`

rb1_RB(a,rb), rbv1_1_RB(a,rb), rb2_RB(a,b,rb), rbv2_1_RB(a,b,rb), rbv2_2_RB(a,b,rb), rb3_RB(a,b,c,rb), rbv3_1_RB(a,b,c,rb), rbv3_2_RB(a,b,c,rb), rbv3_3_RB(a,b,c,rb), rb4_RB(a,b,c,d,rb), rb5_RB(a,b,c,d,e,rb), rb6_RB(a,b,c,d,e,f,rb)

These macros check whether the rich boolean functor `rb` evaluates to true if the arguments are given to it. If not, the information in the rich boolean functor will be given.

See the section called “Using Rich Boolean functors in Rich Boolean macros” on how to make a rich boolean functor.

Note: assertion macros that return a value, should use `rbv<n>_<p>_RB(a,b,c,rb)`, where <n> is the number of arguments and <p> is the index of the value that should be returned.

Corresponding classes: none

Note:

The macros in this section are pointersafe. I.e., if you pass them a dereferenced pointer that is `NULL`, it will not be dereferenced, the evaluation will be considered to fail, and the debugging information will tell that a `NULL` pointer was given.

Example 1. Safely using a pointer that could be `NULL`

```
    int *p = 0;
    MOD_ASSERT(rbEQUAL(5, *p));
```

Here there will be no memory read error. Instead the `MOD_ASSERT` will safely fail, and when streamed out, the Rich Boolean will return "`5':<5> == `*p':@0x00000000 - has bad value" (valid results are given between < and >, invalid results start with @). Actually, pointers whose address is up to `0x0000000f`, are treated the same way as a `NULL` pointer. This is handy with classes where pointers with a low address like `0x01`, `0x02`, ..., have a special meaning. Also, this means that referring to a struct or class member of a null pointer often can be done safely, like `p->name`, as long as the member `name` starts before the 16th byte in the struct.

On Windows 32 and above, uninitialized memory at every memory location is detected in Debug mode (using the function `IsBadReadPtr()`), and handled in the same way. If you can use another function that checks for valid addresses, adjust the file `functors.cpp`. Note that functions that only check whether a pointer is a valid heap pointer, cannot be used for this, because the argument of a rich boolean can also be a variable on the stack.

Comparing strings

The Rich Booleans allows to compare strings in many ways, both strings that use char and widecharacter strings.

`rbSTRING(str1, operator, str2, compareObj)`, `rbvSTRING(str1, operator, str2, compareObj)`

`str1` and `str2` are two strings. The allowed types for these strings are determined by `compareObj`. `operator` can be `==`, `<`, `<=`, `>`, `>=` or `!=`. `compareObj` is an object of a suitable class, of which several classes are provided (see below).

When you check for equality, the output shows which characters are different. This uses dynamic matching, so missing characters are easily spotted. When you don't check for equality, only the first character that causes the condition to fail is marked; mismatches after it are not marked.

Corresponding class: `CompareStrings<class CompareCompletely<class CompareType, class Relation=EqualRelation> class GetValue1_=Value, class GetValue2_=Value>`. Here `CompareType` can be any of the classes discussed below. `Relation` can be any of `EqualRelation`, `LessRelation`, `LessOrEqualRelation`, `MoreRelation`, `MoreOrEqualRelation` and `DifferentRelation` (all still in the namespace `RichBool`).

There are two simplified versions of this Rich Boolean:

- `rbSTRING_CS(str1, operator, str2)`: performs a case sensitive comparison
- `rbSTRING_CI(str1, operator, str2)`: performs a case insensitive comparison

These work with both ordinary and wide strings.

`rbSTRING_BEGINS_WITH(str1, str2, compareObj)`, `rbvSTRING_BEGINS_WITH(str1, str2, compareObj)`

This checks if the string `str1` begins with the string `str2`. The allowed types for these strings are determined by `compareObj`. `compareObj` is an object of a suitable class, of which several classes are provided (see below).

Corresponding class: `CompareStrings<class StringBeginsWith<class CompareType> class GetValue1_=Value, class GetValue2_=Value>`. Here `CompareType` can be any of the classes discussed below.

There are two simplified versions of this Rich Boolean:

- `rbSTRING_BEGINS_WITH_CS(str1, operator, str2)`: performs a case sensitive comparison
- `rbSTRING_BEGINS_WITH_CI(str1, operator, str2)`: performs a case insensitive comparison

These work with both ordinary and wide strings.

`rbSTRING_ENDS_WITH(str1, str2, compareObj)`, `rbvSTRING_ENDS_WITH(str1, str2, compareObj)`

This checks if the string `str1` ends with the string `str2`. The allowed types for these strings are determined by `compareObj`. `compareObj` is an object of a suitable class, of which several classes are provided (see below).

Corresponding class: `CompareStrings<class StringEndsWith<class CompareType> class GetValue1_=Value, class GetValue2_=Value>`. Here `CompareType` can be any of the classes discussed below.

There are two simplified versions of this Rich Boolean:

- `rbSTRING_ENDS_WITH_CS(str1, operator, str2)`: performs a case sensitive comparison
- `rbSTRING_ENDS_WITH_CI(str1, operator, str2)`: performs a case insensitive comparison

These work with both ordinary and wide strings.

`rbSTRING_CONTAINS(str1, str2, compareObj)`, `rbvSTRING_CONTAINS(str1, str2, compareObj)`

This checks if the string `str1` contains the string `str2`. The allowed types for these strings are determined by `compareObj`. `compareObj` is an object of a suitable class, of which several classes are provided (see below).

Corresponding class: `CompareStrings<class StringContains<class CompareType> class GetValue1_=Value, class GetValue2_=Value>`. Here `CompareType` can be any of the classes discussed below.

There are two simplified versions of this Rich Boolean:

- `rbSTRING_CONTAINS_CS(str1, operator, str2)`: performs a case sensitive comparison
- `rbSTRING_CONTAINS_CI(str1, operator, str2)`: performs a case insensitive comparison

These work with both ordinary and wide strings.

Classes to use as the last argument in string comparisons

Note: the string comparisons that use collation, don't take care well of characters that are ignored in a collation. Dynamic matching often matches the surrounding characters, so ignored characters are shown as a superfluous character (or a mismatch), but it is up to you to know which are the false negatives. When you compare strings with collation but not for equality, the comparison therefor doesn't stop at the first offending character, because it could be an ignored character.

Comparing character pointers

In the file `richbool/string.hpp` there is the template class `BasicStrcmp`, that has one template argument, the character type. You can use two specializations, `BasicStrcmp<char>` that uses `strcmp()`, (typedef'ed as `Strcmp`) and `BasicStrcmp<wchar>` that uses `wcscmp()` (typedef'ed as `Wcscmp`). These have a constructor that takes no arguments. These can only have character pointers as arguments, no string objects. For convenience, the Rich Booleans `rbSTRCMP(str1, operator, str2)`, `rbWCSCMP(str1, operator, str2)`, `rbvSTRCMP(str1, operator, str2)` and `rbvWCSCMP(str1, operator, str2)` are defined, that are equivalent to `rbSTRING(str1, operator, str2, RichBool::Strcmp())`, `rbSTRING(str1, operator, str2, RichBool::Wcscmp())`, `rbvSTRING(str1, operator, str2, RichBool::Strcmp())` and `rbvSTRING(str1, operator, str2, RichBool::Wcscmp())` respectively.

Comparing std::string objects

In the file `richbool/stdstring.hpp` there is the template class `BasicStringCompare`, that compares `std::string` objects and/or character pointers using `strcmp` or `wcscmp`, depending on the character type. `BasicStringCompareToUpper` and `BasicStringCompareToLower` do the same, but first make a copy of the strings and convert these copies to respectively uppercase and lowercase. Those three template classes respectively have typedefs `StringCompare`, `StringCompareToUpper` and `StringCompareToLower` for `char`, and `WStringCompare`, `WStringCompareToUpper` and `WStringCompareToLower` for `wchar_t`.

For convenience, there are the Rich Booleans `rbSTD_STRING(str1, operator, str2)` and `rbSTD_WSTRING(str1, operator, str2)` that are respectively equivalent to `rbSTRING(str1, operator, str2, RichBool::StringCompare())` and `rbSTRING(str1, operator, str2, RichBool::WStringCompare())`. So what they check is equivalent to using the operators directly on the strings.

For convenience, there are the following typedefs that make comparisons in ranges easier:

- typedef CompareStrings<CompareCompletely<StringCompare, EqualRelation>, GetStringValue, GetStringValue> StringsEqualCS;
- typedef CompareStrings<CompareCompletely<WStringCompare, EqualRelation>, GetStringValue, GetStringValue> WStringsEqualCS;
- typedef CompareStrings<CompareCompletely<StringCompareToUpper, EqualRelation>, GetStringValue, GetStringValue> StringsEqualCI;

- typedef CompareStrings<CompareCompletely<WStringCompareToUpper, EqualRelation>, GetStringValue, GetStringValue> WStringsEqualCI;

This allows you to e.g. compare strings in two std::vector objects with std::string objects in them, with

```
MOD_ASSERT(
  rbIN_CONTAINERS(vec1, vec2, RichBool::Compare<>().That(RichBool::StringsEqualCS(
);
```

.

Comparing std::string objects with collation

In the same file `richbool/stdstring.hpp` there are also the template classes `BasicCollate`, `BasicCollateToUpper` and `BasicCollateToLower`. They have one template argument, the character type. These have a constructor that has a locale as argument, for which the global locale is the default. `BasicCollate` collates strings using the `compare` method of the given locales `collate` facet. `BasicCollateToUpper` and `BasicCollateToLower` also do this, but first make a copy of the strings, and convert these copies to upper case and lower case respectively, using the `toupper` and `tolower` methods of the given locales `ctype` facet.

Note that converting to uppercase or lowercase only behaves differently if the strings contain characters between 'Z' and 'a', and when you don't check for equality.

Comparing wxString objects

In the file `richbool/wxstring.hpp` there are the classes `WxStringCmp` and `WxStringCmpNoCase`, that compare `wxString` objects and/or character pointers. These have a constructor that takes no arguments. The first one compares strings using the `Cmp` method of `wxString`, the second uses the `CmpNoCase` method of `wxString`. There is also the Rich Boolean macro `rbWX_STRING(str1, op, str2)`, which is equivalent to `rbSTRING(str1, op, str2, RichBool::WxStringCmp())`

Comparing MFC CString objects

There are similar classes for MFC `CString` objects. In the file `richbool/mfcstring.hpp` there are the classes `CStringCompare`, `CStringCompareNoCase`, `CStringCollate` and `CStringCollateNoCase`, that compare `CString` objects and/or character pointers. These have a constructor that takes no arguments. They respectively compare strings using the methods `Compare`, `CompareNoCase`, `Collate` and `CollateNoCase` methods of `CString`. With Visual Studio .NET 2003 and later, you can also use `CStringACompare`, `CStringACompareNoCase`, `CStringACollate`, `CStringACollateNoCase`, `CStringWCompare`, `CStringWCompareNoCase`, `CStringWCollate` and `CStringWCollateNoCase`, which use the same methods on `CStringA` and `CStringW` objects. In template code you can use the template classes `TmplCStringCompare<T>`, `TmplCStringCompareNoCase<T>`, `TmplCStringCollate<T>` and `TmplCStringCollateNoCase<T>`, of which the previous ones are typedefs.

There is also the Rich Boolean macro `rbCSTRING(str1, op, str2)`, which is equivalent to `rbSTRING(str1, op, str2, RichBool::CStringCompare())`

Example usage:

```
    String str1 = "abc";
    MOD_ASSERT(rbSTRING(str1, ==, "abd", RichBool::StringCompare()));
```

Example output:

```
`str1':<abc> == `"abd"':<abd> (locale C) - nok
str1: abc
diff:   X
str2: abd
```

Example usage:

```
    String str1 = "abc", str2 = "ABÉ";
    MOD_ASSERT(rbSTRING(str1, >, str2, RichBool::CollateToUpper(std::locale("fr"))
```

Example output:

```
`str1':<abc> > `str2':<ABÉ> (locale French_France.1252 toupper) - nok
str1: ab   c
diff:      X
str2: ab\xc9
```

The next four macros are only available on compilers that support C++0X regular expressions. At the moment of writing this works with Visual C++ 2010, but not with gcc.

rbREGEXP(str, regex), rbvREGEXP(str, regex)

This checks if the string `str` matches the regular expression in the string `regex` (of the ECMAScript type). The strings `str` and `regex` can be of the `char` or `wchar` type, as long as they are the same type.

If the condition fails, the regular expression is split up in pieces in the analysis, and the analysis shows which pieces could be matched to which pieces of the regular expression.

Corresponding class: `RegExp<class GetValue1_=Value, class GetValue2_=Value>`

rbREGEXP_F(str, regex, flags), rbvREGEXP_F(str, regex, flags)

This checks if the string `str` matches the regular expression in the string `regex` (of the ECMAScript type), using the flags in `flags` with `std::regex_search`. The strings `str` and `regex` can be of the `char` or `wchar` type, as long as they are the same type.

If the condition fails, the analysis is similar to the one created by `rbREGEXP`.

Corresponding class: `RegExp<class GetValue1_=Value, class GetValue2_=Value>`

rbHAS_REGEXP(str, regex), rbvHAS_REGEXP(str, regex)

This checks if the string `str` contains a substring that matches the regular expression in the string `regex` (of the ECMAScript type). The strings `str` and `regex` can be of the `char` or `wchar` type, as long as they are the same type.

If the condition fails, the regular expression is split up in pieces in the analysis, and the analysis shows which pieces could be matched to which pieces of the regular expression.

Corresponding class: `HasRegExp<class GetValue1_=Value, class GetValue2_=Value>`

`rbHAS_REGEXP_F(str, regex, flags)`, `rbvHAS_REGEXP_F(str, regex, flags)`

This checks if the string `str` contains a substring that matches the regular expression in the string `regex` (of the ECMAScript type), using the flags in `flags` with `std::regex_search`. The strings `str` and `regex` can be of the `char` or `wchar` type, as long as they are the same type.

If the condition fails, the analysis is similar to the one created by `rbHAS_REGEXP`.

Corresponding class: `HasRegExp<class GetValue1_=Value, class GetValue2_=Value>`

Filesystem

`rbFILE(file, func)`, `rbvFILE(file, func)`

`file` is the name of a file, a file descriptor or a `FILE` pointer, `func` specifies the condition(s) that the file should fulfill. If the file does not exist, is a directory or does not fulfill the requirements, the Rich Boolean fails, otherwise it succeeds. See below for what `func` can be. If the first argument is a filename, then on Windows it can be a basic character string or a wide character string (except with the Cygwin compiler); on POSIX systems it can only be a basic character string (because POSIX doesn't have wide character filenames).

Example usage:

```
    MOD_ASSERT(rbFILE(filename, RichBool::IsWritable()));
```

Example output 1a (Windows):

```
file `filename':<results.txt> should be writable - nok
```

Example output 1b (Posix):

```
file `filename':<results.txt> should be writable - nok
  -r--r----- 1 mark(1000)+ users(100)+;
  "process user and group: mark(1000) users(100)
```

Example output 2:

```
file `filename':<results> should be writable - ? - nok
is a directory
```

Note that on Posix systems there is an epilogue that shows the file type and the permissions of the file, in almost the same way as the command `ls` does when the `-l` option is given. The name of the user is followed by a plus if the process runs with that id as its real user id, a minus otherwise. The name of the group is followed by a plus if the process runs with a real user id that belongs to that group, a minus otherwise. If the process runs as a superuser, these are both replaced by asterisks.

Corresponding class: `File<class Func, class GetValue1=Value>`

`rbDIRECTORY(name, func)`, `rbvDIRECTORY(name, func)`

`name` is the name of a directory, `func` specifies the checks that the directory should fulfill. If the directory does not exist, is a file or does not fulfill the requirements, the Rich Boolean fails, otherwise it succeeds. See below for what `func` can be. On Windows the name of the directory can be a basic character string or a wide character string (except with the Cygwin compiler); on POSIX systems it can only be a basic character string (because POSIX doesn't have wide character directory names).

Example usage:

```
    MOD_ASSERT(rbDIRECTORY(name, RichBool::IsWritable()));
```

Example output 1:

```
directory `name':<results> should be writable - nok
```

Example output 2:

```
directory `name':<results.txt> should be writable - ? - nok
is a file
```

Corresponding class: `Directory<class Func, class GetValue1=Value>`

Possible checks for `rbFILE`, `rbvFILE`, `rbDIRECTORY` and `rbvDIRECTORY`

The following table lists the classes that you can use in `rbFILE`, `rbvFILE`, `rbDIRECTORY` and `rbvDIRECTORY`, and whether you can use them with a filename, a file descriptor or a directory, and whether it can be used on Windows.

Table 1. Possible checks for `rbFILE`, `rbvFILE`, `rbDIRECTORY` and `rbvDIRECTORY`

Class	Filename	File descriptor / FILE pointer	Directory	Windows
IsReadable	Y	Y	Y	Y
IsWritable	Y	Y	Y	Y
IsExecutable	Y	Y	Y	Y
IsPipe	N	Y	N	Y
IsLink	Y	Y	Y	N
IsRegular	Y	Y	N	N
IsCharacterDevice	Y	Y	N	N
IsBlockDevice	Y	Y	N	N

Note: on Windows a file or a directory is always readable (unless it is in a directory that is not accessible to the process).

Note: on Windows a file is considered executable if its extension is that of an executable file, like exe, com and bat. On Linux a file is considered executable if its mode says so.

Note: on Windows a directory is always considered executable. On Linux a directory is considered executable if its mode says so, meaning that the process can see the contents of the directory.

Note: because these are Rich Boolean functors, you can also negate them with `operator!` and combine them with `operator&`.

```
    // check that the file is not writable:
    MOD_ASSERT(rbFILE(filename, !RichBool::IsWritable()));

    // check that the file is readable and writable:
    MOD_ASSERT(
        rbFILE(filename, RichBool::IsReadable()&RichBool::IsWritable())
    );
```

You could also use `operator|`, `operator||` and `operator^` to combine them, but that usually is not needed. You could also use `operator&&` to combine them, but then you would only see the result of the conditions up to the first one that failed.

Note: if you pass a file descriptor to `rbFILE` or `rbvFILE`, the checks test the properties that the file has, not the way that you opened the file. So if you open a writable file in readonly mode, a check using the file descriptor will tell that the file is writable, just as if the filename was used.

User and group id on Linux

On Linux the permissions to use a file or directory are determined using the file system user id and group id. These are the same as the effective user id and group id, except when they were set to a different value with `setfsuid` and `setfsgid`. Because these can not be queried, the Rich Booleans library cannot know these. If you call these functions, you should therefore call `void SetFsUid(uid_t fsuid)` and/or `void SetFsGid(uid_t fsgid)` (both in the namespace `RichBool`) with the same values, so the Rich Booleans can perform these checks correctly. If you don't call these, the Rich Booleans library uses `geteuid` and `getegid`, i.e. the effective user id and group id.

Note that it is unlikely that you will need this, and that this is only needed when using file descriptors and pointers to `FILE` objects.

`rbFILE_EXISTS(file)`, `rbvFILE_EXISTS(file, func)`

`file` is the name of a file. If the file does not exist, or is a directory, the Rich Boolean fails, otherwise it succeeds. On Windows the filename can be a basic character string or a wide character string; on POSIX systems it can only be a basic character string (because POSIX doesn't have wide character filenames).

Example usage:

```
    MOD_ASSERT(rbFILE_EXISTS(filename));
```

Example outputs:

```
file `filename':<results.txt> should exist - nok
 is a directory
```

Corresponding class: `FileExists<class GetValue1=Value>`

`rbDIRECTORY_EXISTS(name)`, `rbvDIRECTORY_EXISTS(name)`

`name` is the name of a directory. If the directory does not exist, or is a file, the Rich Boolean fails, otherwise it succeeds. On Windows the name of the directory can be a basic character string or a wide character string (except with the Cygwin compiler); on POSIX systems it can only be a basic character string (because POSIX doesn't have wide character directory names).

Example usage:

```
    MOD_ASSERT(rbDIRECTORY_EXISTS(name));
```

Example outputs:

```
directory `name':<results.txt> should exist - nok
 is a file
```

Corresponding class: `DirectoryExists<class GetValue1=Value>`

`rbDOES_NOT_EXIST(name)`, `rbvDOES_NOT_EXIST(name)`

`name` is a name. If a file or directory with that name exists, the Rich Boolean fails, otherwise it succeeds. On Windows the name can be a basic character string or a wide character string (except with the Cygwin compiler); on POSIX systems it can only be a basic character string (because POSIX doesn't have wide character filenames).

Example usage:

```
    MOD_ASSERT(rbDOES_NOT_EXIST(name));
```

Example outputs:

```
`name':<results.txt> should not exist - nok
 is a file
```

Corresponding class: `DoesNotExist<class GetValue1=Value>`

`GetFileLength`

`GetFileLength` is a function that takes one argument, a filename in a `const char *`, `const wchar_t *`, `std::string` or a `std::wstring`, and returns the length of the file (on Linux the filename can only be `const char *` or `std::string`).

If the file does not exist or is a directory, -1 is returned.

Example usage:

```
    MOD_ASSERT(rbLESS(RichBool::GetFileLength(filename), 512));
```

The Rich Booleans in this section are very useful in combination with the test directories you can make in UquoniTest:

```
uqtTEST(ProcessDirectory)
{
    UquoniTest::UseDirectory dir;
    dir.MakeFile("file1.txt", "1 2");
    dir.MakeFile("file2.txt", "3 7");
    ProcessDirectory(dir); // conversion to std::string

    uqtASSERT(rbFILE_EXISTS(dir/"results.txt"));
}
```

Type checking with RTTI

Note: the rich booleans in this section only work when RTTI is enabled in your application (RTTI doesn't have to be enabled when the Rich Booleans library itself is built).

Warning: make sure you don't include the headerfile `typeinfo.h` in files where you use Rich Booleans. That file contains older versions of classes that are in the headerfile `typeinfo`, which is included by the Rich Booleans package. Including both leads to conflicts.

`rbEQUAL_TYPES(pobj1, pobj2)`, `rbvEQUAL_TYPES(pobj1, pobj2)`

`pobj1` and `pobj2` are two pointers to polymorph objects. The Rich Boolean only succeeds if they have the same type, which is checked at runtime. If they have different types, an `Analysis` object is created that tells what the types are.

Example usage:

```
    A *a1 = new B;
    A *a2 = new C;
    MOD_ASSERT(rbEQUAL_TYPES(a1, a2));
```

Example output:

```
typeid(*`a1'):<class B> == typeid(*`a2'):<class C> - nok
```

Corresponding class: `EqualTypes<class GetValue1=Value, class GetValue2=Value>`

This is especially useful when you have methods to clone polymorph objects. Add a non-virtual `clone` method in the base class, that calls a protected virtual method `do_clone` that does the actual cloning, and check if the types are equal after that call. If the `do_clone` method is forgotten in a child class, it will be noticed at runtime.

`rbHAS_TYPE(pobj, type)`, `rbvHAS_TYPE(pobj, type)`

`pobj` is a pointer to a polymorph object. The Rich Boolean only succeeds if it has the given type, which is checked at runtime. If it has a different type, an `Analysis` object is created that tells what the types are.

Example usage:

```
    A *a = new B;
    MOD_ASSERT(rbHAS_TYPE(a, C));
```

Example output:

```
typeid(*`a'):<class B> == typeid(`C'):<class C> - nok
```

Corresponding class: `HasType<typename Type, class GetValue1=Value>`

rbDYNAMIC_CASTABLE(obj, type), rbvDYNAMIC_CASTABLE(obj, type)

`obj` is a polymorph object. The Rich Boolean only succeeds if the object can be casted to the given type with `dynamic_cast`, which is checked at runtime. If it cannot be casted dynamically, an `Analysis` object is created that tells what the types are. The argument types are the same as for `dynamic_cast`, i.e. a pointer to an object and a pointer type, or a reference to an object and a reference type.

Example usage:

```
    A *a = new B;
    MOD_ASSERT(rbDYNAMIC_CASTABLE(*a1, C&));
    MOD_ASSERT(rbDYNAMIC_CASTABLE(a1,  C*)); // equivalent
```

Example output:

```
typeid(`*a'):<class B> -> typeid(`C&'):<class C&> - nok
```

Corresponding class: `DynamicCastable<typename Type, class GetValue1=Value>`

Note: if your compiler can't do partial specialization of templates, you can't use this Rich Boolean. Instead, use `rbDYNAMIC_CASTABLE_PTR(obj, type)` for pointers, or `rbDYNAMIC_CASTABLE_REF(obj, type)` for references. Their corresponding classes respectively are `DynamicCastablePointer<typename Type, class GetValue1=Value>` and `DynamicCastableReference<typename Type, class GetValue1=Value>`.

Type checking with wxWidgets

wxWidgets provides a runtime typechecking system for classes that are derived from `wxObject`. For such classes, there are Rich Booleans that do runtime typechecking, equivalent to the ones for real RTTI.

rbWX_EQUAL_TYPES(pobj1, pobj2), rbvWX_EQUAL_TYPES(pobj1, pobj2)

`pobj1` and `pobj2` are two pointers to polymorph objects. The Rich Boolean only succeeds if they have the same type, which is checked at runtime. If they have different types, an `Analysis` object is created that tells what the types are.

Example usage:

```
    A *a1 = new B;
    A *a2 = new C;
    MOD_ASSERT(rbWX_EQUAL_TYPES(a1, a2));
```

Example output:

```
typeid(*`a1'):<B> == typeid(*`a2'):<C> - nok
```

Corresponding class: `WxEqualTypes<class GetValue1=Value, class GetValue2=Value>`

This is especially useful when you have methods to clone polymorph objects. Add a non-virtual `clone` method in the base class, that calls a protected virtual method `do_clone` that does the actual cloning, and check if the types are equal after that call. If the `do_clone` method is forgotten in a child class, it will be noticed at runtime.

`rbWX_HAS_TYPE(pobj, type)`, `rbvWX_HAS_TYPE(pobj, type)`

`pobj` is a pointer to a polymorph object. `type` is either a pointer to an object of the type `wxClassInfo`, or a `const wxChar *` string that contains the name of the class. The Rich Boolean only succeeds if it has the given type, which is checked at runtime. If it has a different type, an `Analysis` object is created that tells what the types are.

Example usage:

```
    A *a = new B;
    MOD_ASSERT(rbWX_HAS_TYPE(a, "C"));
```

Example output:

```
typeid(*`a'):<class B> == typeid(`C'):<class C> - nok
```

Corresponding class: `WxHasType<class GetValue1=Value>`

`rbWX_IS_KIND_OF(obj, type)`, `rbvWX_IS_KIND_OF(obj, type)`

Here `obj` is a pointer to a polymorph object. `type` is either a pointer to an object of the class `wxClassInfo`, or a `const wxChar *` string that contains the name of the class. The Rich Boolean only succeeds if the object can be casted to the given type, which is checked with the method `IsKindOf`. If it cannot be casted, an `Analysis` object is created that tells what the types are.

Example usage:

```
    A *a = new B;
    MOD_ASSERT(rbWX_IS_KIND_OF(a, "C"));
```

Example output:

```
typeid(`*a'):<B> -> typeid(`C&'):<C&> - nok
```

Corresponding class: `WxIsKindOf<class GetValue1=Value>`

Logical expressions with Rich Booleans

To use the Rich Booleans in this section, include `richbool/richbool.hpp`

rbOR(cond1, cond2)

`cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. Evaluation uses shortcut logic, so it evaluates `cond1`, and if and only if this is false, also `cond2`. If `cond2` is also false, an `Analysis` object is created that contains the `Analysis` objects of both conditions.

Example output:

```
condition 1:
  `a':<2> == `b':<1> - nok
condition 2:
  `a':<3> >= `c':<4> - nok
```

Corresponding class: none

rbAND(cond1, cond2)

`cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. Evaluates `cond1`, and if this is true, also `cond2`. If the first is false, a `Analysis` object is created that contains the `Analysis` objects of the first condition. If the first is true and the second is false, an `Analysis` object is created with the `Analysis` objects of both conditions.

Example output:

```
condition 1:
  `a':<1> == `b':<1> - ok
condition 2:
  `a':<3> >= `c':<4> - nok
```

Example output:

```
condition 1:
  `a':<2> == `b':<1> - nok
condition 2:
  not evaluated
```

Corresponding class: none

rbXOR(cond1, cond2)

`cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. It evaluates `cond1` and `cond2`, and succeeds if one of the conditions is true and one is false. If they both are true, or they both are false, an `Analysis` object is created that contains the `Analysis` objects of the two conditions.

Example output:

```
condition 1:
  `a':<1> == `b':<1> - ok
condition 2:
  `a':<1> <= `c':<1> - ok
```

Corresponding class: none
Note that `rbOR` and `rbAND` can't be nested. If you try to, you will get a compile error, which was introduced on purpose. Otherwise, nesting `rbOR` and `rbAND` would give false results. This has to do with the complex task of maintaining the short circuit logic without losing the `Analysis` object. The next four Rich Booleans remedy that, but they have drawbacks. You most likely won't need them, since nesting `rbOR` and `rbAND` is rarely needed. Note that this problem is non-existent with `rbXOR`, it can be used inside `rbOR` and `rbAND`.

rbOR_DE(cond1, cond2)

The suffix _DE here stands for "double evaluation". `cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. Unlike `rbOR`, it can be used inside `rbOR` and `rbAND`. It can however, evaluate the conditions twice, which may not be desirable, but `cond2` will only be evaluated if `cond1` is false.

Example output: see rbOR

Corresponding class: none

rbOR_BE(cond1, cond2)

The suffix _BE here stands for "both evaluated". `cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. Unlike `rbOR`, it can be used inside `rbOR` and `rbAND`. It will however, evaluate both conditions, even if the first one is true, which may not be desirable.

Example output: see rbOR

Corresponding class: none

rbAND_DE(cond1, cond2)

The suffix _DE here stands for "double evaluation". `cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. Unlike `rbAND`, it can be used inside `rbOR` and `rbAND`. It can however, evaluate the conditions twice, which may not be desirable, but `cond2` will only be evaluated if `cond1` is true.

Example output: see rbAND

Corresponding class: none

rbAND_BE(cond1, cond2)

The suffix _BE here stands for "both evaluated". `cond1` and `cond2` can be Rich Booleans (except for `rbOR` and `rbAND`) or booleans. Unlike `rbAND`, it can be used inside `rbOR` and `rbAND`. It will however, evaluate both conditions, even if the first one is false, which may not be desirable.

Example output: see rbAND

Corresponding class: none

Exceptions

`rbEXCEPTION(exc)`

This macro gives information about the exception `exc`. Include the file `richbool/exceptions.hpp` if you need it. This Rich Boolean is peculiar, because it always fails, as an exception that was thrown is always an error. To use it with a certain type of exception, there should be two functions overloaded in the `RichBool` namespace for the type:

- `GetExceptionTypeName(ExceptionType &)`: this should return a string with the type name
- `GetExceptionInfo(ExceptionType  &)`: this should return a string with the info in the exception

Example 2. Defining overloaded functions to use `rbEXCEPTION(exc)`

```
namespace RichBool {
 inline const char* GetExceptionTypeName(const MyException &) {
  return "MyException";
 }
 inline std::string GetExceptionInfo(const MyException &exc) {
  return exc.info();
 }
}
```

You can use the macro `RICHBOOL_MAKE_EXCEPTION_TYPE_NAME` to define the function `GetExceptionTypeName(ExceptionType &)`, so the code is simplified to

Example 3. Defining overloaded functions to use `rbEXCEPTION(exc)`

```
RICHBOOL_MAKE_EXCEPTION_TYPE_NAME(MyException);
namespace RichBool {
 inline std::string GetExceptionInfo(const MyException &exc) {
  return exc.info();
 }
}
```

If you have a class hierarchy of exceptions, with a base class that has a (virtual) method to give information about the exeption, it usually suffices to overload `GetExceptionInfo(ExceptionType &)` for the base class, and use the macro `RICHBOOL_MAKE_EXCEPTION_TYPE_NAME` for the derived classes.

Corresponding class: `RichBool::Exception`
This Rich Boolean is useful in a catch block, because it lets you easily fire an assertion:

Example usage:

```
try {
    ...
}
catch (MyException &exc) {
    MOD_ASSERT(rbEXCEPTION(exc));
    ...
}
```

STD exceptions

`GetExceptionTypeName(ExceptionType &)` and `GetExceptionInfo(ExceptionType &)` are already overloaded for all the exceptions that are defined in the C++ standard. To use these overloads, include the file `richbool/stdexceptions.hpp`.

MFC exceptions

`GetExceptionTypeName(ExceptionType &)` and `GetExceptionInfo(ExceptionType &)` are already overloaded for all MFC exceptions. To use these overloads, include the file `richbool/mfcexceptions.hpp`.

Checking a range

To use the Rich Booleans in this section, include `richbool/richbool.hpp`.

`rbIN_RANGE(begin, end, check)`

This macro gives begin and end to the rich boolean functor `check`.

Corresponding class: none

`rbIN_ARRAY(array, check)`

This macro gives the begin and end of the array to the rich boolean functor `check`. It is equivalent to `rbIN_RANGE(array, sizeof(array)/sizeof(array[0]), check)`. The variable `array` should be an array, not a pointer.

Corresponding class: none

`rbIN_VALUES(initializer_list, check)`

This macro is still experimental. It is only available on compilers that support C++0X initializer lists. At the moment of writing this works on gcc 4.5, but not with Visual C++.

This macro gives the begin and end of the initializer list to the rich boolean functor `check`. The initializer list should be enclosed in parens.

Corresponding class: none

Example 4.

```
rbIN_VALUES(({ 2, 4, 8}), Sorted<>())
```

Available Rich Boolean functor classes for checking a range

There are several Rich Boolean functor classes provided that work on a range, and thus can be given as the last argument to Rich Boolean macros that work on a range, like `rbIN_RANGE`. Some of these can be used directly, others are actually Rich Boolean functor factories, i.e. they create rich boolean functors on the fly, when you give a Rich Boolean functor to their factory method. It should be a Rich Boolean functor that takes one argument. See the section called “Using Rich Boolean functors in Rich Boolean macros” on how to make a rich boolean functor.

Sorted

The template class `Sorted<bool multiPass=true>` takes two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. It checks whether the given range is sorted. The template argument tells whether the range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Its default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are ok) or only the failing ones; the default value is false (i.e. only show the failing ones).

Example 5. Using Sorted to check the elements of a range

```
    using namespace RichBool;
    int array[] = { 1, 5, 3, 6 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, Sorted<>()));
```

The last line can be read as "assert that the range [array, array+4) is sorted".

Output:

```
range is sorted: `array'-`array+4' - nok
X: [1]:<5> <= [2]:<3> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

Example 6. Using Sorted to check the elements of a range, also showing the good ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, array+4, Sorted<>(true)));
```

Output:

```
range is sorted: `array'-`array+4' - nok
M: [0]:<1> <= [1]:<5> - ok
X: [1]:<5> <= [2]:<3> - nok
M: [2]:<3> <= [3]:<6> - ok
```

An 'M' at the begin of a line indicates success, an 'X' a failure.

Example 7. Using Sorted to check the elements of a range, using the number of elements

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, 4, Sorted<>()));
```

Output:

```
range is sorted: `array'-`4' - nok
X: [1]:<5> <= [2]:<3> - nok
```

SortedStrictly

The template class `SortedStrictly<bool multiPass=true>` takes two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. It checks whether the given range is sorted strictly (i.e. sorted and no two successive elements are equal). The template argument tells whether the range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Its default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are ok) or only the failing ones; the default value is false (i.e. only show the failing ones).

Example 8. Using SortedStrictly to check the elements of a range

```
    using namespace RichBool;
    int array[] = { 1, 3, 3, 6 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, SortedStrictly<>()));
```

The last line can be read as "assert that the range [array, array+4) is sorted strictly".

Output:

```
range is sorted strictly: `array'-`array+4' - nok
X: [1]:<3> < [2]:<3> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

Example 9. Using SortedStrictly to check the elements of a range, also showing the good ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, array+4, SortedStrictly<>(true)));
```

Output:

```
range is sorted strictly: `array'-`array+4' - nok
M: [0]:<1> < [1]:<3> - ok
X: [1]:<3> < [2]:<3> - nok
M: [2]:<3> < [3]:<6> - ok
```

An 'M' at the begin of a line indicates success, an 'X' a failure.

AllUnique

The class `AllUnique` takes two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. It checks whether the elements in the given range are all unique. The range should be multi-pass (see the section called "Multi-pass and single-pass ranges"). This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the comparisons (even the ones that are ok) or only the failing ones; the default value is false (i.e. only show the failing comparisons).

Example 10. Using AllUnique to check the elements of a range

```
    using namespace RichBool;
    int array[] = { 1, 4, 6, 4 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, AllUnique()));
```

The last line can be read as "assert that the elements in the range [array, array+4) are all unique".

Output:

```
all elements are unique in range: `array'-`array+4' - nok
X: [1]:<4> != [3]:<4> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

An 'M' at the begin of a line indicates success, an 'X' a failure.

All

The template class `All<bool multiPass=true>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `Are` creates, check whether the given Rich Boolean functor applies to *all* elements in the given range. These objects take two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. The template argument tells whether the range is single pass or multi-pass (see the section called "Multi-pass and single-pass ranges"). Its default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones for which it is ok) or only the failing ones; the default value is false (i.e. only show the failing ones).

To create a Rich Boolean functor, pass a Rich Boolean functor to its method `Are`.

Example 11. Using All to check the elements of a range

```
    using namespace RichBool;
    struct IsEven {
        bool operator()(int n) const { return n%2==0; }
    };
    int array[] = { 4, 6, 7, 10 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, All<>().Are(Pred1<IsEven>())));
```

The last line can be read as "assert that in the range [array, array+4) all elements are even".

Output:

```
predicate applies to all elements in range `array'-`array+4' - nok
X: predicate([2]:<7>) - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

Example 12. Using All to check the elements of a range, also showing the good ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, array+4, All<>(true).Are(Pred1<IsEven>())));
```

Output:

```
predicate applies to all elements in range `array'-`array+4' - nok
M: predicate([0]:<4>) - ok
M: predicate([1]:<6>) - ok
X: predicate([2]:<7>) - nok
M: predicate([3]:<10>) - ok
```

An 'M' at the begin of a line indicates success, an 'X' a failure.

Example 13. Using All to check the elements of a range, using the number of elements to use

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, 4, All<>().Are(Pred1<IsEven>())));
```

Output:

```
predicate applies to all elements in range `array'-`4' - nok
X: predicate([2]:<7>) - nok
```

Has

The template class `Has<bool multiPass=true>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `That` creates, check whether the given Rich Boolean functor applies to *at least one element* in the given range. These functors take two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. The template argument tells whether the range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Its default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range or just tell that there was no element for which the rich boolean passes; the default value is false (i.e. don't show the elements).

Example 14. Using Has to check the elements of a range

```
    using namespace RichBool;
    struct IsEven {
        bool operator()(int n) const { return n%2==0; }
    };
    int array[] = { 3, 5, 7, 9 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, Has<>().That(Pred1<IsEven>())));
```

The last line can be read as "assert that the range [array, array+4) has elements that are even".
Output:

```
predicate applies to all elements in range `array'-`array+4' - nok
```

Example 15. Using Has to check the elements of a range, showing all the elements

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, array+4, Has<>(true).That(Pred1<IsEven>())));
```

Output:

```
predicate applies to all elements in range `array'-`array+4' - nok
X: predicate([0]:<3>) - nok
X: predicate([1]:<5>) - nok
X: predicate([2]:<7>) - nok
X: predicate([3]:<9>) - nok
```

Unique

The template class `Unique<bool multiPass=true>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `That` creates, check whether the given Rich Boolean functor applies to *exactly one element* in the given range. These functors take two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. The template argument tells whether the range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Its default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range or only the ones for which the rich boolean passes; the default value is false (i.e. only show the succeeding ones).

Example 16. Using Unique to check the elements of a range

```
    using namespace RichBool;
    struct IsEven {
        bool operator()(int n) const { return n%2==0; }
    };
    int array[] = { 3, 6, 7, 10 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, Unique<>().That(Pred1<IsEven>())));
```

The last line can be read as "assert that in the range [array, array+4) there is a unique element that is even".

Output:

```
predicate applies to exactly one element in range `array'-`array+4' - nok
M: predicate([1]:<6>) - ok
M: predicate([3]:<10>) - ok
```

Note that this is different from the previous ones: now the elements for which the rich boolean passes, are shown. Because there should be only one, and there are two, these are shown.

Example 17. Using Unique to check the elements of a range, also showing the good ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, array+4, Unique<>(true).That(Pred1<IsEven>())));
```

Output:

```
predicate applies to exactly one element in range `array'-`array+4' - nok
X: predicate([0]:<3>) - nok
M: predicate([1]:<6>) - ok
X: predicate([2]:<7>) - nok
M: predicate([3]:<10>) - ok
```

An 'M' at the begin of a line indicates success, an 'X' a failure (although the real failure here is that there is more than one element that doesn't 'fail').

Adjacent

The template class `Adjacent<bool multiPass=true>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `Are` creates check whether the given Rich Boolean functor applies to *all adjacent elements* in the given range. These functors take two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. The template argument tells whether the range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Its default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones for which it is ok) or only the failing ones; the default value is false (i.e. only show the failing ones).

Example 18. Using Adjacent to check the elements of a range

```
    using namespace RichBool;
    int array[] = { 10, 6, 8, 4 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, Adjacent<>().Are(More<>())));
```

The last line can be read as "assert that in the range [array, array+4) for every two adjacent elements, the first is more than the second".

Output:

```
predicate applies to adjacent elements in range `array'-`array+4' - nok
X: [1]:<6> > [2]:<8> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

Example 19. Using Adjacent to check the elements of a range, also showing the good ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGE(array, array+4, Adjacent<>(true).Are(More<>())));
```

Output:

```
predicate applies to adjacent elements in range `array'-`array+4' - nok
M: [0]:<10> > [1]:<6> - ok
X: [1]:<6> > [2]:<8> - nok
M: [2]:<8> > [3]:<4> - ok
```

An 'M' at the begin of a line indicates success, an 'X' a failure.

AllPairs

The template class `AllPairs<>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `Are` creates check whether the given Rich Boolean functor applies to *all pairs of elements* in the given range. These functors take two arguments: the begin and end of a range, or the begin of a range and the number of elements to use in that range. The range should be multi-pass (see the section called “Multi-pass and single-pass ranges”). This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the comparisons in the range (even the ones for which it is ok) or only the failing comparisons; the default value is false (i.e. only show the failing comparisons).

Example 20. Using AllPairs to check the elements of a range

```
    using namespace RichBool;
    int array[] = { 10, 6, 8, 6 };
    MOD_ASSERT(rbIN_RANGE(array, array+4, AllPairs<>().Are(Different<>())));
```

The last line can be read as "assert that in the range [array, array+4) for every pair of elements, the elements are different".

Output:

```
predicate applies to all pairs in `array'-`array+4' - nok
X: [1]:<6> != [3]:<6> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

An 'M' at the begin of a line indicates success, an 'X' a failure.

Rich Boolean functor classes that take one argument

One operation that is often performed, is checking whether a certain value is present in a range, or all values are less than a certain value. This could be done by binding an argument of the Rich Boolean functor classes like `Equal` and `Less`. To make this easier, you can use the following Rich Boolean functor classes that take one argument:

- `Equals<typename T, GetValue=Value>`: checks whether its argument equals the argument given in its constructor

- `IsLessThan<typename T, GetValue=Value>`: checks whether its argument is less than the argument given in its constructor
- `IsLessOrEqualTo<typename T, GetValue=Value>`: checks whether its argument less than or equal to the argument given in its constructor
- `IsMoreThan<typename T, GetValue=Value>`: checks whether its argument is more than the argument given in its constructor
- `IsMoreOrEqualTo<typename T, GetValue=Value>`: checks whether its argument more than or equal to the argument given in its constructor
- `IsDifferentFrom<typename T, GetValue=Value>`: checks whether its argument differs from the argument given in its constructor

Their constructors take an argument of type T, which is copied. You can also use the following template functions to create these Rich Boolean Functors, that save you from specifying the template argument:

- `EqualTo(const T &t)`: returns a Rich Boolean Functor of the class `Equals<typename T, Value>` and gives `t` as an argument of the constructor.
- `LessThan(const T &t)`: returns a Rich Boolean Functor of the class `IsLessThan<typename T, Value>` and gives `t` as an argument of the constructor.
- `LessOrEqualTo(const T &t)`: returns a Rich Boolean Functor of the class `IsLessOrEqualTo<typename T, Value>` and gives `t` as an argument of the constructor.
- `MoreThan(const T &t)`: returns a Rich Boolean Functor of the class `IsMoreThan<typename T, Value>` and gives `t` as an argument of the constructor.
- `MoreOrEqualTo(const T &t)`: returns a Rich Boolean Functor of the class `IsMoreOrEqualTo<typename T, Value>` and gives `t` as an argument of the constructor.
- `IsDifferentFrom(const T &t)`: returns a Rich Boolean Functor of the class `IsDifferentFrom<typename T, Value>` and gives `t` as an argument of the constructor.

Elements of a range are then compared to the given argument. For `Equals<typename T, GetValue=Value>` this is especially useful with `Has<...>` and `Unique<...>`, for the others it is also useful with `All<...>`.

Example 21. Checking if a value is present in a range

This example checks if 5 is present in the range [begin, end) of integers.

```
MOD_ASSERT(rbIN_RANGE(begin, end, Has<>().That(EqualTo(5))));
```

Example 22. Checking if a value is present exactly once in a range

This example checks if 5 is present exactly once in the range [begin, end) of integers.

```
MOD_ASSERT(rbIN_RANGE(begin, end, Unique<>().That(EqualTo(5))));
```

Example 23. Checking if all values in a range are less than 5

This example checks if all the integers in the range [begin, end) are less than 5.

```
MOD_ASSERT(rbIN_RANGE(begin, end, All<>().Are(LessThan(5))));
```

Multi-pass and single-pass ranges

The methods of the classes in the previous section have iterators as arguments. These iterators can be multi-pass or single-pass. If they are multi-pass (a copy of an old iterator still points to the same value), they only have to conform to the concept of forward iterators. This is specified by their boolean template argument, which should be `true` if you pass multi-pass iterators, `false` if you pass single-pass iterators. The default is `true`. Sometimes the concept of multi-pass input iterators is used; these can also be used as multi-pass iterators. Using single-pass iterators with a class that expects multi-pass iterators, results in undefined behaviour. Using multi-pass iterators with a class that expects single-pass iterators works fine, but is less performant.

Comparing two ranges

To use the Rich Booleans in this section, include `richbool/richbool.hpp`.

rbIN_RANGES(begin1, end1, begin2, end2, check)

This macro gives `begin1`, `end1`, `begin2` and `end2` to the rich boolean functor `check`.

Corresponding class: none

rbIN_ARRAYS(array1, array2, check)

This macro gives the begin and end of both arrays to the rich boolean functor `check`. It is equivalent to `rbIN_RANGES(array1, array1+sizeof(array1)/sizeof(array1[0]), array2, array2+sizeof(array2)/sizeof(array2[0]), check)`. The variables `array1` and `array2` should be arrays, not pointers.

Corresponding class: none

rbIN_RANGE_ARRAY(begin, end, array, check)

This macro gives `begin`, `end` and the begin and end of `array` to the rich boolean functor `check`, in that order. It is equivalent to `rbIN_RANGES(begin, end, array, array+sizeof(array)/sizeof(array[0]), check)`. The variable `array` should be an array, not a pointer.

Corresponding class: none

rbIN_ARRAY_RANGE(array, begin, end, check)

This macro gives the begin and end of `array` and `begin` and `end` to the rich boolean functor `check`, in that order. It is equivalent to `rbIN_RANGES(array, array+sizeof(array)/sizeof(array[0]), begin, end, check)`. The variable `array` should be an array, not a pointer.

Corresponding class: none

The following 5 macros (that have VALUES in their name) are still experimental. They are only available on compilers that support C++0X initializer lists. At the moment of writing this works on gcc 4.5, but not with Visual C++.

`rbIN_VALUES_VALUES(il1, il2, check)`

This macro gives the begin and end of both initializer lists to the rich boolean functor `check`. The initializer lists should be enclosed in parens.

Corresponding class: none

`rbIN_RANGE_VALUES(begin, end, il, check)`

This macro gives `begin`, `end` and the begin and end of `il` to the rich boolean functor `check`, in that order. The initializer list should be enclosed in parens.

Corresponding class: none

`rbIN_VALUES_RANGE(il, begin, end, check)`

This macro gives the begin and end of `il` and `begin` and `end` to the rich boolean functor `check`, in that order. The initializer list should be enclosed in parens.

Corresponding class: none

`rbIN_ARRAY_VALUES(array, il, check)`

This macro gives the begin and end of `array` and the begin and end of `il` to the rich boolean functor `check`, in that order. The initializer list should be enclosed in parens.

Corresponding class: none

`rbIN_VALUES_ARRAY(il, array, check)`

This macro gives the begin and end of `il` and the begin and end of `array` to the rich boolean functor `check`, in that order. The initializer list should be enclosed in parens.

Corresponding class: none

Available classes for performing checks on two ranges

There are several Rich Boolean functor classes provided that work on two ranges, and thus can be given as the last argument to Rich Boolean macros that work on two ranges, like `rbIN_RANGES`. Some of these can be used directly, others are actually Rich Boolean functor factories, i.e. they create rich boolean functors on the fly, when you give a Rich Boolean functor to their factory method. It should be a Rich Boolean functor that takes two arguments. See the section called “Using Rich Boolean functors in Rich Boolean macros” on how to make a rich boolean functor.

AllEqual

The template class `AllEqual<bool multiPass1=true, bool multiPass2=true>` checks whether the elements in the two given ranges are equal. It takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Their default values are both true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the different ones; the default value is false (i.e. only show the different ones).

Example 24. Using AllEqual to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 4, 6, 8, 10 }, array2[] = { 4, 6, 7, 10 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4, AllEqual<>()));
```

The last line can be read as "assert that in the ranges [array1, array1+4) and [array2, array2+4) all elements are equal".

Output:

```
predicate applies to ranges `array1'-`array1+4' and `array2'-`array2+4' - nok
X: [2]:<8> == [2]:<7> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

Example 25. Using AllEqual to check the elements of a range, and also show the equal ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4, AllEqual<>(true)));
```

Output:

```
predicate applies to ranges `array1'-`array1+4' and `array2'-`array2+4' - nok
M: [0]:<4> == [0]:<4> - ok
M: [1]:<6> == [1]:<6> - ok
X: [2]:<8> == [2]:<7> - nok
M: [3]:<10> == [3]:<10> - ok
```

An 'M' at the begin of a line indicates success, an 'X' a failure.

Example 26. Using AllEqual to check the elements of a range, using the number of elements to use

```
    using namespace RichBool;
    int array1[] = { 4, 6, 8, 10 }, array2[] = { 4, 6, 7, 10 };
    MOD_ASSERT(rbIN_RANGES(array1, 4, array2, 4, AllEqual<>()));
```

The last line can be read as "assert that in the ranges [array1, array1+4) and [array2, array2+4) all elements are equal".

Output:

```
predicate applies to ranges `array1'-`4' and `array2'-`4' - nok
X: [2]:<8> == [2]:<7> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

AllEqualUnordered

The template class `AllEqualUnordered<bool multiPass1=true>` checks whether the elements in the two given ranges are equal, *in any order*. It takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether the first range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). The second range should be muli-pass. The default value is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the different ones; the default value is false (i.e. only show the different ones).

Example 27. Using AllEqualUnordered to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 4, 6, 8, 10 }, array2[] = { 6, 7, 4, 10 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        AllEqualUnordered<>()));
```

The last line can be read as "assert that in the ranges [array1, array1+4) and [array2, array2+4) all elements are equal, in any order".

Output:

```
predicate applies to ranges `array1'-`array1+4' and `array2'-`array2+4' - nok
1: [2]:<8> X  - nok
2: X [1]:<7> - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

Example 28. Using AllEqualUnordered to check the elements of a range, and also show the equal ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        AllEqualUnordered<>(true)));
```

Output:

```
predicate applies to ranges `array1'-`array1+4' and `array2'-`array2+4' - nok
M: [0]:<4> == [2]:<4> - ok
M: [1]:<6> == [0]:<6> - ok
M: [3]:<10> == [3]:<10> - ok
1: [2]:<8> X  - nok
2: X [1]:<7> - nok
```

An 'M' at the begin of a line indicates success, an 'X' a failure.

IsSubsetOf

The template class `IsSubsetOf<bool multiPass1=true, bool multiPass2=true>` checks whether the first range is a subset of the second. It takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). At least one should be multi-pass. The default value for both is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the mismatched elements; the default value is false (i.e. only show the mismatches).

Example 29. Using IsSubsetOf to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 5, 3, 1 }, array2[] = { 4, 5, 1, 7, 9 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        IsSubsetOf<>()));
```

The last line can be read as "assert that the range [array1, array1+4) is a subset of the range [array2, array2+4)".

Output:

```
`array1'-`array1+4' is subset of `array2'-`array2+4' - nok
1: [1]:<3> X  - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

IsSubsetOfSorted

The template class `IsSubsetOfSorted<bool multiPass1=true>` is identical to `IsSubsetOf<bool multiPass1=true, bool multiPass2=true>`, except that the second range should be random access and its elements should be sorted. This has the advantage that it is faster than `IsSubsetOf<bool multiPass1=true, bool multiPass2=true>`. Note that this does not check if the second range is sorted.

IsOrderedSubsetOf

The template class `IsOrderedSubsetOf<bool multiPass=true>` checks whether the first range is a subset of the second, where the elements of the first range are found in the same order in the second. It takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether one or both the ranges are single pass, or both multi-pass (see the section called “Multi-pass and single-pass ranges”). The default value is true, i.e. both are multi-pass. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the mismatched elements; the default value is false (i.e. only show the mismatches).

Example 30. Using IsOrderedSubsetOf to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 5, 3, 1 }, array2[] = { 4, 5, 1, 7, 3 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        IsOrderedSubsetOf<>()));
```

The last line can be read as "assert that the range [array1, array1+4) is an ordered subset of the range [array2, array2+4)".

Output:

```
`array1'-`array1+4' is ordered subset of `array2'-`array2+4' - nok
1: [2]:<1> X  - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >. In this output you see that the `1` could not be matched after the `3` was matched, because the algorithm doesn't walk backwards.

IsOrderedSubsetOfSorted

The template class `IsOrderedSubsetOfSorted<bool  multiPass1=true>` is identical to `IsOrderedSubsetOf<bool multiPass1=true, bool multiPass2=true>`, except that the second range should be random access and its elements should be sorted. This has the advantage that it is faster than `IsOrderedSubsetOf<bool multiPass1=true, bool multiPass2=true>`. Note that this does not check if the second range is sorted.

IsMultiSubsetOf

The template class `IsMultiSubsetOf<bool multiPass1=true, bool multiPass2=true>` checks whether the first range is a subset of the second, where an element in the first range can appear more than once if it appears at least once in the second range. It takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). At least one should be multi-pass. The default value for both is true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the mismatched elements; the default value is false (i.e. only show the mismatches).

Example 31. Using IsSubsetOf to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 5, 3, 5, 1, 1, 5 }, array2[] = { 4, 5, 1, 7, 9 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        IsMultiSubsetOf<>()));
```

The last line can be read as "assert that the range [array1, array1+4) is a multisubset of the range [array2, array2+4)".

Output:

```
`array1'-`array1+4' is multisubset of `array2'-`array2+4' - nok
1: [1]:<3> X  - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

IsMultiSubsetOfSorted

The template class `IsMultiSubsetOfSorted<bool  multiPass1=true>` is identical to `IsMultiSubsetOf<bool multiPass1=true, bool multiPass2=true>`, except that the second range should be random access and its elements should be sorted. This has the advantage that it is faster than `IsMultiSubsetOf<bool multiPass1=true, bool multiPass2=true>`. Note that this does not check if the second range is sorted.

Note that the concepts of `ordered subset' and `multisubset' don't exist in the usual mathematical theory of sets, because there an element can appear only once in a set, and sets don't have an order. Many C++ containers don't have these restrictions.

Note: it is obvious that an optimization could be made for subsets and multisubsets if the second range is a `std::set`, `std::multiset` etc. This will be done in a future release.

Compare

The template class `Compare<bool multiPass1=true, bool multiPass2=true>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `That` creates, check whether the given Rich Boolean functor applies to the *corresponding elements* in the given ranges. These functors can be used in Rich Booleans that work on two ranges or containers (because they take four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range - you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). Their default values are both true. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones for which it is ok) or only the failing ones; the default value is false (i.e. only show the failing ones).

Example 32. Using Compare to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 1, 4, 8, 9, 11 }, array2[] = { 4, 6, 7, 10 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+5, array2, array2+4,
        Compare<>().That(Less<>())));
```

The last line can be read as "assert that when the ranges [array1, array1+5) and [array2, array2+4) are compared, the elements in the first are less than the elements in the second".

Output:

```
predicate applies to ranges `array1'-`array1+5' and `array2'-`array2+4' - nok
X: [2]:<8> < [2]:<7> - nok
1: [4]:<11> X - nok
```

The number between [ ] is the index of the element in the range. The value of the element is between < >. An 'M' at the begin of a line indicates success, an 'X' a failure, a '1' an unmatched element in the first range, a '2' an unmatched element in the second range.

Example 33. Using Compare to check the elements of a range, also showing the good ones

```
    // continued from previous listing
    MOD_ASSERT(rbIN_RANGES(array1, array1+5, array2, array2+4,
        Compare<>(true).That(Less<>())));
```

Output:

```
predicate applies to ranges `array1'-`array1+4' and `array2'-`array2+4' - nok
M: [0]:<1> < [0]:<4> - ok
M: [1]:<4> < [1]:<6> - ok
X: [2]:<8> < [2]:<7> - nok
M: [3]:<9> < [3]:<10> - ok
1: [4]:<11> X - nok
```

CompareUnordered

The template class `CompareUnordered<bool multiPass1=true>` is a Rich Boolean functor factory class. The Rich Boolean functors that its method `That` creates, check whether the given rich boolean functor applies to the elements in the given ranges, *in any order*. These objects can be used in Rich Booleans that work on two ranges or containers (because they take four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range - you can mix these two types). The template argument tells whether the first range is single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). The default value is true. The second range should always be multi-pass. This class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones for which it is ok) or only the failing ones; the default value is false (i.e. only show the failing ones).

Example 34. Using CompareUnordered to check the elements of a range

```
    using namespace RichBool;
    int array1[] = { 2, 3, 1 }, array2[] = { 3, 0, 2 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+3, array2, array2+3,
        CompareUnordered<>().That(Equal<>())));
```

The last line can be read as "assert that when the ranges [array1, array1+3) and [array2, array2+3) are compared unordered, the elements in the first are equal to the elements in the second".

Output:

```
predicate applies to unordered ranges `a'-`a+3' and `b'-`b+3' - nok
1: [2]:<1> X  - nok
2: X [1]:<0> - nok
```

The number between [ ] is the index of the element in the range. The value of the element is between < >. A '1' at the begin of a line indicates an unmatched element in the first range, a '2' an unmatched element in the second range.

Example 35. Using CompareUnordered to check the elements of a range, also showing the good ones

```
    using namespace RichBool;
    int array1[] = { 2, 3, 1 }, array2[] = { 3, 0, 2 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+3, array2, array2+3,
        CompareUnordered<>(true).That(Equal<>())));
```

Output:

```
predicate applies to unordered ranges `a'-`a+3' and `b'-`b+3' - nok
M: [0]:<2> == [2]:<2> - ok
M: [1]:<3> == [0]:<3> - ok
1: [2]:<1> X  - nok
2: X [1]:<0> - nok
```

An 'M' at the begin of a line indicates success, a '1' an unmatched element in the first range, a '2' an unmatched element in the second range.

MatchesSubsetOf

The template class `MatchesSubsetOf<bool multiPass1=true, bool multiPass2=true>` is a Rich Boolean factory, that has a method `That` that has a Rich Boolean functor `rb` as an argument, and returns a Rich Boolean functor that checks whether for every element `a` in the first range there is an element `b` in the second for which `rb(a, b)` returns `true`, where every element in the second range can be matched to only one element in the first range. The Rich Boolean object that is returned by `That` can be used in Rich Booleans that work on two ranges or containers (because it takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range - you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). At least one should be multi-pass. The default value for both is true. This template class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the mismatched elements; the default value is false (i.e. only show the mismatches).

Example 36. Using MatchesSubsetOf to check the elements of a range

```
    using namespace RichBool;
    Pred2<ProductIs12> productIs12;
    int array1[] = { 4, 3, 1, 3 }, array2[] = { 12, 4, 5, 9 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        MatchesSubsetOf<>().That(productIs12)));
```

The last line can be read as "assert that the range [array1, array1+4) matches with a subset of the range [array2, array2+4) where the product is 12".

Output:

```
`array1'-`array1+4' matches subset of `array2'-`array2+4' - nok
1: [0]:<4> X  - nok
1: [3]:<3> X  - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >

MatchesMultiSubsetOf

The template class `MatchesMultiSubsetOf<bool multiPass1=true, bool multiPass2=true>` is a Rich Boolean factory, that has a method `That` that has a Rich Boolean functor `rb` as an argument, and returns a Rich Boolean that checks whether for every element `a` in the first range there is an element `b` in the second for which `rb(a, b)` returns `true`, where every element in the second range can be matched to more than one element in the first range. The Rich Boolean object that is returned by `That` takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). At least one should be multi-pass. The default value for both is true. This template class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the mismatched elements; the default value is false (i.e. only show the mismatches).

Example 37. Using MatchesMultiSubsetOf to check the elements of a range

```
    using namespace RichBool;
 Pred2<ProductIs12> productIs12;
    int array1[] = { 4, 3, 1, 3 }, array2[] = { 12, 4, 5, 9 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        MatchesMultiSubsetOf<>().That(productIs12)));
```

The last line can be read as "assert that the range [array1, array1+4) matches with a subset of the range [array2, array2+4) where the product is 12".

Output:

```
`array1'-`array1+4' matches subset of `array2'-`array2+4' - nok
1: [0]:<4> X  - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >. Note that both threes in the first range are matched to the four in the second range.

MatchesOrderedSubsetOf

The template class `MatchesOrderedSubsetOf<bool multiPass1=true, bool multiPass2=true>` is a Rich Boolean factory, that has a method `That` that has a Rich Boolean functor `rb` as an argument, and returns a Rich Boolean that checks whether for every element `a` in the first range there is an element `b` in the second for which `rb(a, b)` returns `true`, where the matched elements in the second range are in the same order as the elements in the first range. The Rich Boolean object that is returned by `That` takes four arguments for two ranges, that are each the begin and end of a range, or the begin of a range and the number of elements to use in that range (you can mix these two types). The template arguments tell whether the ranges are single pass or multi-pass (see the section called “Multi-pass and single-pass ranges”). At least one should be multi-pass. The default value for both is true. This template class has a constructor that takes a boolean, that tells whether the analysis should contain the analysis for all the elements in the range (even the ones that are equal) or only the mismatched elements; the default value is false (i.e. only show the mismatches).

Example 38. Using MatchesOrderedSubsetOf to check the elements of a range

```
    using namespace RichBool;
 Pred2<ProductIs12> productIs12;
    int array1[] = { 4, 3, 1, 6 }, array2[] = { 4, 2, 12, 9 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+4,
        MatchesOrderedSubsetOf<>().That(productIs12)));
```

The last line can be read as "assert that the range [array1, array1+4) matches with a subset of the range [array2, array2+4) where the product is 12".

Output:

```
`array1'-`array1+4' matches subset of `array2'-`array2+4' - nok
1: [0]:<4> X  - nok
1: [3]:<6> X  - nok
```

The value between [ ] is the index of the element in the range. The value of the element is between < >. Note that the 6 could not be matched with the 2, because the 1 was already matched with the 12, so matching the 6 with the 2 would violate the order.

IsSubsetOfCustomSorted

The template class `IsSubsetOfCustomSorted<bool multiPass1=true>` is a Rich Boolean factory, whose method `By` takes a predicate `pred` as an argument and returns a Rich Boolean identical to `IsSubsetOf<bool multiPass1=true>`, except that the second range should be sorted by `pred`, i.e. such that for every element `a` that is followed by an element `b`, `pred(a, b)` returns `true`. This has the advantage that it is faster than `IsSubsetOf<bool multiPass1=true, bool multiPass2=true>`. Note that this does not check if the second range is sorted.

Example 39. Using IsSubsetOfCustomSorted to check the elements of two ranges

```
    using namespace RichBool;
    int array1[] = { 4, 3, 1, 6 }, array2[] = { 9, 6, 4, 2, 1 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+5,
        IsSubsetOfCustomSorted<>().By(more)));
```

The last line can be read as "assert that the range [array1, array1+4) is a subset of the range [array2, array2+5)".

Output:

```
`array1'-`array1+4' is subset of `array2'-`array2+5' - nok
1: [2]:<1> X  - nok
```

IsMultiSubsetOfCustomSorted

The template class `IsMultiSubsetOfCustomSorted<bool multiPass1=true>` is a Rich Boolean factory, whose method `By` takes a predicate `pred` as an argument and returns a Rich Boolean identical to `IsMultiSubsetOf<bool multiPass1=true>`, except that the second range should be sorted by `pred`, i.e. such that for every element `a` that is followed by an element `b`, `pred(a, b)` returns `true`. This has the advantage that it is faster than `IsMultiSubsetOf<bool multiPass1=true, bool multiPass2=true>`. Note that this does not check if the second range is sorted.

Example 40. Using IsMultiSubsetOfCustomSorted to check the elements of two ranges

```
    using namespace RichBool;
    int array1[] = { 4, 3, 6, 1, 6 }, array2[] = { 9, 6, 4, 2, 1 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+5, array2, array2+5,
        IsMultiSubsetOfCustomSorted<>().By(more)));
```

The last line can be read as "assert that the range [array1, array1+=54) is a multi subset of the range [array2, array2+5)".

Output:

```
`array1'-`array1+5' is multisubset of `array2'-`array2+5' - nok
1: [2]:<3> X  - nok
```

Note that the six in the first range is matched twice with the same six in the second range.

IsOrderedSubsetOfCustomSorted

The template class `IsOrderedSubsetOfCustomSorted<bool multiPass1=true>` is a Rich Boolean factory, whose method `By` takes a predicate `pred` as an argument and returns a Rich Boolean identical to `IsOrderedSubsetOf<bool multiPass1=true>`, except that the second range should be sorted by `pred`, i.e. such that for every element `a` that is followed by an element `b`, `pred(a, b)` returns `true`. This has the advantage that it is faster than `IsOrderedSubsetOf<bool multiPass1=true, bool multiPass2=true>`. Note that this does not check if the second range is sorted.

Example 41. Using IsOrderedSubsetOfCustomSorted to check the elements of two ranges

```
    using namespace RichBool;
    int array1[] = { 4, 6, 3, 1 }, array2[] = { 9, 6, 4, 2, 1 };
    MOD_ASSERT(rbIN_RANGES(array1, array1+4, array2, array2+5,
        IsOrderedSubsetOfCustomSorted<>().By(more)));
```

The last line can be read as "assert that the range [array1, array1+4) is a subset of the range [array2, array2+5)".

Output:

```
`array1'-`array1+4' is subset of `array2'-`array2+5' - nok
1: [1]:<6> X  - nok
1: [2]:<3> X  - nok
```

Note that the 6 could not be matched, because the 4 was already matched in the second range, which comes after the 6 in the second range.

Multi-pass and single-pass ranges

The methods of the classes in the previous section have iterators as arguments. These iterators can be multi-pass or single-pass. If they are multi-pass (a copy of an old iterator still points to the same value), they only have to conform to the concept of forward iterators. This is specified by the two boolean template arguments of the classes, which should be `true` if you pass multi-pass iterators, `false` if you pass single-pass iterators. The first boolean applies to the first range, the second boolean applies to the second range. If one is single-pass and the other is multi-pass, the single-pass should be the first range (the specializations `<true, false>` are not defined, because the specializations `<false, true>` can be used by swapping the arguments). The default for both is `true`. Sometimes the concept of multi-pass input iterators is used; these can also be used as multi-pass iterators. Using single-pass iterators where multi-pass iterators are expected, results in undefined behaviour. Using multi-pass iterators where single-pass iterators are expected works fine, but is less performant.

Note: if both ranges are single-pass, you have to supply both template arguments, but in that case you can not define the object inside the macro, because the preprocessor doesn't understand templates, and would suppose that it is two arguments of the macro, and therefore it doesn't compile. Adding parentheses only works portably if you give an argument to its constructor (because of C++ parsing rules), so give false or true as an argument to the constructor. An alternative is to define the object before the macro.

Dynamic matching

`Compare<...>` and `AllEqual<...>` from the previous sections perform dynamic matching. This means that if a mismatch is found between two elements, it will try to search a match for each of the two in the following ten elements (if at least one of the two ranges is multi-pass). In fact it even does more than that if the two ranges are both multi-pass; in that case it checks which of the ten next elements in both ranges matches with which, and finds an optimal path.

This means that elements may be unmatched. This is shown with a '1' or '2' instead of 'M' or 'X' at the begin of a line, to indicate that an unmatched element is only in the first or second range. Suppose you have two ranges of integers with the values { 1, 2, 12, 3 } and { 1, 10, 2, 3 }, and compare them with `AllEqual<...>`. Then the output would be

```
predicate does not apply on ranges `array1'-`array1+' and `array2'-`array2+4'
M: [0]:<1> == [0]:<1> - ok
2: X   [1]:<10>
M: [1]:<2> == [2]:<2> - ok
1: [2]:<12>   X
M: [3]:<3> == [3]:<3> - ok
```

An X represents the missing element in the other range.

Containers with methods begin() and end()

To use the Rich Booleans in this section, include `richbool/containers.hpp`.

Note: The containers that are given to the Rich Booleans in this section, should have const methods `begin()` and `end()` that return iterators that together specify the range of the container. Containers in the standard template library have such methods, but every other container that has these methods can be used. Furthermore the elements in the containers should be streamable (see the section called “Choose your strings”).

Note: The containers that are given to the Rich Booleans that take two containers, can be of different types, e.g. `std::vector` and `std::list`. Even their elements can be of different types, as long as the Rich Boolean functor can have them as arguments.

rbIN_CONTAINER(container, check)

This macro checks whether the elements in the container fulfill the check, by giving `container.begin()` and `container.end()` to `check`. It is equivalent to `rbIN_RANGE(container.begin(), container.end(), check)`. See the section called “Available Rich Boolean functor classes for checking a range” for possible choices for `check`

Corresponding class: `InContainer<class Algorithm, class GetValue1=Value>` This class has the following constructors:

- a default constructor
- a constructor that takes one argument of the type `Algorithm`

If the default constructor is used, the default constructor of `Algorithm` is called. Objects of this template class can also be made with the template function `template<Algorithm> InContainer<class Algorithm> MakeInContainer(Algorithm algorithm)`.

rbIN_CONTAINERS(container1, container2, check)

This macro checks whether the elements in the containers fulfill the check, by giving `container1.begin()`, `container1.end()`, `container2.begin()` and `container2.end()` to `check`. It is equivalent to `rbIN_RANGES(container1.begin(), container1.end(), container2.begin(), container2.end(), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: `InContainers<class Algorithm, class GetValue1=Value>` This class has the following constructors:

- a default constructor
- a constructor that takes one argument of the type `Algorithm`

If the default constructor is used, the default constructor of `Algorithm` is called. Objects of this template class can also be made with the template function `template<Algorithm> InContainers<class Algorithm> MakeInContainers(Algorithm algorithm)`.

rbIN_CONTAINER_RANGE(container, begin, end, check)

This macro checks whether the elements in the container and the range `[begin, end)` fulfill the check, by giving `container.begin()`, `container.end()`, and `begin` and `end` to `check`, in that order. It is equivalent to `rbIN_RANGES(container.begin(), container.end(), begin, end, check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

rbIN_RANGE_CONTAINER(begin, end, container, check)

This macro checks whether the elements in the range `[begin, end)` and the container fulfill the check, by giving `begin`, `end`, `container.begin()` and `container.end()` to `check`, in that order. It is equivalent to `rbIN_RANGES(begin, end, container.begin(), container.end(), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

rbIN_CONTAINER_ARRAY(container, array, check)

This macro checks whether the elements in the container and the array fulfill the check, by giving `container.begin()`, `container.end()`, and the begin and end of the array to `check`, in that order. It is equivalent to `rbIN_RANGES(container.begin(), container.end(), array, array+sizeof(array)/sizeof(array[0]), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

rbIN_ARRAY_CONTAINER(array, container, check)

This macro checks whether the elements in the array and the container fulfill the check, by giving the begin and end of the array, and `container.begin()`, `container.end()` to `check`, in that order.

It is equivalent to `rbIN_RANGES(array, array+sizeof(array)/sizeof(array[0]), container.begin(), container.end(), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

The following 2 macros (that have VALUES in their name) are still experimental. They are only available on compilers that support C++0X initializer lists. At the moment of writing this works on gcc 4.5, but not with Visual C++.

rbIN_CONTAINER_VALUES(container, il, check)

This macro checks whether the elements in the container and the array fulfill the check, by giving `container.begin()`, `container.end()`, and the begin and end of the array to `check`, in that order. It is equivalent to `rbIN_RANGES(container.begin(), container.end(), array, array+sizeof(array)/sizeof(array[0]), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

rbIN_VALUES_CONTAINER(il, container, check)

This macro checks whether the elements in the array and the container fulfill the check, by giving the begin and end of the array, and `container.begin()`, `container.end()` to `check`, in that order. It is equivalent to `rbIN_RANGES(array, array+sizeof(array)/sizeof(array[0]), container.begin(), container.end(), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

Checking maps

For maps, the rich boolean functor class `EqualPair` can be used, which checks if the first and second element of two pairs are both equal. E.g. `rbIN_CONTAINERS(map1, map2, Compare<>().That(EqualPair()))`.

Example output:

```
predicate doesn't apply on ranges `map1.begin()'-`map1.end()'
and `map2.begin()'-`map2.end()'
M: keys: {(1) == (1) - ok}, values: {(a) == (a) - ok} - ok
X: keys: {(2) == (2) - ok}, values: {(b) == (q) - nok} - nok
M: keys: {(3) == (3) - ok}, values: {(c) == (c) - ok} - ok
```

Note: The matching algorithm uses partial matches for this rich boolean, which means that two different elements in the range of which either the key or the value is equal, will be matched in preference to elements that are totally different

Creating an inline container

The function `Values` takes from one to eight arguments of the same type, and returns a container that contains these values. This container can be used in any Rich Boolean where a container is expected.

```
std::vector<int> vec;
...
// check that vec contains the values 2, 4 and 5:
RB_ASSERT(rbIN_CONTAINERS(RichBool::Values(2, 4, 5), vec, RichBool::AllEqual<>()
```

Containers with overloaded funtions `begin()` and `end()`

To use the Rich Booleans in this section, include `richbool/xcontainers.hpp`.

Note: the containers that are given to the Rich Booleans that take two containers, can be of different types. Even their elements can be of different types, as long as the Rich Boolean functor can have them as arguments.

The Rich Booleans in this section work with every container for which overloaded non-member functions `begin` and `end` exist that return iterators over the container, and whose elements can be streamed out (see the section called “Choose your strings”). A method to stream out the whole container not necessary.

Defining overloaded functions that return iterators can be done easily for the `wxArray` (whether they are defined with `WX_DEFINE_ARRAY` or with `WX_DECLARE_OBJ_ARRAY` and `WX_DEFINE_OBJ_ARRAY`) and `wxList` containers in wxWidgets, with the macros `WX_DEFINE_ARRAY_ITERATOR` and `WX_DEFINE_LIST_ITERATOR`, that are defined in the file `richbool/wx_iter.hpp`.

Example 42. Making iterators for a wxArray

```
#include "richbool/xcontainers.hpp"
#include "richbool/wx_iter.hpp"

WX_DECLARE_OBJARRAY(MyClass, MyArray);
WX_DEFINE_ARRAY_ITERATOR(MyClass, MyArray, MyArrayIterator);
MyArray arr1, arr2;
...
MOD_ASSERT(rbIN_XCONTAINERS(arr1, arr2, AllEqual<>()));
```

For arrays that contain primitive elements, add a suffix _P to avoid compiler warnings about `operator->`:

```
WX_DEFINE_ARRAY_INT(int, IntArray);
WX_DEFINE_ARRAY_ITERATOR_P(int, IntArray, IntArrayIterator);
IntArray arr1, arr2;
...
MOD_ASSERT(rbIN_XCONTAINERS(arr1, arr2, AllEqual<>()));
```

Note that wxWidgets 2.6 added the methods `begin` and `end` to many of the containers (but not all), so that the equivalent Rich Booleans for STL-like containers can be used. For these containers it is easier to use these versions instead of the ones in this section, since you don't have to define iterators on them.

rbIN_XCONTAINER(container, check)

This macro checks whether the elements in the container fulfill the check, by giving `begin(container)` and `end(container)` to `check`. It is equivalent to `rbIN_RANGE(begin(container), end(container), check)`. See the section called “Available Rich Boolean functor classes for checking a range” for possible choices for `check`

Corresponding class: `InXContainer<class Algorithm, class GetValue1=Value>` This class has the following constructors:

- a default constructor
- a constructor that takes one argument of the type `Algorithm`

If the default constructor is used, the default constructor of `Algorithm` is called. Objects of this template class can also be made with the template function `template<Algorithm> InXContainer<class Algorithm> MakeInXContainer(Algorithm algorithm)`.

rbIN_XCONTAINERS(container1, container2, check)

This macro checks whether the elements in the containers fulfill the check, by giving `begin(container1)`, `end(container1)`, `begin(container2)` and `end(container2)` to `check`. It is equivalent to `rbIN_RANGES(begin(container1), end(container1), begin(container2), end(container2), check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: `InXContainers<class Algorithm, class GetValue1=Value, class GetValue2=Value>` This class has the following constructors:

- a default constructor
- a constructor that takes one argument of the type `Algorithm`

If the default constructor is used, the default constructor of `Algorithm` is called. Objects of this template class can also be made with the template function `template<Algorithm> InXContainers<class Algorithm> MakeInXContainers(Algorithm algorithm)`.

rbIN_XCONTAINER_RANGE(container, begin, end, check)

This macro checks whether the elements in the container and the range `[begin, end)` fulfill the check, by giving `begin(container)`, `end(container)`, and `begin` and `end` to `check`, in that order. It is equivalent to `rbIN_RANGES(begin(container), end(container), begin, end, check)`. See the section called “Available classes for performing checks on two ranges” for possible choices for `check`

Corresponding class: none

rbIN_RANGE_XCONTAINER(begin, end, container, check)

This macro checks whether the elements in the range `[begin, end)` and the container fulfill the check, by giving `begin`, `end`, `begin(container)` and `end(container)` to `check`, in that order.

It is equivalent to `rbIN_RANGES(begin, end, begin(container), end(container), check)`. See the section called "Available classes for performing checks on two ranges" for possible choices for `check`

Corresponding class: none

rbIN_XCONTAINER_ARRAY(container, array, check)

This macro checks whether the elements in the container and the array fulfill the check, by giving `begin(container)`, `end(container)`, and the begin and end of the array to `check`. It is equivalent to `rbIN_RANGES(begin(container), end(container), array, array+sizeof(array)/sizeof(array[0]), check)`. See the section called "Available classes for performing checks on two ranges" for possible choices for `check`

Corresponding class: none

rbIN_ARRAY_XCONTAINER(array, container, check)

This macro checks whether the elements in the array and the container fulfill the check, by giving the begin and end of the array, and `begin(container)`, `end(container)` to `check`, in that order. It is equivalent to `rbIN_RANGES(array, array+sizeof(array)/sizeof(array[0]), begin(container), end(container), check)`. See the section called "Available classes for performing checks on two ranges" for possible choices for `check`

Corresponding class: none

Deprecated Rich Booleans

Working on one range

`rbIN_RANGE_ARG(begin, end, check, arg)`, `rbIN_CONTAINER_ARG(container, check, arg)` and `rbIN_XCONTAINER_ARG(container, check, arg)`.

These are similar to their counterparts without the suffix `_ARG`, but `check` is a Rich Boolean functor factory, and `arg` a Rich Boolean functor (that takes one argument), that is given to `check` to create a Rich Boolean functor that performs the check on the range.

Corresponding class: none

Working on two ranges

`rbIN_RANGES_ARG(begin1, end1, begin2, end2, check, arg)`, `rbIN_RANGE_ARRAY_ARG(begin, end, array, check, arg)`, `rbIN_CONTAINERS_ARG(container1, container2, check, arg)`, `rbIN_CONTAINER_ARRAY_ARG(container, array, check, arg)`, `rbIN_XCONTAINERS_ARG(container1, container2, check, arg)` and `rbIN_XCONTAINER_ARRAY_ARG(container, array, check, arg)`

.

These are similar to their counterparts without the suffix `_ARG`, but `check` is a Rich Boolean functor factory, and `arg` a Rich Boolean functor (that takes two arguments), that is given to `check` to create a Rich Boolean functor that performs the check on the two ranges.

Using Rich Boolean functors in Rich Boolean macros

The Rich Booleans that end in _RB, like `rb2_RB`, and the ones that work on one or two ranges, have a Rich Boolean functor as their last argument, so you have to create such an object. For the ones that work on ranges, you can use a Rich Boolean functor factory like `All`, `Has`, `Unique` or `Compare`, but you still need to supply another Rich Boolean functor to their factory method. Many Rich Boolean macros have a corresponding class, of which you should construct an object to provide as that argument. See the previous sections to see what the corresponding functor class of a Rich Boolean is, if it has one.

This allows you to do powerful comparisons of two ranges or containers. E.g. the following checks whether the integers in one vector are smaller than the integers in another vector:

```
    rbIN_CONTAINERS(vec1, vec2, Compare<>().That(Less<>()))
```

The following checks whether two vectors of vectors are equal:

```
    rbIN_CONTAINERS(vec1, vec2,
        Compare<>().That(MakeInContainers(Compare<>().That(Equal<>()))))
```

The following checks whether the integers in one vector of vectors are smaller than the integers in another vector of vectors:

```
    rbIN_CONTAINERS(vec1, vec2,
        Compare<>().That(MakeInContainers(Compare<>().That(Less<>())))) )
```

Getters

The classes that correspond to Rich Booleans can have optionally template parameters that customize how the value that it uses, is to be retrieved. The default is `RichBool::Value`, which simply returns the value itself.

Dereferencing

`RichBool::Pointer<class  GetValue=RichBool::Value>` can be used to dereference pointers. E.g. the following compares a vector of pointers to integers, with a vector of integers:

```
    rbIN_CONTAINERS(vec1, vec2,
        Compare<>().That(Equal<Pointer<>, Value >()))
```

For iterators, smart pointers and other pointerlike objects, use the class `RichBool::PointerLike<typename T, class GetValue=RichBool::Value>`, where T is the type of the objects that are returned when it is dereferenced. E.g. the following compares a vector of iterators to integers with a vector of integers:

```
    rbIN_CONTAINERS(vec1, vec2,
        Compare<>().That(Equal<PointerLike<int>, Value >()))
```

These can be nested, e.g. the following compares a vector of pointers to pointers to integers with a vector of integers:

```
    rbIN_CONTAINERS(vec1, vec2,
        Compare<>().That(Equal<Pointer<Pointer> >, Value >()))
```

Note: the second template argument `Value` isn't necessary in these examples because that is the default type, but it was shown for clarity.

Note: if your compiler doesn't support partial template specialization, you can't nest `Pointer`, but you can use `PointerLike` instead.

When you nest, you may run into troubles, because `Pointer` and `PointerLike` return a const reference, which may refer to a temporary that no longer exists. In these cases, you should use the alternatives `PointerToValue` and `PointerLikeToValue`, that return a value. Otherwise they are exactly the same as the former.

`PointerLike` can also be nested. In this case, the template type that tells which value is returned, should always be the same. The following demonstrates this:

```
    rbIN_CONTAINERS(vec1, vec2,
        Compare<>().That(Equal<PointerLike<int, PointerLike<int> >, Value>()))
```

Using a member of an object

If you want the Rich Boolean to work on a member of a given object instead of the object itself, use the template class `GetMember<T, Return>` instead of `Value`, where `T` is the type of the object, and `Return` is the type of the member. The constructor of this template class takes a pointer to a member and a `const char*` value that describes the member (starting with a period for readability). The following illustrates this:

```
class MyClass
{
public:
 int a;
 ...
};

std::vector<MyClass> vec1, vec2;
std::vector<int> int_vec;

...

RichBool::GetMember<MyClass, int> gm(&MyClass::a, ".a");

RichBool::Equal<RichBool::GetMember<MyClass, int> > equal_useMemberOfFirst(gm);
MOD_ASSERT(rbIN_CONTAINERS(vec1, int_vec,
    Compare<>().That(equal_useMemberOfFirst)));

RichBool::Equal<
    RichBool::GetMember< MyClass, int>,
    RichBool::GetMember< MyClass, int>
    >
    equal_useMemberOfBoth(gm, gm);
MOD_ASSERT(rbIN_CONTAINERS(vec1, vec2,
    Compare<>().That(equal_useMemberOfBoth)));
```

Using a method of an object

You can do the same with the return value of a method of a given object instead of the object itself, using the template class `CallMethod<T, Return>` instead of `Value`, where `T` is the type of the object, and `Return` is the return type of the member. The constructor of this template class takes a pointer to a method and a `const char*` value that describes the method (ideally preceded by a periond). The method should be `const`. The following illustrates this:

```
class MyClass
{
public:
 int GetA() const;
 ...
};

std::vector<MyClass> vec1;
std::vector<int> int_vec;

...

RichBool::CallMethod<MyClass, int> gm(&MyClass::GetA, ".GetA()");
RichBool::Equal<RichBool::CallMethod<MyClass, int> > equal_useMethod(gm);

MOD_ASSERT(rbIN_CONTAINERS(vec1, int_vec,
    Compare<>().That(equal_useMethod)));
```

GetMemberArg and CallMethodArg for older compilers

Note: some older compilers, e.g. Visual C++ 6.0, have problems with the way templates are used in the classes `GetMember` and `CallMethod`. If this is the case for your compiler, adjust the file `config.hpp` in the directory `include/richbool` so that the symbol

`RICHBOOL_NO_TEMPLATE_TYPE_LOOKUP` is defined. This will instruct the preprocessor to remove these classes. Instead you can still use the classes `GetMemberArg` and `CallMethodArg`, which have a third mandatory template argument, which is the type that is returned. If you don't nest another class in it but the default, this is the same as the second template argument.

Composing getters

You can also compose objects of the classes `Pointer`, `PointerToValue`, `PointerLike`, `PointerLikeToValue`, `GetMember`, `CallMethod`, `GetMemberArg` and `CallMethodArg` by making the template argument a class of another, and pass an object of that class to the constructor if it is not an empty object (e.g. objects of the classes `GetMember` and `CallMethod`). Then the nesting order is important. First the outer object works on its argument, then the one inside it. E.g. if you have pointers to list-iterators that hold integers, use `Pointer<PointerLike<int> >`, so that the pointer is dereferenced by `Pointer`, after which the iterator is dereferenced by . If you have list-iterators that hold pointers to integers, use `PointerLike<int, Pointer<> >`, to do the same operations in the opposite order.

If you have a class with a pointer `pa` to an integer in it, use `GetMember<MyClass, int, RichBool::Pointer<> > gmp(&MyClass::pa, ".pa")`. If you do this with `GetMemberArg`, use `GetMemberArg<MyClass, int, int*, RichBool::Pointer<> > gmp(&MyClass::pa, ".pa")`, because the third template argument is the type that the enclosed.

Multiple levels of dereferencing

Rich Boolean functors can also be used in the macros `rb1_RB`, `rb2_RB`, `rb3_RB` and so on. This is only useful if you have e.g. a pointer to a pointer, and want to check if every level of dereferencing is safe. Note that one level of dereferencing is always safe with the Rich Booleans. For two levels, you need `Pointer` as a template parameter. Suppose `p` is a pointer to a pointer to an integer:

```
rb2_RB(*p, 5, Equal<Pointer<> >())
```

Now if either `p` or `*p` is an invalid pointer, it will be safely handled.

Defining custom data retrievers (advanced)

You can also define your own classes to retrieve data, and use these as template parameters in Rich Boolean functors, like you use `Pointer` or `GetMember`. With the available classes the section called “Using Rich Boolean functors in Rich Boolean macros”, this is usually not necessary. If you want to do it, you need to create a class that has these 5 methods:

- `ReturnValue operator()(const T &t)` to get the desired value from the object
- `bool BadPtrChain(const T &t)` to check if the address of the object is valid
- `template<class Stringize_> std::string ToString(const TwoInts &t, const Stringize_ &str) const` to convert the value to a string
- `template<typename T> std::string AddressToString(const T *t) const` to convert an address to a string (usually the same as in the example below). Its argument will be a pointer to the object that is given to `operator()`.

- `SharedExpression Convert(SharedExpression expr) const` to convert an expression to another expression that adds information to it. The return type should be derived from `Expression`. `PrefixedExpression` and `SuffixedExpression` are useful classes for this, see the section called "Expression objects"

Note: replace `std::string` by `wxString` above if you use wxWidgets.

If you want to nest this in `Pointer`, you also need to add a nested template class `Return` with one template parameter. The template parameter specifies the type that is given to `operator()`. In that template class there should be a typedef `Type` that is the return type of `operator()` when an object of the template type is given to it. Usually the template parameter doesn't influence the return type (unlike in e.g. `Pointer`).

Example 43. Writing a class to let Rich Boolean functors retrieve a value in a custom way

Suppose you have a struct that has two ints:

```
struct TwoInts
{
    int a, b;
};
```

and you want to check against one of them, then you can create a struct as follows:

```
struct GetA
{
    // the return type of operator() is independent of the type of its argument
    template<typename T>
    struct Return { typedef int Type; };

    const int& operator()(const TwoInts &t) const { return t.a; }

    bool BadPtrChain(const TwoInts &t) const
    {
        return RichBool::BadPtr(&t);
    }

    template<class Stringize_>
    std::string ToString(const TwoInts &t, const Stringize_ &str) const
    {
        if (RichBool::BadPtr(&t))
            return RichBool::PtrToString(&t);
        else
            return str(t.a)+" (b="+str(t.b)+")";
    }

    template<typename T>
    std::string AddressToString(const T *t) const
    {
        return PtrToString(t);
    }

    RichBool::SharedExpression Convert(RichBool::SharedExpression expr) const
    {
        return new RichBool::SuffixedExpression(".a", expr);
    }
};
```

You can then use GetA as the template argument of a Rich Boolean functor as follows:

```
    std::vector<int> vecInt;
    std::vector<TwoInts> vecTwoInts;

    ...

    RichBool::Equal<RichBool::Value, GetA> rb;

    MOD_ASSERT(rbIN_CONTAINERS(vecInt, vecTwoInts, Compare<>().That(rb)));
```

Note: if the member a had been a pointer to an integer, the method BadPtrChain could also check if it is a valid pointer:

```
    bool BadPtrChain(const TwoInts &t) const
    {
        return RichBool::BadPtr(&t) || RichBool::BadPtr(t.a);
    }
```

Then the method ToString should also handle the case where the pointer could have a non-dereferenceable value:

```
    template<class Stringize_>
    String ToString(const TwoInts &t, const Stringize_ &str) const
    {
        if (RichBool::BadPtr(&t))
            return RichBool::PtrToString(&t);
        else if (RichBool::BadPtr(t.a))
            return RichBool::PtrToString(&t)+" -> a="+RichBool::PtrToString(t.a);
        else
            return str(*t.a)+" (b="+str(t.b)+")";
    }
```

Combining Rich Boolean functors in logical operations

Rich Boolean functors can be combined in logical operations by using `operator&`, `operator&&`, `operator|`, `operator||` and `operator^`. If `a` and `b` are Rich Boolean functors that take the same number of arguments, then

- `a & b` is a new Rich Boolean functor that succeeds if and only if they both succeed. It doesn't use short circuiting when it creates the analysis, so the conditions of both are then evaluated to give both analyses, and neither when points for partial matching are requested, so the points of both are given.
- `a && b` is a new Rich Boolean functor that succeeds if and only if they both succeed. It always uses short circuiting, so the condition of the second is only evaluated if the first succeeds. This means that the analysis of the second condition may not be available, and partial matching may not work optimal.
- `a | b` is a new Rich Boolean functor that succeeds if and only if at least one of them succeeds. It doesn't use short circuiting when it creates the analysis, so the conditions of both are then evaluated to give both analyses, and neither when points for partial matching are requested, so the points of both are given.
- `a || b` is a new Rich Boolean functor that succeeds if and only if at least one of them succeeds. It uses short circuiting, so the condition of the second is only evaluated if the first doesn't succeed. This means that the analysis of the second condition may not be available, and partial matching may not work optimal.
- `a ^ b` is a new Rich Boolean functor that succeeds if and only if one of them succeeds and the other fails. It never uses short circuiting, since this is not possible with this operator.
- `!a` is a new Rich Boolean functor that succeeds if and only if `a` doesn't succeed.

You should use the versions that don't use short circuiting (i.e. &, | and ^), unless you really have to use the ones that use short circuiting, because the versions that don't use short circuiting always give the analysis of both Rich Boolean functors if it fails.

Note: you should not use input iterators on combinations of two or more Rich Boolean functors, because the range may be traversed twice. However, negating a Rich Boolean functor is safe with input iterators.

Examples: the following checks whether for the objects of the class `MyClass` in `vec1` and `vec2` the return value of either `GetA()` or `GetB()` is the same:

```
std::vector<MyClass> vec1, vec2;
...
typedef CallMethod<MyClass, int> GetInt;
GetInt getA(&MyClass::GetA, ".GetA()"), getB(&MyClass::GetB, ".GetB()");
Equal<GetInt, GetInt> equalA(getA, getA), equalB(getB, getB);
MOD_ASSERT(rbIN_CONTAINERS(vec1, vec2, Compare<>().That(equalA | equalB)));
```

The following checks whether `vec1` is an ordered subset of `vec2` or have the same elements in any order:

```
MOD_ASSERT(rbIN_CONTAINERS(vec1, vec2,
      IsOrderedSubSetOf<>() | AllEqualUnordered<>()));
```

The following checks whether `vec1` is not a subset of `vec2`:

```
MOD_ASSERT(rbIN_CONTAINERS(vec1, vec2, !IsSubSetOf<>()));
```

Expression objects

The class `Expression` (in the RichBool namespace) is an abstract base class for expressions. Objects of these classes contain the description of the arguments that are given to a Rich Boolean macro. This can be e.g. text or an index.

`Expression` has these methods:

- `virtual Expression::Type GetType() const`: returns an enum to indentify the type of expression
- `bool operator==(const Expression &expr) const`: returns whether the other expression is the same
- `virtual operator std::string() const`: returns a conversion of the expression to a string (if you use wxWidgets, a `wxString` is returned)
- `virtual Expression* Clone() const`: returns a clone of the expression

The following concrete classes are derived from `Expression` in the RichBool namespace:

- `IndexExpression`: used to represent the index in a container; used in Rich Booleans that work on ranges and containers
- `TextExpression`: used to hold the expression as a `const char *` that is given in the constructor; that text is not copied, so the pointer has to exist as long as the expression object exists
- `StringExpression`: used to hold the expression as a `std::string` object (or a `wxString` object if you use wxWidgets) that is given in the constructor; that string is copied
- `PrefixedExpression`: used to add a prefix to an expression; its constructor takes a `const char *` pointer with the prefix (which is not copied) and a `SharedExpression` object; its method to convert to a string will return the prefix followed by the conversion of the contained expression to a string
- `SuffixedExpression`: used to add a suffix to an expression; its constructor takes a `const char *` pointer with the suffix (which is not copied) and a `SharedExpression` object; its method to convert to a string will return the conversion of the contained expression to a string followed by the suffix

Binding parameters on a Rich Boolean functor

Rich Boolean functors that take two or three arguments can bind one or more of their arguments, creating a new Rich Boolean functor that takes one argument less. The template classes `template <class RichBool_, typename T> class BindArg<M>Of<N>`, where N is the number of arguments that the original Rich Boolean functor takes (can be 2 or 3), and M is the index of the argument to be binded (between 1 and N), are Rich Boolean functor classes that do this. T is the type of the argument you want to bind. These classes have two constructors (besides the copy constructor). One takes an argument of type T, and is the parameter to be binded. The other one has an extra argument, that is a Rich Boolean functor; this is necessary for Rich Boolean functors that are not empty objects (e.g. binding Rich Boolean functors themselves).

Binding a parameter is necessary if e.g. you want to check that every number in an array is less than 10.

```
RichBool::BindArg2Of2<RichBool::Less<>, int> lessThan10(10);
```

```
MOD_ASSERT(rbIN_RANGE(b, e, All<>().Are(lessThan10)));
```

Another interesting example is checking whether the numbers in two containers differ by less than a given number.

```
RichBool::BindArg3Of3<RichBool::Near<>, double> areClose(0.01);
MOD_ASSERT(rbIN_CONTAINERS(vec1, vec2, Compare<>().That(areClose)));
```

Bindings can also be nested, e.g. if you want to check that all the numbers in a range are close to a given number:

```
typedef RichBool::BindArg3Of3<RichBool::Near<>, double> GeneralNear;
GeneralNear generalNear(0.01);
RichBool::BindArg2Of2<GeneralNear, double> near5(generalNear, 5.0);

MOD_ASSERT(rbIN_CONTAINER(vec, All<>().Are(near5)));
```

If you want to dereference pointers or retrieve data in another way from an argument, specify this in the encapsulated rich boolean functor. The binding classes cannot take care of this.

```
typedef RichBool::Less<RichBool::Value, RichBool::Pointer<> > LessValPtr;
RichBool::BindArg1Of2<LessValPtr, int> rb(5);
MOD_ASSERT(rbIN_CONTAINER(vec, All<>().Are(rb)));
```

The constructor can optionally have text in a const char*, an integer or an `Expression` as the last argument. If one of these is given, it is converted to an expression, and will be used in the generated analysis.

```
RichBool::BindArg2Of2<RichBool::Less<>, int> lessThanN(N, "N");
MOD_ASSERT(rbIN_RANGE(b, e, All<>().Are(lessThanN)));
```

Selecting arguments

When combining Rich Boolean functors, it may happen that one of the functors has to do a check on only one of the arguments. You can do this with the template function `Get1Arg<N>(RB)`, where `N` is the index of the argument that you want (starting counting at 1), and the argument is the Rich Boolean functor that you want to use on the single argument.

The following checks whether for corresponding elements in `vec1` and `vec2`, the elements are the same or the first is `0`:

```
std::vector<int> vec1, vec2;
...
MOD_ASSERT(rbIN_CONTAINERS(vec1, vec2, Compare<>().
  That(Equal<>()|Get1Arg<1>(EqualTo(0)))));
```

There are also similar functions `GetNArgs<...>(RB)`, where N can be from 2 to 5, but these are only useful in UquoniTest.

This doesn't work with Microsoft Visual Studio 6, but you can use an alternative syntax: `Get1Arg_<N>()(RB)`.

Removing the text from analyses

If you define the symbol `RICHBOOL_NO_TEXT` before including `"richbool/richbool.hpp"`, the analyses will not contain the expressions of the top level. So instead of

```
`a':<1> == `b':<2> - nok
```

you would see

```
<1> == <2> - nok
```

Note that nested analyses usually still contain the expression, because they are index expressions, not text expressions. So you would still see

```
predicate does not apply on ranges
M: [0]:<4> == [2]:<4> - ok
M: [1]:<6> == [0]:<6> - ok
M: [3]:<10> == [3]:<10> - ok
1: [2]:<8> X  - nok
2: X [1]:<7> - nok
```

where the text expressions that contain the arrays are not visible, but the index expressions are still visible.

The advantage of defining the symbol `RICHBOOL_NO_TEXT` is that the size of your executable is reduced. For executables that are distributed to customers, the loss of the text is not so bad, because the expressions can still be looked up in the source code, and it shouldn't happen so often. For applications that you're still working on, it is better to not define the symbol, because the text helps you to identify the problem easier, and looking it up in the source code is not so practical then.

Making your own rich booleans (advanced)

To make your own Rich Booleans, you need to make a Rich Boolean functor class, and if you want to use it standalone define a macro that uses that class. Defining the macro is best done by simply defining it as `rbn_RB`, where n is the number of arguments (between 1 and 4) that your Rich Boolean takes, that takes the arguments of your Rich Boolean macro followed by an instance of your class. You can do the same with `rbvn_p_RB` for Rich Booleans that should be used in assertions that return a value, where `n` is the number of arguments (between 1 and 4), and `p` is the index (starting at 1) of the argument that has to be retured, so it is between 1 and `n`. Here is an example for a Rich Boolean that takes two arguments:

```
class MyRichBool
{
 ...
};

#define rbMY_RICHBOOL(a,b) rb2_RB(a,b, MyRichBool())
#define rbvMY_RICHBOOL(a,b) rbv2_1_RB(a,b, MyRichBool())
```

Writing Rich Boolean functor classes using the wrappers

A Rich Boolean functor is somewhat like a functor in the usual sense, but it is more than that because it can also create an analysis of the condition that it tests, for which it has an extra method. The easiest way to make a Rich Boolean functor class is by using the classes `RichBool::WrappernArg`, where n can range from 1 to 4 and indicates the number of arguments that it takes. This is a class template, of which the first template argument should be a class that does the essential work, the Rich Boolean Functor Core. The actual Rich Boolean functor class should derive from `RichBool::WrappernArg<MyFunctorCore, ...>` where `MyFunctorCore` is the functor core class. `MyFunctorCore` should be followed by n `GetValue` template class parameters and one `Stringize` template class parameter, that are ideally template class parameters of your Rich Boolean functor. `RichBool::WrappernArg<MyCheck, ...>` has two constructors, one that takes n template GetValue arguments, and a second that also has these but preceded by an object of your functor core class (otherwise its default constructor is called).

If your rich boolean functor core is not a template class, and has only a default constructor, you can define your rich boolean functor with the macro `RICHBOOL_WRAPPERnARG(functor, core)`, where n is the number of arguments (from 1 to 4), `core` is the class of your Rich Boolean Functor Core and `functor` the name of your Rich Boolean Functor.

This functor core class can be made in three ways.

RichBool::FunctorCore

The easiest is to derive your Rich Boolean Functor Core from `RichBool::FunctorCore`, and to add the method `bool operator(...) const`, that takes the arguments of the Rich Boolean and returns whether the condition succeeds. You can also add `const char* GetTextBeforep() const`, where p is between 1 and the number of arguments, and is placed before the p-th argument, and `const char* GetTextAfterp() const` that is placed after the p-th argument. You don't have to add all of these, but it is recommended to add several because it makes the analysis more readable.

Optionally you can add values in the analysis that are not an argument of the Rich Boolean, e.g. a value that is stored in the functor core object to compare it with arguments. To do so, add the method `std::string GetExtraValuep() const`, where p is again between 1 and the number of arguments. This means you can add more than one extra value. This will appear in the analysis after the argument with the same index. You can also add `SharedExpression GetExtraExpressionp() const` that returns the expression of the extra value, and will be shown before the p-th value. You can also add `const char* GetTextBeforeExtrap() const`, where p is between 1 and the number of arguments, and is placed before the p-th extra value and expression, and `const char* GetTextAfterExtrap() const` that is placed after the p-th extra value. You don't have to add all of these, but it is recommended to add several because it makes the analysis more readable.

Optionally you can add values in the analysis that are calculated from the arguments, e.g. a difference of two arguments. To do so, add the method `std::string GetResultp(...) const`, that has the same arguments as `bool operator()(...) const` , and where p is again between 1 and the number of arguments. This means you can add more than one calculated value. This will appear in the analysis after the argument with the same index. You can also add `const char* GetTextBeforeResultp() const`, where n is between 1 and the number of arguments, and is placed before the p-th result, and `const char* GetTextAfterResultp() const` that is placed after the p-th result. You don't have to add all of these, but it is recommended to add several because it makes the analysis more readable.

Here is an example on how to do it for a Rich Boolean that checks whether two integers are equal modulo 5:

```
class EqualModulo5Core: public RichBool::FunctorCore
{
public:
 bool operator()(int a, int b) const
 {
  return a%5==b%5;
 }

 const char* GetTextAfter1() const
 {
  return "%5";
 }

 const char* GetTextBefore2() const
 {
  return " == ";
 }
 const char* GetTextAfter2() const
 {
  return "%5";
 }

 std::string GetResult1(int a, int ) const
 {
  return RichBool::ToString(a%5);
 }
 const char* GetTextBeforeResult1() const
 {
  return ":";
 }

 std::string GetResult2(int , int b) const
 {
  return RichBool::ToString(b%5);
 }
 const char* GetTextBeforeResult2() const
 {
  return ":";
 }
};

template <class GV1=RichBool::Value, class GV2=RichBool::Value,
  class Stringize=RichBool::MakeString>
struct EqualModulo5: public
  RichBool::Wrapper2Arg<EqualModulo5Core, GV1, GV2, Stringize>
{
 EqualModulo5(GV1 gv1=GV1(), GV2 gv2=GV2()):
  RichBool::Wrapper2Arg<EqualModulo5Core, GV1, GV2, Stringize>(gv1, gv2)
 {}
};

#define rbEQUAL_MODULO_5(a,b) rb2_RB(a,b, EqualModulo5())
#define rbvEQUAL_MODULO_5(a,b) rbv2_1_RB(a,b, EqualModulo5())
```

The drawback is that the generated analyses can contain only text, the values of the arguments, extra values and calculated values (so you could not generate an analysis like that of e.g. `rbEQUAL_BITWISE`), but that is sufficient in most cases.

Writing Rich Boolean functor classes with a custom analysis

If you need a more elaborate analysis, (e.g. you want to add a bit representation), you should use the alternative method of creating the class that checks the condition, namely by deriving it from `RichBool::CustomFunctorCore<false>` instead of `RichBool::FunctorCore`, and adding the methods `bool operator()(...)` (similar to the one as with `RichBool::FunctorCore`) and `Analysis* Analyse`, that has the arguments of the Rich Boolean, each immediately followed by a const reference to a `std::string` object (that will contain the value of the argument as a string) and a `SharedExpression` object (that describes the argument), and a boolean after all these that indicates whether the condition succeeded (sometimes an analysis is wanted even if the condition succeeded). This method is only called if an analysis is needed, so it should never return NULL.

In addition you can add the same methods as with `RichBool::FunctorCore`, except `std::string GetResultn(...) const` and the methods that give the text before and after a result (they wouldn't be used). These are called if one or more of the arguments contains a non-dereferenceable value, so your `Analyse` method cannot be called and an analysis similar to the one as with `RichBool::FunctorCore` is created instead.

```
class EqualModulo5Core: public RichBool::CustomFunctorCore<false>
{
public:
  bool operator()(int a, int b) const
  {
    return a%5==b%5;
  }

  RichBool::Analysis* Analyse(
    int a, const std::string &str1, SharedExpression expr1,
    int b, const std::string &str2, SharedExpression expr2,
    bool ok) const
  {
    ...
  }

  const char* GetTextAfter1() const
  {
    return "%5";
  }

  const char* GetTextBefore2() const
  {
    return " == ";
  }
  const char* GetTextAfter2() const
  {
    return "%5";
  }
};

template <class GV1=RichBool::Value, class GV2=RichBool::Value,
  class Stringize=RichBool::MakeString>
struct EqualModulo5: public
  RichBool::Wrapper2Arg<EqualModulo5Check, GV1, GV2, Stringize>
```

```
{
  EqualModulo5(GV1 gv1=GV1(), GV2 gv2=GV2()):
    RichBool::Wrapper2Arg<EqualModulo5Check, GV1, GV2, Stringize>(gv1, gv2)
  {}
};

#define rbEQUAL_MODULO_5(a,b) rb2_RB(a,b, EqualModulo5())
#define rbvEQUAL_MODULO_5(a,b) rbv2_RB(a,b, EqualModulo5())
```

Objects of the class `Expression` represent the expression of an argument of a rich boolean. They have the method `operator std::string() const`, so you can assign them to a `std::string` object or give them as the argument of a function where a `std::string` is needed. `Expression` objects therefore allow to delay creation of a string until it is needed, to improve performance. They can be constructed with a string object or a `const char*` (when it describes an element that is used directly in a rich boolean) or a `size_t` (when it describes an element in a range). They can also be converted to `bool` because they have the method `operator bool() const`; if they were created with the default constructor, they return `false`, otherwise they return `true`. The class `SharedExpression` is a shared pointer to `Expression` objects, so you can dereference them to get the expression.

Writing Rich Boolean functor classes with a custom analysis, where Analyse does the checking

In some rare cases, you need to check the condition and create the analysis at the same time, or checking the condition first can only be done in a non-performant way, e.g. if you do a check on input iterators. This is less efficient, because if it turns out that the condition succeeded, you have to destroy the analysis that you create.

If you need this, you can do this by deriving your functor core from `RichBool::CustomFunctorCore<true>` instead of `RichBool::CustomFunctorCore<false>`, and adding the methods `bool operator()(...)` (similar to the one as with `RichBool::FunctorCore`) and `Analysis* Analyse`, that has now the arguments of the Rich Boolean, each immediately followed by an expression and a stringizer object, and a boolean after all these that indicates whether an analysis is wanted when the condition succeeds. The expression and the stringizer object should be template types, because they can be different types. In the present version of Rich Booleans, the expression arguments can be `size_t` or `const char*`, and the stringizer objects can only be empty objects, so it is best to pass them both by value.

If the condition succeeds, the method should return NULL, except if the boolean in the arguments is true. If the condition fails, it should not return NULL. If you have to make an analysis, you can make `Expression` objects from the expression argument, by passing it to the function `RichBool::MakeExpression`, and get a string representation of each argument by giving it to `operator()` of the stringizer object. This allows you to delay the creation of the expression and string until you need them.

`bool operator()(...)` and `Analysis* Analyse` will never be both called when you use a functor object.

Note: `bool operator()(...)` is still necessary, because sometimes the calling code doesn't want an analysis but still wants to know if the condition succeeds (e.g. in ModAssert in a `MOD_CHECK` macro when reporting for checks is disabled).

In addition you can add the same methods as with `RichBool::FunctorCore`, except `std::string GetResultn(...) const` and the methods that give the text before and after a result (they wouldn't be used). These are called if one or more of the arguments contains a non-dereferenceable value, so your `Analyse` method cannot be called and an analysis similar to the one as with `RichBool::FunctorCore` is created instead.

```
class EqualModulo5Core: public RichBool::CustomFunctorCore<true>
{
public:
  bool operator()(int a, int b) const
  {
    return a%5==b%5;
  }

  template <typename Expr1, typename Expr2,
      class Stringize1, class Stringize2>
  RichBool::Analysis* Analyse(int a, Expr1 expr1, Stringize1 str1,
    int b, Expr2 expr2, Stringize2 str2,
    bool analysisIfSuccess) const
  {
    ...
    GeneralAnalysis *ga = new GeneralAnalysis(false);

    ga.AddExpression(RichBool::MakeExpression(expr1), ": ");
    ga.AddValue(str1(a), true);
    // true, because it's a valid value if we get here
    ...
    ga.AddExpression(RichBool::MakeExpression(expr2), ": ");
    ga.AddValue(str2(b), true);
    // true, because it's a valid value if we get here
    ...
  }

  const char* GetTextAfter1() const
  {
    return "%5";
  }

  const char* GetTextBefore2() const
  {
    return " == ";
  }
  const char* GetTextAfter2() const
  {
    return "%5";
  }
};

template <class GV1=RichBool::Value, class GV2=RichBool::Value,
  class Stringize=RichBool::MakeString>
struct EqualModulo5: public
  RichBool::Wrapper2Arg<EqualModulo5Check, GV1, GV2, Stringize>
{
  EqualModulo5(GV1 gv1=GV1(), GV2 gv2=GV2()):
    RichBool::Wrapper2Arg<EqualModulo5Check, GV1, GV2, Stringize>(gv1, gv2)
  {}
};
```

```
#define rbEQUAL_MODULO_5(a,b) rb2_RB(a,b, EqualModulo5())
#define rbvEQUAL_MODULO_5(a,b) rbv2_RB(a,b, EqualModulo5())
```

Adding partial matching

When comparing two ranges with your Rich Boolean, it might be interesting to indicate whether there is a partial match. When several elements don't match, a partial match will be shown in preference. To do this, add a typedef of a number type (preferably an integer) to `Points` in your rich boolean functor core class. Furthermore you should add the methods `Points GetPoints(const T &t1, const T &t2) const`, that returns a number that is higher as the match is better, and `Points GetGood()` that returns the maximum value that `GetPoints` can return. Here is an example of a Rich Boolean that compares two integers, and uses 10 minus the absolute value of the difference as points, or zero if that number is negative:

```
struct EqualPCore: public RichBool::FunctorCore
{
 typedef unsigned int Points;

 bool operator()(int a, int b) const
 {
  return a==b;
 }

 const char* GetTextBefore2() const
 {
  return " == ";
 }

 Points GetPoints(int a, int b) const
 {
  int diff = abs(a-b);
  return diff > 10 ? 0 : 10-diff;
 }
 Points GetGood() const
 {
  return 10;
 }
};

template <class GV1=RichBool::Value, class GV2=RichBool::Value,
  class Stringize=RichBool::MakeString>
struct EqualP: public
  RichBool::Wrapper2Arg<EqualPCore, GV1, GV2, Stringize>
{
  EqualP(GV1 gv1=GV1(), GV2 gv2=GV2()):
    RichBool::Wrapper2Arg<EqualModulo5Check, GV1, GV2, Stringize>(gv1, gv2)
  {}
};

#define rbEQUALP(a,b) rb2_RB(a,b, EqualP())
#define rbvEQUALP(a,b) rbv2_RB(a,b, EqualP())
```

This can also be done with functor cores that are derived from `RichBool::CustomFunctorCore`.

Writing a Rich Boolean Functor without the wrappers

In case that using a wrapper is not sufficient for you (which is unlikely), or you need to adjust a custom Rich Boolean Functor that was made before the Rich Boolean Functor Wrappers were introcduced, you can do so as explained in this section. A Rich Boolean Functor class should have the members `bool operator(...) const` and and `RichBool::Analysis* Analyse(...) const`. `bool operator(...) const` has the same arguments as the rich boolean macro (these can be of course, and often are, template arguments), and should return whether the condition is true or not. `RichBool::Analysis* Analyse(...) const` also has the same arguments as the rich boolean macro, followed by a number of `Expr` arguments, which should be templates (one for each argument of the Rich Boolean macro), and an extra argument, `bool analyseOnSucceed`. It should return `NULL` if the condition is true and `analyseOnSucceed` is false, and a non-NULL pointer to a `RichBool::Analysis` object otherwise. This object could be of a class that is provided with the RichBool package (`RichBool::GeneralAnalysis` can be useful in many cases), or your own type (in this case, remember to let `GetType()` return a string that is unique to your class, code that processes `RichBool::Analysis` objects like dialog boxes and loggers may rely on this).

The `Expr` arguments can be converted to `Expression` objects with the function `RichBool::MakeExpression`, that takes such an argument and returns a `Expression` object. Objects of the class `Expression` represent the expression of an argument of a rich boolean. It can be converted to a string object. `Expression` objects therefore allow to delay creation of a string until it is needed, to improve performance. In this version of the Rich Booleans they can be constructed with a string object or a `const char*` (when a rich boolean is used directly) or a `size_t` (when it is in a range). They can also be converted to `bool`; if they were created with the default constructor, they return `false` which means there is no expression, otherwise they return `true`.

If your rich boolean has two arguments, and you plan to use the rich boolean class in the comparison of two ranges (e.g. with `Compare<...>`), you should add 4 methods. The first two are `GetString1` and `GetString2`, that take one argument, respectively the first and second argument of the Rich Boolean, and return a string object. The easiest is to return `RichBool::ToString(arg)`. The next two methods are `IsValid1` and `IsValid2`, that take one argument, respectively the first and second argument of the Rich Boolean, and return a boolean that indicates whether the value can be dereferenced well. You should also add a typedef `Points` that tells which scoring mechanism is used to dynamically find which elements in the ranges should be matched with each other. For now we will use `bool`, that simply tells whether two elements match or not.

So usually a Rich Boolean Functor class looks like

```
class MyRichBool
{
public:
    typedef RichBool::MakeString Stringize1;
    typedef RichBool::MakeString Stringize2;
    typedef bool Points;

    bool operator(int a, int b) const
    {
        ...
    }
    template <typename Expr1, typename Expr2>
    RichBool::Analysis* Analyse(int a, int b,
  Expr1 expr1, Expr2 expr2,
  bool analyseIfSuccess) const
    {
        ...
    }
    std::string GetString1(int a) const
    {
        return RichBool::ToString(a);
    }
    std::string GetString2(int b) const
    {
        return RichBool::ToString(b);
    }
    bool IsValid1(int ) const
    {
        // we don't dereference anything, so always return true
        return true;
    }
    bool IsValid2(int ) const
    {
        // we don't dereference anything, so always return true
        return true;
    }
};

#define rbMY_RICH_BOOL(a,b) rb2_RB(a,b, MyRichBool())
#define rbvMY_RICH_BOOL(a,b) rbv2_1_RB(a,b, MyRichBool())
```

When comparing ranges, it might be interesting to indicate whether there is a partial match. When several elements don't match, a partial match will be shown in preference. To do this, change the typedef of `Points` in your Rich Boolean Functor class from `bool` to a numeric type. Furthermore you should add the methods `Points GetPoints(const T &t1, const T &t2) const`, that returns a number that is higher as the match is better, and `Points GetGood()` that returns the maximum value that `GetPoints` can return.

Suppose you need a class Person:

```
class Person
{
public:
    std::string firstName, surName;
    Date birthDate;
};
```

If you often need to check if two `Person` objects are the same, you could write a custom Rich Boolean:

Example 44. A Rich Boolean for the class `Person`

```
class EqualPersons
{
  typedef bool Points;
  bool operator()(const Person &person1, const Person &person2) const
  {
    return person1==person2;
  }

  template <typename Expr1, typename Expr2>
  RichBool::Analysis* Analyse(const Person &person1, const Person &person2,
            Expr1 expr1, Expr2 expr2,
            bool analyseOnSucceed=false) const
  {
    bool ok = person1==person2;
    if (ok && !analyseOnSucceed)
        return 0;
    RichBool::GeneralAnalysis *analysis = new RichBool::GeneralAnalysis(ok);
    if (expr1 && expr2)
    {
      analysis->AddText("Comparison of ")
        ->AddExpression(RichBool::MakeExpression(expr1))
        ->AddText(" and ")
        ->AddExpression(RichBool::MakeExpression(expr2))
        ->AddText(":")->AddNewLine();
    }

    analysis->("first names: ")
      ->AddAnalysis(
        RichBool::Equal<>().Analyse(person1.firstName, person2.firstName,
          "1", "2", true))
      ->AddNewLine();
    analysis->AddText("surnames: ")
      ->AddAnalysis(
        RichBool::Equal<>().Analyse(person1.surName, person2.surName,
          "1", "2", true))
      ->AddNewLine();
    analysis->AddText("birthdates: ")
      ->AddAnalysis(
        RichBool::Equal<>().Analyse(person1.birthDate, person2.birthDate,
          "1", "2", true))
      ->AddNewLine();
    return analysis;
  }
  std::string GetString1(const Person &person1) const
  {
    return RichBool::ToString(person1);
  }
  std::string GetString2(const Person &person2) const
  {
    return RichBool::ToString(person2);
  }
  // we don't dereference anything, so always return true
  bool IsValid1(const Person &) const { return true; }
  bool IsValid2(const Person &) const { return true; }
};

#define EQUAL_PERSONS(person1, person2) \
 rb2_RB(person1, person2, EqualPersons())
```

Note that the constructor of most classes derived from `RichBool::Analysis` have a boolean as argument, that indicates whether the condition succeeded or not.

Also note that we give text strings to `RichBool::Equal::Analyse` where it takes template `Expr` arguments, just like `Person::Analyse`. We could also pass `expr1` and `expr2`, but that may be a long expression, and is already in the analysis.

Also note that the last argument that we pass to `RichBool::Equal::Analyse` is `true`, which means that we want an analysis even if the comparison of that part succeeds. This makes the debugging information more useful.

If you want to take advantage of partial matching when two ranges of `Person` objects are compared, the class `EqualPersons` should be changed to this:

Example 45.

```
class EqualPersons
{
public:
  typedef int Points;

  Points GetPoints(const Person &person1, const Person &person2) const
  {
    return
      (person1.firstName==person2.firstName ? 1 : 0)+
      (person1.surName  ==person2.surName   ? 1 : 0)+
      (person1.birthDate==person2.birthDate ? 1 : 0);
  }

  Points GetGood()
  {
    return 3;
  }

  ...
};
```

Creating a Rich Boolean without the macros rbn_RB

In exceptional cases you may need to write a Rich Boolean for which you can't use a Rich Boolean Functor, so you can't use the macros `rbn_RB` as described above. Some examples of such Rich Booleans are `rbAND` and `rbOR`.

If you do this, you should know that a Rich Boolean macro should evaluate to a `RichBool::TmpBool` object. This is a simple class that contains a pointer to an object of the abstract type `RichBool::Analysis` and a boolean. This class is necessary to prevent accidental use with boolean operators, which would cast the pointer to `true` or `false`. The class also allows to assign boolean values to it, so boolean expressions can still be used instead of Rich Booleans. Objects of this class can only be constructed with a boolean or a pointer to an object of the type `RichBool::Analysis` as argument, which can be `NULL`. If a `RichBool::Analysis` pointer is given, a `NULL` pointer means success, a non-`NULL` pointer can mean failure or success, depending on what `GetState()` returns.

A Rich Boolean macro should check the value of `richbool_level`, which tells how specific the evaluation should be:

- 0 means that only the arguments should be evaluated; you should return `RichBool::TmpBool(true)`
- 1 means that the arguments and the condition should be evaluated, but no analysis should be created; you should return `RichBool::TmpBool(b)`, where `b` states whether the condition succeeded
- 2 means that the arguments and the condition should be evaluated, and an analysis should be created if the condition failed; you should return `RichBool::TmpBool(analysis)`, where `analysis` is NULL if the condition failed
- 3 means that the arguments and the condition should be evaluated, and an analysis should be created, even if the condition succeeds; you should return `RichBool::TmpBool(analysis)`, where `analysis` is the analysis, that is not NULL

Creating an analysis

Analyses should be given as pointers to objects of classes that are derived from `RichBool::Analysis`. If you derive a class from this class, you should implement three virtual methods:

- `void StreamOut(std::ostream &stream, int indent) const`: this method should stream out the analysis to the stream. `indent` tells how many spaces should come after a newline. It should not end with a newline or whitespace.
- `const std::string& GetType() const`: this method should return a string that uniquely identifies the type.
- `bool Equals(const Analysis* analysis) const`: this protected method should return whether it is equal to `analysis`. The method can safely assume that `analysis` is of the same type, so it can be casted to the derived type.

`RichBool::Analysis` has two constructors. The first takes a boolean that indicates whether the condition was true or false. The second takes an enumerated value `RichBool::Analysis::State`, that can be `Ok`, `NotOk` or `NotEvaluated`. The last is used in Rich Booleans that should evaluate more than one condition, but don't always do them all due to short circuiting. Note that `NotOk` could mean that the condition failed or that one of the arguments is a non-dereferenceable value; in a future version there will be a distinction between these.

You can also use one of the provided classes that are derived from `RichBool::Analysis` discussed below.

GeneralAnalysis

This is a class that can contain plain text, values, expressions and other analyses. It is easier to use it with the class `RichBool::MakeGeneralAnalysis`. Its constructor takes a boolean that indicates whether the condition succeeded, and creates a `RichBool::GeneralAnalysis` on the heap. This object is returned when the `RichBool::MakeGeneralAnalysis` object is casted to a pointer to `RichBool::Analysis`. It has these methods, that all return a reference to the object itself, so the methods can be used in a chain:

- `operator()(const detail::String &str, bool validPtr)`: this adds the value of an argument of the Rich Boolean converted to a string to the analysis; the boolean indicates whether it could be safely dereferenced

- `template<typename T, class GetValue, class Stringize_> operator()(const T &t, GetValue getValue, Stringize_ str)`: this converts the first argument to a string using the second and third argument, and adds the string to the analysis
- `operator()(const char *text)`: this adds plain text to the analysis
- `operator()(const std::string &text)`: this adds plain text to the analysis
- `expr(SharedExpression expression, const char *sz)`: this adds an expression and a separator (e.g. ":") to the analysis
- `idx(int n, const char *sz)`: this adds an index expression and a separator (e.g. ":") to the analysis
- `expr(const char *expression, const char *sz)`: this adds a textual expression and a separator (e.g. ":") to the analysis
- `result(const std::string &text)`: this adds the value of a result to the analysis, typically something that was calculated from the arguments of the Rich Boolean
- `operator()(SharedAnalysis otherAnalysis)`: this adds an analysis to the analysis
- `operator()()`: this adds a newline to the analysis

CombinedAnalysis

The constructor of this class takes two `RichBool::Bool` objects each containing the result of a condition (from a Rich Boolean or a boolean expression) and a boolean that tells whether the combined condition is true or not.

OrAnalysis

This class is derived from `RichBool::CombinedAnalysis`. Its constructor takes two `RichBool::Bool` objects each containing the result of a condition (from a Rich Boolean or a boolean expression).

AndAnalysis

This class is derived from `RichBool::CombinedAnalysis`. Its constructor takes two `RichBool::Bool` objects each containing the result of a condition (from a Rich Boolean or a boolean expression).

XorAnalysis

This class is derived from `RichBool::CombinedAnalysis`. Its constructor takes two `RichBool::Bool` objects each containing the result of a condition (from a Rich Boolean or a boolean expression).

BriefAnalysisOfTwoSequences

This class is used to compare two sequences, where each element of the sequence and the connection between two corresponding elements of the sequences, can be represented by a single character (e.g. a string or bits). It has two constructors. The first constructor takes two `RichBool::SharedExpression` objects, that represent the two sequences, and three `const char *` strings, with the text that should be added before, in between and after these expressions. The second constructor takes three `RichBool::SharedExpression` objects, that represent the two sequences and additional data (e.g. a buffer size), and four `const char *` strings, with text that should be added before, in between and after these expressions, and after the additional data. It has these methods:

- `void SetString1(const std::string &str)`: this sets the first sequence, where every element of the sequence is represented by one character
- `void SetString2(const std::string &str)`: this sets the second sequence, where every element of the sequence is represented by one character
- `void SetStringDiff(const std::string &str)`: this sets the connections between elements of the sequences, where every connection is represented by one character
- `void SetTitle1(const char *title)`: this sets the title shown before the first sequence; this should be short
- `void SetTitle2(const char *title)`: this sets the title shown before the second sequence; this should be short
- `void SetTitleDiff(const char *title)`: this sets the title shown before the connections of the sequences; this should be short
- `void SetBlockSize(int blockSize)`: this sets the size of the blocks in which the sequences are shown; each block is separated by a space
- `void SetBlocksPerLine(int blocksPerLine)`: sets the number of blocks shown per line

If you don't call `void SetBlockSize(int blockSize)` and `void SetBlocksPerLine(int blocksPerLine)`, or you set them both to 0, the sequences are shown on one line. `void SetTitle1(const char *title)`, `void SetTitle2(const char *title)` and `void SetTitleDiff(const char *title)` should have text of the same length.

Single

This class is used when two ranges are compared, and an element cannot be matched or even mismatched to an element of the other range (due to dynamic matching or an abundance in one of the ranges). Its state can only be set in its two constructors:

- `Single(int idx, int total, const detail::String &value, SharedExpression expr, bool valid)`: `idx` indicates the range (usually 0 or 1), `total` the number of ranges (usually 2), `value` the string with the data of the element, `expr` the expression of the element, and `valid` indicates whether it could be dereferenced well.
- `template <typename T> Single(const T &t, int idx, int total, const detail::String &value, SharedExpression expr, bool valid)`: `idx` indicates the range (usually 0 or 1), `total` the number of ranges (usually 2), `value` the string with the data of the element, `expr` the expression of the element, and `valid` indicates whether it could be dereferenced well.

Writing macros that use rich booleans (advanced)

Rich Booleans are not straight forward to use directly, so it's best to write a macro to use them. If you use a library with macros that can have Rich Booleans as an argument (such as ModAssert and UquoniTest), you don't have to read this section.

Macros that have a Rich Boolean as an argument (or one of their arguments), must make sure that a non-constant integer called `richbool_level` is declared (locally or globally), as well as two `RichBool::TmpBool` objects called `richbool_tmp1` and `richbool_tmp2` (preferably locally). `richbool_level` tells the Rich Boolean macro whether it should evaluate its arguments and its condition.

- 0 means that only the arguments will be evaluated; in this case the rich boolean will always evaluate to true
- 1 means that the arguments and the condition will be evaluated, but no analysis will be created
- 2 means that the arguments and the condition will be evaluated, and an analysis will be created if the condition failed
- 3 means that the arguments and the condition will be evaluated, and an analysis will be created, even if the condition succeeds

Therefore Rich Booleans are usually used inside macros, that define a small local scope where these variables are defined.

Rich Booleans usually evaluate to a `RichBool::TmpBool`, sometimes to a `RichBool::Bool` object, which both can be assigned to a `RichBool::Bool` object. Boolean expressions can also be assigned to such objects, so they both can be used. `RichBool::Bool` has a method `bool operator() const`, that tells whether the expression evaluated to `true` or `false`, and a method `const SharedAnalysis& GetAnalysis() const`, that returns the `RichBool::Analysis` object in a shared pointer if a Rich Boolean was assigned that failed (if a boolean was assigned, it is always `NULL`).

Here is an example of how a macro called `CHECK` could use a Rich Boolean:

Example 46. Writing a macro that processes a Rich Boolean macro

```
#define CHECK(condition) \
 do { \
  int richbool_level = 2; \
  RichBool::TmpBool richbool_tmp1, richbool_tmp2; \
  RichBool::Bool richbool = (condition); \
  if (!richbool()) \
   OnFailure(#condition, richbool.GetAnalysis(), \
       __FILE__, __LINE__); \
 } while (false)
```

Processing Rich Booleans (advanced)

Eventually, the macros that use Rich Booleans will pass these on to display or log the information in it. If you use a library that processes Rich Booleans, you don't have to worry about this.

A Rich Boolean macro evaluates to a `RichBool::Bool` or `RichBool::TmpBool` object, which both can be assigned to a `RichBool::Bool` object. A `RichBool::Bool` has a method `const SharedAnalysis& GetAnalysis() const`, that returns a shared pointer that holds a `RichBool::Analysis` object. The shared pointer has the `operator->()`, so all methods of `RichBool::Analysis` can be used through that. It also has `operator bool()`, which is false if a `NULL` pointer was given, so `!analysis` can be used to check if that was the case.

The class `RichBool::Analysis` has a virtual method `void streamout(std::ostream &stream) const`, which is called by `std::ostream & operator<<(std::ostream &stream, const RichBool::Analysis &analysis)` (`std::ostream` is replaced by `wxTextOutputStream` if you use wxWidgets). Derived classes implement this method to give the debugging information in plain text. This way you can stream the data to a file, a string or to the screen. The method `RichBool::ToString` can convert a `RichBool::Analysis` object to a string using `operator<<` (but it's more efficient to stream it to a stream with operator<<).

However, depending on the medium where the information is stored or displayed, it may be interesting to cast to the actual class, and take advantage of the mediums functionality. E.g. when you store the information in a HTML page, you could use colors or bold text to emphasize the values in a `RichBool::GeneralAnalysis` object, and distinguish between valid and invalid values. For this purpose the base class `RichBool::Analysis` has a virtual method `const std::string& GetType() const` (or `const wxString& GetType() const` if you use wxWidgets). In each derived class a unique name is returned here. A more efficient way to do this would be to return an enum, but since `RichBool::Analysis` derived classes can be developed independently from the classes that process them, this wouldn't be easy. Another way to do it would be by using RTTI, but since not everyone likes to use that, this alternative was added.

Considerations related to macros

Some programmers avoid macros because of the problems that are associated with them. However, the macros in the Rich Booleans package are constructed with great care, to avoid these dangers.

- The arguments of the macros are evaluated only once. Therefore constructions like `rbEQUAL(a++, b++)` are safe and do what you would expect. But remember that if you use them in an assertion macro, e.g. `MOD_ASSERT`, the Rich Boolean may be eliminated, so then it is best to have no side effects in the Rich Boolean, but it is still safe in a `MOD_VERIFY` or `MOD_CHECK` macro, because in these they will always be evaluated.
- The arguments of the macros are separated by commas and parentheses from other expressions in the macros, so they can't interfere with these other expressions in the macros
- The Rich Boolean macros cannot be used as statements, so the problem of using a macro in a if-else construct is not of concern here

Thread safety

The Rich Booleans package is unaware of threads, i.e. it doesn't lock.

Normally, if a Rich Boolean creates an analysis object, which is held in a `SharedAnalysis` object, this analysis is processed on the same thread and destroyed after that. E.g. ModAssert and UquoniTest do this. Because it usually stays on the same thread, a shared pointer implementation that is not thread safe was chosen for better performance.

If for some reason you do need the analysis in another thread, make sure that no `SharedAnalysis` objects pointing to the same analysis are created or released at the same time on different threads. Otherwise you can clone the analysis by calling its method `Clone()`, and pass it to a `SharedAnalysis` object on the other thread.

If this recommendation is followed, Rich Booleans is thread safe

Exception safety

The Rich Booleans package never throws an exception itself.

Many template methods will call your code, which may throw exceptions. Where necessary, care is taken that no resources will leak. Code in the Rich Booleans package doesn't catch exceptions that are thrown in your code.

Warning levels

Where possible, the Rich Booleans are compiled with the highest warning levels. In some cases this was not possible:

- With Visual C++ 6.0, there are too many warnings in the STL headers, so it is set to level 3 for building the library, and probably also best for code that uses Rich Booleans
- With any version of Visual C++ there will be a warning about unreachable code in `richbool/getvalue.hpp`
- With gcc, -W and -Wall should not be used in code that uses Rich Boolean macros, because there will be warnings about the lefthand of a comma operator not having any effect. The flags -ansi -pedantic -Wconversion -Wshadow -Wcast-qual -Wwrite-strings can be used, the Rich Booleans don't produce warnings if these are enabled. But -pedantic should be omitted if you use wxWidgets, because this gives an error in some of the header files of wxWidgets.

Converting objects to a string

Rich Booleans convert values to a string to provide extra information. By default the string type is `std::string`, but if you defined the symbol `RICHBOOL_USE_WX` it is `wxString`. `RichBool::detail::String` is typedef'ed to the appropriate one of these two classes, and is used internally throughout the Rich Booleans package, but you never need this type directly, except when you write extensions that need to work in both environments. Normally you should use `std::string` or `wxString` if you write extensions, depending on which you need.

In the default situation the STL mechanism is used to convert to a `std::string` object, i.e. with `std::ostream& operator<<(std::ostream &os, const T &obj)`, and a `std::ostringstream` is given as the first argument. If you use wxWidgets and therefore defined the symbol `RICHBOOL_USE_WX`, the overloaded function `wxTextOutputStream& operator<<(wxTextOutputStream &str, const T &obj)` is used instead, and a `wxTextOutputStream` object that writes to a `wxMemoryOutputStream` object is given as the first argument, to convert the object to a `wxString` object. All the headerfiles that the richbool headerfiles need, are then included automatically, provided you added the necessary include path.

With both mechanisms, primitive types can already be converted to strings, as well as string objects and `const char*` values, without the need for writing conversion functions.

One obvious way to be able to use objects in Rich Booleans, is to overload the streamout operator. This is most convenient, because you can reuse that in other situations.

Another way to do it is to overload the function `RichBool::ToString`, that should have one argument, an object of your class (usually by const reference), and return a `std::string` (or `wxString` if you use wxWidgets). This can be handy if you want objects of a certain class streamed in a different way for Rich Booleans.

Example 47. Overloading RichBool::ToString

```
class MyClass
{
public:
 // ...
};

namespace RichBool
{
 inline std::string ToString(const MyClass &obj)
 {
  // ...
 }
}
```

Note: here it is done inline, but this is of course not necessary.

STL containers

If you use the STL mechanism, `RichBool::ToString` is already overloaded for `std::vector`, `std::list`, `std::deque`, `std::set` `std::multiset`, `std::map` and `std::multimap`.

Compilers that can't do partial template specialization

However, if your compiler can't do partial template specialization, this is only done for containers with default allocators and predicates that contain `char`, `short`, `int`, `long`, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, `double`, `std::string` or `const char*`, and not at all for `std::map` and `std::multimap`. For other containers, use the appropriate macro from the list below at global scope. From that point on, RichBool can convert containers of that type to a string.

- If you want to do it for another non-associative container, use the macro `RICHBOOL_CONTAINER_TO_STRING` with the container type as the argument.
- For maps with default predicate and allocator, use the macro `RICHBOOL_STD_MAP_TO_STRING` with the keytype and valuetype as arguments.
- For maps with a non-default predicate and the default allocator, use the macro `RICHBOOL_STD_MAP_TO_STRING_P` with the keytype, valuetype and predicate type as arguments.

- For maps with a non-default predicate and allocator, use the macro `RICHBOOL_STD_MAP_TO_STRING_PA` with the keytype, valuetype, predicate type and allocator type as arguments.
- For multimaps with default predicate and allocator, use the macro `RICHBOOL_STD_MULTIMAP_TO_STRING` with the keytype and valuetype as arguments.
- For multimaps with a non-default predicate and the default allocator, use the macro `RICHBOOL_STD_MULTIMAP_TO_STRING_P` with the keytype, valuetype and predicate type as arguments.
- For multimaps with a non-default predicate and allocator, use the macro `RICHBOOL_STD_MULTIMAP_TO_STRING_PA` with the keytype, valuetype, predicate type and allocator type as arguments.
- For containers that contains pairs, e.g. a vector of pairs, or a map where the keytype or valuetype is a pair, additionally use the macro `RICHBOOL_PAIR_TO_STRING` with the two types as the arguments.

Example 48. Defining a STL container to string conversion

```
#include <richbool/richbool.hpp>

RICHBOOL_CONTAINER_TO_STRING(std::vector<MyClass>)

void foo()
{
 std::vector<MyClass> vec1, vec2;
 // ...
 MOD_ASSERT(rb2_PRED(vec1, vec2, MyPredicate()));
}

typedef std::pair<float, std::string> PairFloatString;
RICHBOOL_PAIR_TO_STRING(float, std::string)
RICHBOOL_STD_MAP_TO_STRING(int, PairFloatString)

void bar()
{
 std::map<int, std::pair<float, std::string> > mymap;
 // ...
 MOD_ASSERT(rb1_PRED(mymap, MyMapPredicate()));
}
```

Note: these macros have no effect with compilers that can do partial template specialization, so you can use them in code that should compile under different compilers.

Containers with overloaded non-member functions begin and end

If you use the WxWidgets mechanism, by default no containers can be converted to a string. The same applies for other containers. However, if you have overloaded non-member functions `begin` and `end` that return iterators over the whole container (see the section called "Containers with overloaded funtions `begin()` and `end()`"), you can do it with the macro `RICHBOOL_XCONTAINER_TO_STRING` with the container type as the argument.

Example 49. Defining a WxWidgets container to string conversion

```
#include <richbool/richbool.hpp>

WX_DEFINE_ARRAY_INT(int, IntArray);
WX_DEFINE_ARRAY_ITERATOR_P(int, IntArray, IntArrayIterator)
RICHBOOL_XCONTAINER_TO_STRING(IntArray)

void foo()
{
 IntArray vec1, vec2;
 // ...
 MOD_ASSERT(rb2_PRED(vec1, vec2, MyPredicate()));
}
```

When converting containers to a string is needed

Note however that all this is not necessary if you use Rich Booleans that work on containers like `rbIN_CONTAINER`. It might however be necessary if you create custom Rich Booleans that work on containers, or a Rich Boolean that takes a predicate that works on a container, like with `rb1_PRED`. You would also need it with some other Rich Booleans that can have a container as an argument, e.g. `rbEQUAL`, but `rbIN_CONTAINERS` is better suited for that.

Windows types that can be converted to a string

On windows, `LARGE_INTEGER` and `ULARGE_INTEGER` have an overloaded function `RichBool::ToString`, so you can use these in Rich Booleans.

If you use MFC, you can also convert objects of the type `POINT`, `CPoint`, `SIZE`, `CSize`, `RECT`, `CRect` and `CTime` to a string. Include the file `richbool/mfcstream.hpp` to do so.

The package RichBoolTest (advanced)

RichBoolTest, which is in the same download, is a package that contains tests for every Rich Boolean in this package. This is mainly of interest if you want to write Rich Booleans, but also if you want to see examples of how Rich Booleans behave.

If you want to write Rich Booleans, you should have a look at the RichBoolTest package. This is a simple testing framework for Rich Booleans. The tests are easy to understand by looking at the existing tests. The tests are self registering, i.e. if you write

```
RB_TEST(TestName)
{
    // your tests here
)
```

in a .cpp file, the tests in the body will be automatically executed if the program is launched.

In tests, there are two types of checks: `RB_PASS` and `RB_FAIL`. The first has a rich boolean macro as its argument, and checks whether it passes. The second has a rich boolean macro and a pointer to an

`RichBool::Analysis` derived object as its arguments, and checks whether the rich boolean macro fails indeed with the given `RichBool::Analysis` derived object.

Every test is run four times, each time with a different value for richbool_level. It is checked that the Rich Booleans behave as expected for that value of richbool_level, e.g. no analysis when it is 0 or 1, always an analysis when it is 3.

Index

A

Adding partial matching, 85
Adjacent, 48
All, 45
AllEqual, 52
AllEqualUnordered, 54
AllPairs, 49
AllUnique, 44
Analysis, creating, 90
Analysis::State, 90
AndAnalysis, 91

B

BasicCollate, 30
BasicCollateToLower, 30
BasicCollateToUpper, 30
BasicStrcmp, 29
BasicStringCompare, 29
BasicStringCompareToLower, 29
BasicStringCompareToUpper, 29
BindArg, 77
Binding parameters on a Rich Boolean functor, 77
Bool, 93, 93
BriefAnalysisOfTwoSequences, 91

C

CallMethod, 72
CallMethodArg, 72
Checking a range, 42
CombinedAnalysis, 91
Combining Rich Boolean functors in logical operations, 76
Compare, 57
CompareUnordered, 58
Comparing strings, 27
Comparing two ranges, 51
Comparing values, 16
Composing getters, 73
Considerations related to macros, 94
Creating a Rich Boolean Functor without the wrappers, 86
Creating a Rich Boolean without the macros rbn_RB, 89
Creating an analysis, 90
Creating macros that use rich booleans, 93
Creating your own rich booleans, 79

CStringACollate, 30
CStringACollateNoCase, 30
CStringACompare, 30
CStringACompareNoCase, 30
CStringCollate, 30
CStringCollateNoCase, 30
CStringCompare, 30
CStringCompareNoCase, 30
CStringWCollate, 30
CStringWCollateNoCase, 30
CStringWCompare, 30
CStringWCompareNoCase, 30
Custom data retrievers, 73
CustomFunctorCore, 82, 83

D

Defining custom data retrievers, 73
Deprecated Rich Boolean macros, 69
Dereferencing, 70
Dynamic matching, 63

E

EqualPair, 66
Equals, 49
EqualTo, 49
Exception safety, 95
Exceptions, 41
Expression objects, 77

F

File system, 32
FunctorCore, 80

G

GeneralAnalysis, 90
Get1Arg, 78
Get2Args, 78
Get3Args, 78
Get4Args, 78
Get5Args, 78
GetFileLength, 35
GetMember, 71
GetMemberArg, 72
GetPoints, 85

H

Has, 46

I

IndexExpression, 77
inline container, 66
Installation, 12
IsBlockDevice, 33

IsCharacterDevice, 33
IsDifferentFrom, 49, 49
IsExecutable, 33
IsLessOrEqualTo, 49
IsLessThan, 49
IsLink, 33
IsMoreOrEqualTo, 49
IsMoreThan, 49
IsMultiSubsetOf, 56
IsMultiSubsetOfCustomSorted, 62
IsMultiSubsetOfSorted, 57
IsOrderedSubsetOf, 55
IsOrderedSubsetOfCustomSorted, 63
IsOrderedSubsetOfSorted, 56
IsReadable, 33
IsRegular, 33
IsSubsetOf, 54
IsSubsetOfCustomSorted, 62
IsSubsetOfSorted, 55
IsWritable, 33

L

LessOrEqualTo, 49
LessThan, 49
Logical expressions, 39
Logical operations on Rich Boolean functors, 76

M

MakeGeneralAnalysis, 90
Making a Rich Boolean Functor without the wrappers, 86
Making a Rich Boolean without the macros rbn_RB, 89
Making your own rich booleans, 79
MatchesMultiSubsetOf, 60
MatchesOrderedSubsetOf, 61
MatchesSubsetOf, 59
Member of an object, 71
Method of an object, 72
MFC exceptions, 42
MoreOrEqualTo, 49
MoreThan, 49
Multiple levels of dereferencing, 73

N

NotEvaluated, 90
NotOk, 90

O

Ok, 90
operator!, 76
operator&, 76
operator&&, 76
operator^, 76
operator|, 76

operator||, 76
OrAnalysis, 91

P

Partial matching, 85
Pointer, 70
Pointer safety, 26
PointerLike, 70
PointerLikeToValue, 70
PointerToValue, 70
PrefixedExpression, 77
Processing Rich Booleans, 93

R

rb1_PRED, 26
rb1_RB, 26
rb2_PRED, 26
rb2_RB, 26
rb3_PRED, 26
rb3_RB, 26
rb4_PRED, 26
rb4_RB, 26
rb5_RB, 26
rb6_RB, 26
rbAND, 39
rbAND_BE, 40
rbAND_DE, 40
rbBITS_ARE, 25
rbBITS_OFF, 25
rbBITS_ON, 24
rbDIFF, 17
rbDIFF_NEAR, 20
rbDIFF_REL_NEAR, 23
rbDIRECTORY, 33
rbDIRECTORY_EXISTS, 35
rbDOES_NOT_EXIST, 35
rbDYNAMIC_CASTABLE, 37
rbEQUAL, 16
rbEQUAL_BITWISE, 23
rbEQUAL_DATA, 24
rbEQUAL_DATA_BITWISE, 24
rbEQUAL_PTR, 16
rbEQUAL_TYPES, 36
rbEQUAL_USING_LESS, 17
rbEQUAL_USING_MORE, 18
rbEXCEPTION, 41
rbFILE, 32
rbFILE_EXISTS, 34
rbHAS_REGEXP, 31
rbHAS_REGEXP_F, 32
rbHAS_TYPE, 36
rbIN_ARRAY, 42
rbIN_ARRAYS, 51

rbIN_ARRAY_CONTAINER, 65
rbIN_ARRAY_RANGE, 51
rbIN_ARRAY_VALUES, 52
rbIN_ARRAY_XCONTAINER, 69
rbIN_CONTAINER, 64
rbIN_CONTAINERS, 64
rbIN_CONTAINERS_ARG, 69
rbIN_CONTAINER_ARG, 69
rbIN_CONTAINER_ARRAY, 65
rbIN_CONTAINER_ARRAY_ARG, 69
rbIN_CONTAINER_RANGE, 65
rbIN_CONTAINER_VALUES, 66
rbIN_INTERVAL, 18
rbIN_INTERVAL_NEAR, 21
rbIN_OPEN_INTERVAL, 18
rbIN_RANGE, 42
rbIN_RANGES, 51
rbIN_RANGES_ARG, 69
rbIN_RANGE_ARG, 69
rbIN_RANGE_ARRAY, 51
rbIN_RANGE_ARRAY_ARG, 69
rbIN_RANGE_CONTAINER, 65
rbIN_RANGE_VALUES, 52
rbIN_RANGE_XCONTAINER, 68
rbIN_VALUES, 42
rbIN_VALUES_ARRAY, 52
rbIN_VALUES_CONTAINER, 66
rbIN_VALUES_RANGE, 52
rbIN_VALUES_VALUES, 52
rbIN_XCONTAINER, 68
rbIN_XCONTAINERS, 68
rbIN_XCONTAINERS_ARG, 69
rbIN_XCONTAINER_ARG, 69
rbIN_XCONTAINER_ARRAY, 69
rbIN_XCONTAINER_ARRAY_ARG, 69
rbIN_XCONTAINER_RANGE, 68
rbLESS, 16
rbLESS_NEAR, 20
rbLESS_OR_EQUAL, 17
rbLESS_REL_NEAR, 22
rbMORE, 17
rbMORE_NEAR, 20
rbMORE_OR_EQUAL, 17
rbMORE_REL_NEAR, 23
rbNEAR, 19
rbOR, 39
rbORDER2, 19
rbORDER2_NEAR, 21
rbORDER3, 19
rbORDER3_NEAR, 21
rbORDER4, 19
rbORDER4_NEAR, 22
rbOR_BE, 40
rbOR_DE, 40

rbREGEXP, 31
rbREGEXP_F, 31
rbREL_NEAR, 22
rbSTD_STRING, 29
rbSTD_WSTRING, 29
rbSTRCMP, 29
rbSTRING, 27
rbSTRING_BEGINS_WITH, 28
rbSTRING_BEGINS_WITH_CI, 28
rbSTRING_BEGINS_WITH_CS, 28
rbSTRING_CI, 27
rbSTRING_CONTAINS, 28
rbSTRING_CONTAINS_CI, 28
rbSTRING_CONTAINS_CS, 28
rbSTRING_CS, 27
rbSTRING_ENDS_WITH, 28
rbSTRING_ENDS_WITH_CI, 28
rbSTRING_ENDS_WITH_CS, 28
rbWCSCMP, 29
rbWX_EQUAL_TYPES, 37
rbWX_HAS_TYPE, 38
rbWX_IS_KIND_OF, 38
rbXOR, 39
Removing the text from analyses, 79
Rich Boolean functors in logical operations, 76
Rich Boolean functors in Rich Boolean macros, 70
Rich Boolean macros, 15
    Checking a range, 42
    Comparing strings, 27
    Comparing two ranges, 51
    Comparing values, 16
    Deprecated, 69
    Exceptions, 41
    File system, 32
    Logical expressions, 39
    Type checking, 36
    Type checking with wxWidgets, 37
richbool_level, 93
RICHBOOL_MAKE_EXCEPTION_TYPE_NAME, 41
RICHBOOL_NO_TEXT, 79
richbool_tmp1, 93
richbool_tmp2, 93

S

Selecting arguments, 78
SetFsGid, 34
SetFsUid, 34
SharedExpression, 77
Single, 92
Sorted, 43
SortedStrictly, 44
State, 90
STD exceptions, 42

Strcmp, 29
String type, 14
StringCompare, 29
StringCompareToLower, 29
StringCompareToUpper, 29
StringExpression, 77
StringsEqualCI, 29
StringsEqualCS, 29
SuffixedExpression, 77

T

TextExpression, 77
Thread safety, 94
TmpBool, 93, 93
Type checking, 36
Type checking with wxWidgets, 37

U

Unique, 47

V

Values, 66

W

Warning levels, 95
Wcscmp, 29
Wrappers to make Rich Boolean functor classes, 79
Writing a Rich Boolean Functor without the wrappers, 86
Writing macros that use rich booleans, 93
Writing Rich Boolean functor classes using the wrappers, 79
WStringCompare, 29
WStringCompareToLower, 29
WStringCompareToUpper, 29
WStringsEqualCI, 29
WStringsEqualCS, 29
WxStringCmp, 30
WxStringCmpNoCase, 30
WX_DEFINE_ARRAY_ITERATOR, 67
WX_DEFINE_LIST_ITERATOR, 67

X

XorAnalysis, 91