Modular Assertions version 2.0.6

An avanced assertion package for C++, that can use Rich Booleans

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

<info@q-mentum.com>

Abstract

In C and C++ the traditional assert function is used to tell the user that something unexpected happened, that might cause the program to fail. Macros were written in other packages to extend this mechanism, like `ASSERT_EQUAL(a,b)`, `ASSERT_NOT_EQUAL(a,b)`, `ASSERT_LESS(a,b)`, ..., to give a short explanation like "lefthand is <9>, righthand is <10>". However, if we want similar VERIFY macros which still evaluate their arguments in non-debug mode, we get `VERIFY_EQUAL`, `VERIFY_NOT_EQUAL`, ..., or macros that add a message like `ASSERT_EQUAL_MSG`, `ASSERT_NOT_EQUAL_MSG`, ..., so we could end up with hundreds of macros. Clearly this is not practical.

The Modular Assertion macros solve this problem. They cannot only have a simple boolean expression as their argument, but also a Rich Boolean, that checks a condition, and if it fails, it holds an analysis of why it fails. E.g., the Rich Boolean `rbEQUAL(a,b)` could hold the information "`a':<1> == `b':<2> nok" after a failed assertion. The assertion macros themselves only vary in their extra arguments (expressions, messages, level, group, optional action). This package has 128 different types of assertion macros (`MOD_ASSERT`, `MOD_VERIFY`, `MOD_CHECK`, `MOD_ASSERT_P`, ...). So one could write `MOD_ASSERT(rbEQUAL(a,b))`, `MOD_VERIFY(rbLESS(foo(), 10))`, `MOD_ASSERT_PG(a, ModAssert::Fatal, rbEQUAL(bar(a),b))` etc. 128 may seem like too much, but there are 8 basic assertions, that each have 4 optional attributes (making 16 variations), so they are easy to learn.

The macros allow an arbitray number of expressions to be evaluated and shown when an assertion or check fails. A level can be assigned to each assertion (Info, Warning, Error or Fatal). Groups can be defined and assigned to assertions. An assertion can have an optional action. Displaying and logging of assertions can each be controlled at runtime per case, level, group, source code file or all at once. This can also be done at compiletime per level, group, source code file or all at once, to reduce the size of the executable.

Failed assertions are processed by objects that implement a given interface, so handling of assertions is pluggable. There is a distinction between objects that inform the user and ask what action to undertake, and objects that simply log the information. Three implementations of the first are provided, one for console applications, one for Win32 and one for wxWidgets. Two implementations of the second are provided, and another specific for wxWidgets. You can add a filter to a responder or a logger, to filter out certain assertions.

ModAssert makes a distinction between assertions and checks. Assertions are meant for detecting bugs, while checks are meant for detecting other errors, e.g. invalid input by the user. Checks have a failure action, that is executed if the condition fails. Loggers and responders are aware of this distinction through an argument.

ModAssert doesn't have compile time assertions, because that is an entirely different kind of assertion.

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

New in version 2.0.6

- The autoconf and automake files are made mostly non-recursive, to solve some problems.

New in version 2.0.5

- Minor adjustments for the next version of UquoniTest.

New in version 2.0.4b

- Some problems with project files were solved.
- Code::Blocks project files for LibInDLL and LibInExe were added.

New in version 2.0.4

- The functions `GetState` and `SetState` are added to use one instance of ModAssert in an executable and the DLLs that are loaded.
- Project files for VC++ 2010 are added.

New in version 2.0.3

- A problem with the configuration scripts is solved.

New in version 2.0.2

- The assertion macro MOD_VERIFY_B and variations are added
- The macros IF_NOT_MOD_ASSERT_REPORTS, IF_MOD_ASSERT_REPORTS_ELSE, IF_NOT_MOD_CHECK_REPORTS and IF_MOD_CHECK_REPORTS_ELSE are added
- The Properties class can tell whether the assertion or check can stop displaying. The provided responders use this feature.

- The provided assertion handlers for consoles, Win32 and WxWidgets now allow to break into code when NDEBUG is defined
- When you remove the text from a condition, it is no longer reported as unconditional
- Bug fixed: the WxWidgets dialog box now correctly uses GetAllowBreakIntoDebugger() on Linux
- Bug fixed: && is now shown correctly in the Windows dialog box

New in version 2.0.1

- The bug that caused a compile error when ModAssert::UseBool is used as a return value of a function, is solved.

New in version 2.0

- There are now variations of `MOD_VERIFY` that return the result of the condition argument, or the first argument of a Rich Boolean if the condition is a Rich Boolean, and variations of `MOD_CHECK` that return a value or a boolean.
- There is now a default responder, that terminates the application if an assertion fails or shows a warning (not if it is a check) depending on the level. You can set the minimum level for both of these.
- There is another default responder, that shows a warning if an assertion fails (not if it is a check). You can set the minimum level for this.
- The class AutoResponder is renamed ResponderSwitcher (but there is a typedef to AutoResponder), and can now also add objects derived from Responder.
- The functions `SetConsoleHandler`, `SetWin32Handler`, and `SetWxHandler` are deprecated (but still available) and are replaced by `SetupForConsole`, `SetupForWin32` and `SetupForWxWidgets`, which show debug information in debug builds and continue silently in release builds. They all have an overloaded version that takes a `Mode` argument, to specify what should happen if an assertion fails: show debug information, show a simple warning, continue silently or terminate.
- The new headerfile `modassert/assertd.hpp` has definitions for `ASSERT` and `VERIFY` to map to ModAssert equivalents.
- The new headerfile `modassert/verifyv.hpp` has definitions for `ASSERT` and a `VERIFY` macro that returns its argument to map to ModAssert equivalents.
- The new headerfile `modassert/wxassert.hpp` has definitions for all runtime assertions in wxWidgets to map them to ModAssert equivalents.
- If ModAssert is not yet active, or no longer active, failing assertions are redirected to assert() (checks are ignored at those stages).
- ModAssert groups that are used as member variables of objects should now be declared mutable if you want to use them in const methods. This makes more sense because it stresses that the group is not logically part of the state of the object, and avoids const-casts in ModAssert.

New in version 1.5.1

- By default remote drives are no longer used in Win32, but you can turn this on.

New in version 1.5

- You can now specify a default optional action per source code file, but still override it if necessary.
- The classes InfoProvider and InfoUser are moved to the namespace InfoProviders, as well as the classes derived from InfoProvider. Their interfaces have changed a little.
- You can define the preprocessor symbol MOD_ASSERT_NO_TEXT to remove the literal text with the condition from assertions and checks, to reduce the size of the application.
- The application infoprovider is added.
- Infoproviders for Win32 that give the computername, the username, information about the available drives, the amount of free space on them, the total amount of RAM and available amount of RAM, are added.
- Filter objects that only allow assertions and checks, or allow all except assertions and checks, are added.
- A bug that caused the info of the Win32 current directory infoprovider to end in a null character, is solved.
- Project files for the Code::Blocks IDE are added.
- The optional logger is now in the ModAssert package itself. The ModAssertConsole, ModAssertWin32 and ModAssertWxGui libraries use that one instead of their own version.
- The optional logger can now have begin and end messages, if desired.
- Loggers can be removed by their id, that is returned by AddLogger(...).
- Setup code in the demoes is moved to a separate function.

New in version 1.4.1

- The headerfiles that cause autolinking for the ModAssertConsole, ModAssertWin32 and ModAssertWxGui libraries, now also cause autolinking for the ModAssert library.

New in version 1.4

- You can now specify default parameters per sourcefile
- You can now specify a global optional action
- For Win32, infoproviders are added that give the current directory, the OS version, and info about the processor
- For wxWidgets, infoproviders are added that give the current directory, the OS version, the last system error and the free memory
- The Filter derived class FilterMessages is added
- InfoUser can now also take InfoProvider objects to include or exclude
- With Visual Studio, auto linking is now used

- Warning levels are increased to the highest level where possible, and most remaining warnings are eliminated

New in version 1.3

- The license has changed to the wxWindows licence
- You can now specify default groups
- The macro MOD_CHECK_FAIL is added
- The interface of InfoProvider has changed.
- Each InfoProvider derived object can determine whether it should be reported in case of an assertion, at the beginning of logging and at the end of logging.
- For each logger and responder you can override whether an InfoProvider should be used with it
- When you add an InfoProvider with ModAssert, you can attach a filter to it. Filters can now also filter out InfoProviders when messages are logged
- You can specify at runtime whether the Exit action, BreakIntoDebugger action and Optional action are allowed by responders. The provided responders take this into account by disabling these options.
- The Win32ProcessIdInfoProvider is added for Win32, and WxProcessIdInfoProvider for WxWidgets platforms
- The InfoProviders that give the treadid, now also tell whether it is the main thread or not
- Adjustments to Rich Booleans 1.3

New in version 1.2.1

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

New in version 1.2

The major change in this release is the strict distinction between assertions and checks, that is introduced in many parts of the library. Unfortunately this means backwards incompatibilty, but only in the backend processing of assertions - the assertion macros haven't changed.

- The class Displayer is renamed to Responder.
- The responder, the loggers, the hooks and the infoproviders are now aware whether the assertion comes from a check macro (i.e. the macros that start with `MOD_CHECK`) or not, and their methods are now const. Unfortunately this changes their interface: their arguments are now grouped in two objects.
- The functions GetDisplayAll, SetDisplayAll, GetLogAll, SetLogAll, SetDisplayInFile, GetDisplayInFile SetLogInFile, GetLogInFile, Level<...>::GetDisplay, Level<...>::SetDisplay, Level<...>::GetLog and Level<...>::SetLog now have an extra argument to distinguish between assertions and checks.
- The definition of the symbols MOD_ASSERT_REPORT, MOD_ASSERT_DONT_REPORT, MOD_ASSERT_REPORT_FILE, MOD_ASSERT_DONT_REPORT_FILE and MOD_ASSERT_LEVEL now only influence assertion macro, not check macros. For check macros there are now the equivalent symbols

MOD_CHECK_REPORT, MOD_CHECK_DONT_REPORT, MOD_CHECK_REPORT_FILE, MOD_CHECK_DONT_REPORT_FILE and MOD_CHECK_LEVEL

- The Responder class's method `OnAssert` also has an extra argument `bool display`, that tells if the assertion should be displayed. ModAssert now always passes assertions to the active responder, so responders have to decide themselves whether they handle an assertion.
- The Logger class now has a new pure virtual method AddMessage(const RichBool::String & msg).
- The inclusion of headerfiles has changed a little. Where you include `modassert/assert.hpp`, you may have to replace it with, or add, `modassert/handler.hpp`.
- The method GetType of InfoProvider now returns by const reference instead of by value.
- You can now add a filter to loggers and the responder, so they're used only for certain types of assertions. Many implementations of these filters are provided.
- The class GroupList now has the method Has(const char*) that returns true only if it has a group with that name.
- The provided responders let you optionally log to another logger, that you can set.
- The loggers and responders for Win32, wxWidgets and console applications are now put in libraries, to make it easier to use them.
- An AutoResponder class is added that implements the Responder interface, that doesn't display anything to the user but decides what response to give based on filters that you add to it.
- The class TimeInfoProvider is added, that inherits from InfoProvider, and gives the date and time. This is automatically used if you call ModAssert::SetWin32Handler, ModAssert::SetWxHandler or ModAssert::SetConsoleHandler.
- The function SetConsoleModAssertHandler is renamed to SetConsoleHandler, and is put in the ModAssert namespace (this was forgotten in the previous release).
- ModAssert now adds a message to a logger when it is added with the date and time, as well as when it is removed.
- Assertions are ignored as long as no responder is set and no logger is added, to prevent that the synchronization mechanism is used before it is initialized.
- Assertions are ignored after the destructor of a ModAssert::AutoShutdown object is called, to prevent that the synchronization mechanism is used after it is disabled.

New in version 1.1

- You can now add hooks that are notified before the loggers and the displayer
- You can now add InfoProviders, to return extra information in a string, that is used by the loggers and the displayer
- A logger is now provided that logs the assertion information to a file, but opens the file for every assertion and appends the information
- The dialogs now show a green icon if an assertion succeeds
- The dialog for Windows 32 now shows the return value of GetLastError() if it is not 0, and the corresponding text.

- The functions SetWin32ModAssertHandler, SetWxModAssertHandler and SetConsoleModAssertHandler are renamed to SetWin32Handler, SetWxHandler and SetConsoleHandler, and are put in the ModAssert namespace
- The ModAssert handler for Win32 now returns 0 if an assertion or check fails in another thread and no debugger is attached

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 `modassert`
- The ModAssert handler for MFC is replaced with a ModAssert handler for Win32, using only the pure Win32 API.
- A bug is solved where the analysis in ModAssert dialogs was not legible on high resolution screens, by setting the font size to 0, so a reasonable default is used.
- A bug is solved where the ModAssert handler for console application allowed '0' as part of the input.
- MOD_ASSERT_ENABLE is renamed to MOD_ASSERT_REPORT, MOD_ASSERT_DISABLE is renamed to MOD_ASSERT_DONT_REPORT, MOD_ASSERT_ENABLE_FILE is renamed to MOD_ASSERT_REPORT_FILE, and MOD_ASSERT_DISABLE_FILE is renamed to MOD_ASSERT_DONT_REPORT_FILE.
- The project files for Visual Studio now have in addition to MultiThread DLL code generation also configurations for MultiThread and Singlethread code generation.
- The macro BREAK_HERE is renamed to MOD_ASSERT_BREAK_HERE.
- Level groups can now only be added to a group with operator%, which allows only one level group per combination.
- Some non-locking methods are added in handler.cpp to prevent a deadlock on platforms that don't have recursive locks

New in version 0.9

- Threadsafety: only one thread at a time can now log and display assertion information
- The level `ModAssert::Debug` is replaced by `ModAssert::Warning`, and level `ModAssert::Error` is now the default level (Debug doesn't sound like a severity level, and Error seems more natural for a default)
- `ModAssert::IfSuccess` and group objects with template argument `ModAssert::ReportAll` now have `operator()` to combine them with other group objects to replace `operator&&`, to preserve their behavior with succeeding assertions
- Group objects with template argument `ModAssert::ReportAll` now appear as two groups when displayed, one for succeeding assertions and one for failing assertions

New in version 0.8.1

- Tests and demos are adjusted to Rich Booleans version 0.8 (the ModAssert library itself is unchanged).

- Breaking into code is tested and works with Dev-C++.

New in version 0.8

- It is now possible to have the information of selected assertions displayed and logged if the conditions succeeds.
- The class `ModAssert::Type` is renamed to `ModAssert::GroupList`, to avoid confusion with the C++ meaning of a type. Therefore the suffix T of the assertion macros is changed to G.
- The class `ModAssert::Off` is renamed to `ModAssert::ReportNone`, and the class `ModAssert::On` is renamed to `ModAssert::ReportFailure`, to avoid confusion with the macros `MOD_ASSERT_ON` and `IF_MOD_ASSERT_ON`.
- The class `ModAssert::ValueList` is renamed to `ModAssert::ParameterList`, to better show the correspondence with the suffix P of the macros.
- A bug that caused a crash when `wxASSERT` (i.e. without a message) is rerouted to MOD_ASSERT, is solved.
- A bug that disabled the option to abort when `wxASSERT` (or another wxWidgets assertion macro) is rerouted to MOD_ASSERT, is solved.
- A bug where the Rich Booleans `rbAND` and `rbOR` couldn't be used in `MOD_VERIFY` and `MOD_CHECK` when assertions are compiled out, is solved.

New in version 0.7

- It is now possible to group assertions by adding a group object to an assertions type. Displaying and logging the assertion data can be set per group. Assertions can also be compiled out per group, drastically improving the possibility to selectively compile out assertions.
- Groups and levels can be combined with && and ||.
- Reusable code is separated from the samples, to make reuse easier (especially for MFC)
- The class `ModAssert::InformUser` is renamed `ModAssert::Displayer`
- The interface of `ModAssert::Displayer` and `ModAssert::Logger` have changed to allow the new functionality of the groups.
- MOD_ASSERT_KEEP is renamed MOD_ASSERT_ENABLE, and MOD_ASSERT_DISABLE is added to compile out assertions when NDEBUG is not defined.
- MOD_ASSERT_ENABLE_FILE and MOD_ASSERT_DISABLE_FILE can be defined in a file to locally turn on or off assertions at compiletime
- Methods to control logging for all assertions or per source code file at runtime, are added
- `ModAssert::SetIgnoreAll` and `ModAssert::GetIgnoreAll` are renamed to `ModAssert::SetDisplayAll` and `ModAssert::GetDisplayAll`, and the meaning of the boolean is inverted
- `ModAssert::SetIgnoreFile` and `ModAssert::GetIgnoreFile` are renamed to `ModAssert::SetDisplayInFile` and `ModAssert::GetDisplayInFile`, and the meaning of the boolean is inverted

- The type `ModAssert::Action` is renamed to `ModAssert::Response`, to avoid confusion with optional actions and failure actions.
- A bug where the failure action was not performed when assertions are compiled out, is solved.

New in version 0.6

- Code for wxWidgets is adjusted to wxWidgets 2.6.x (adjusting include and library directories)
- Adjustments to Rich Booleans 0.6 (some classnames changed)
- It is now possible to specify more than one action if an assertion failed, e.g. 'ignore file' and 'ignore level' at the same time
- Logic to ignore a file or a level is moved to the assertion library
- The suffix L to specify a level in an assertion macro changed to T, because in a future version the argument will be generalized to a type of assertion
- The suffix A to specify an optional action in an assertion macro changed to O, to make the distinction between optional actions and failure actions more clear

New in version 0.5

- A demo console application is added, with a reusable class inherited from `ModAssert::InformUser`
- Header files have now the extension .hpp, because they all contain C++ code
- A bug that caused a chrash when 'debug' is pressed in Release mode, is solved
- A bug that caused a chrash when the wxWidgets assertion dialog was closed in Release mode, is solved

New in version 0.4

- The macro `MOD_CHECK` - and variations with suffixes P, L and A - are added, that have an action as an extra argument that will be executed if the condition fails
- A bug was solved that made it impossible to use `break` as an optional action
- The macro `IF_MOD_ASSERT_ON` is added, that evaluates its argument only if ModAssert is on

New in version 0.3.1

- Sample code is adjusted to changes in Rich Booleans 0.4

New in version 0.3

- The first argument of the macro `MOD_ASSERT_P` and other macros that can have parameters, now has the form `expr1 << expr2 << ... << exprN` instead of `maPARAM(expr1)(expr2)...(exprN)`; the level can no longer be given here
- The macros `MOD_ASSERT_1P`, `MOD_ASSERT_2P` and other macros that could take one or two parameters, have been removed

- Variations of the `MOD_ASSERT` and others were made that take a level as a separate argument
- Assertions can be compiled out per level, if the compiler optimizes
- The class `ModAssert::AssertLogger` is renamed to `ModAssert::Logger`, and the methods `ModAssert::AddAssertLogger` and `ModAssert::RemoveAssertLogger` are renamed to `ModAssert::AddLogger` and `ModAssert::RemoveLogger`
- The class `ModAssert::StreamLogger` is made, that implements the interface `ModAssert::Logger`, and its constructor has a stream object as its argument, to which output of failing assertions is streamed
- The class `ModAssert::Context` is introduced, that contains the filename and linenumber where an assertion occurs. For compilers that support it, it also has the function name. This class can be derived by the user to contain more information, which can be automatically used by all assertion macros
- The class `ValueList::List` has been renamed `ModAssert::ValueList`, and the class `ModAssert::AssertData` has been removed.
- The class `ModAssert::AskForAction` has been renamed `ModAssert::InformUser`, and the function `ModAssert::SetAskForAction` has been renamed `ModAssert::SetInformUser`.
- The method `ModAssert::RemoveLogger` has been added

New in version 0.2

- The class `ModAssert::AssertHandler` is removed, and the classes `ModAssert::AskForAction` and `ModAssert::Logger` were added to replace it
- Functionality to ignore assertions per case, file and level is implemented
- Adjustments for changes in new version of Rich Booleans

License

ModAssert 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 ModAssert.

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

Q-Mentum provides support for ModAssert, together with Rich Booleans. For more information, contact us at `<sales@q-mentum.com>`.

Requirements

- A C++ compiler that can compile namespaces and templates with template specialization (partial template specialization is not needed)

- STL or wxWidgets
- Rich Booleans [http://sourceforge.net/projects/rich-booleans/] version 1.5 or better

Installation

Make sure the Rich Booleans library is installed properly.

Download the file and decompress its contents to a directory.

Build the library

Below are instructions to build the ModAssert library in the configurations that you need. You certainly need to build the ModAssert library, and possibly also the ModAssertConsole, ModAssertWin32 and/or ModAssertWxGui libraries. The projects whose name contains 'demo' may be of interest to you to see how you can use the ModAssert library. The projects whose name contains 'test' are only provided for experienced users who wish to test ModAssert 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 VC++ 6.0, `*7.1.sln` and `*7.1.vcproj` for VC++ 2003, `*8.0.sln` and `*8.0.vcproj` for VC++ 2005, `*9.0.sln` and `*9.0.vcproj` for VC++ 2008, `*10.0.sln` and `*10.0.vcproj` for VC++ 2010). Converting these files to another version may require adjustments.

Note that these project files are not tested with Visual Studio Express but you probably can just use the same project files.

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.

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 ModAssert 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 ModAssert

To use ModAssert in your application or library, you need to specify the include and library directory. There are two options to do this: one option makes ModAssert available to all your projects, the other option is to specify it only for the projects in which you want to use ModAssert. The first option is preferred, especially if you plan to use ModAssert 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 `modassert` directory and only add its parent directory to the include path.

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

Microsoft Visual C++

Add the directory called `include` in the ModAssert 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 ModAssert 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 `modassert`, 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), 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.
For other compilers, consult the documentation of your compiler.

Adding ModAssert to individual projects

If you chose to not make ModAssert 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. Note that you also should make sure that the Rich Booleans library is available, see the documentation of the Rich Booleans.

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 MODASSERT, that contains the directory where ModAssert is located, to make upgrading easier. Then add `$(MODASSERT)/include` to your include path (Project Settings -> C/C++ -> Preprocessor). Add `$(MODASSERT)/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.

Adding ModAssert to your application

- Link the ModAssert and the Rich Booleans library with your executable. This is not necessary for Visual C++, because ModAssert and RichBool use autolinking with Visual C++, i.e. their headerfiles contain directives that cause the linker to include the correct library for the configuration that you use, if you include `modassert/handler.hpp` in at least one implementation file. In theory this should also work with Borland C++ Builder, but this hasn't been tested.
- Set the assertion handler at initialization time (typically in the `main` method, or a `App` object).
  - If you use the Windows 32 API, you can use the `ModAssert::HandlerWin32` class by calling `ModAssert::SetWin32Handler()` (see the section called “Windows 32”).
  - If you use wxWidgets, you can use the `ModAssert::HandlerWx` class by calling `ModAssert::SetWxHandler()` (see the section called “wxWidgets”).
  - For console applications, you can use the `ModAssert::HandlerConsole` class by calling `ModAssert::SetConsoleHandler()` (see the section called “Console applications”).
  - Otherwise, you can write your own ModAssertHandler (see the section called “Processing assertions (advanced)” on how to do this).
- Include the headerfile `modassert/assert.hpp` in every file where you want to use Modular Assertions.

The example projects in the directory `demos` that are included in the download might help clarify how to do all this.

Breaking into code

If an assertion or check is reported, and a `Responder` object is installed (see the section called “The Responder object”), you get the option to debug, i.e. to break into the code. There is code to do this for different compilers, but this has only been tested with Microsoft Visual C++ 6.0, .NET and Dev-C++. If it doesn't work with your compiler, add the definition `#define MOD_ASSERT_BREAK_HERE ModAssert::ManualBreak()` in the file `include/modassert/assert.hpp` after the other definitions of `MOD_ASSERT_BREAK_HERE` (depending on your compiler, it might already be defined to this), and manually set a breakpoint in the method `ManualBreak()` in the file `src/handler.cpp`.

Note that breaking into the code is only possible if a debugger is attached. Therefore this option is disabled if the symbol `NDEBUG` is defined. If it is not defined, the Win32 ModAssert handler checks if this is the case using the function `IsDebuggerPresent()`. If not, the option to debug is disabled. Other ModAssert handlers don't do this; in that case selecting the debug option may cause a crash.

The purpose of assertion macros

Assertion macros allow a programmer to check conditions of which they know that they *should* be true at a certain point, if all goes well. These errors are also called *unexpected* errors. This can show problems long before they manifest themselves as a bug or a crash. E.g. if a method to find a value in a container returns an

index bigger than the size of the container, this is obviously an error. If that index is stored, and used some time later to index the array, we will notice this error, but we may not know where the index was calculated. Adding an assertion after the calculation of the index like `MOD_ASSERT(idx<vec.size())` would immediately reveal the problem. With a Rich Boolean you could write `MOD_ASSERT(rbLESS(idx, vec.size()))`, so you would also know the value of `idx` and `vec.size()`.

One common usage for assertion macros are contracts. These are pre or post conditions, or invariants (in case of an object). Pre and post conditions should be fulfilled at respectively the entry or exit of a function. Invariants apply to the state of an object, that should be fulfilled at both the exit and entry of every method.

Another example is a method that initializes a pointer, and that should be called only once. The method could then check that the pointer is `NULL` when the method is entered.

It takes some experience to learn to recognize where a programmer can use assertions. A good starting point is to add assertions if you encounter a nasty bug. See what assertions you could add in the affected code, by identifying conditions that should hold, but might not judged on what goes wrong.

The purpose of macros for error handling

ModAssert also provides macros for handling errors (see the section called “MOD_CHECK”), that are not due to bugs, but factors out of control of the application developer, e.g. invalid input of the user, or a network connection that can't be made. These errors are called *expected* errors. Typically, you would return an error code or throw an exception. With the macros for error handling you can do the same, but also have the information logged. These macros are called *checks* in this library. The reasons to use these are entirely different from the reasons to use assertion macros, but their functionality is so similar that ModAssert provides macros for these as well. Actually this package is called ModAssert for historical reasons.

Comparison to other assertion libraries

There are other assertion libraries which have about the same functionality as this one, but are different in two key areas:

- Many other assertion libraries have just one assert macro, that takes a boolean expression, and allow you to add expressions to it using `operator()` and levels using `.level(Error)`. So you have something like

  ```
  assert(foo()<a)(foo())(a)(b).level(Error);
  ```

  However, by design such libraries need to create a local class and do all the work in a member function of that class. The main disadvantage of this is that if you want to break into the code, you break into that function, and have to step out of it with the debugger. Another disadvantage is that selectively compiling out assertions is not possible.

- Many other assertion libraries don't give extra information, e.g. if you want to check if two numbers are equal, you only get a yes/no answer, not the value of the numbers, which would help immensely when you solve bugs in your code. Some assertion libraries have macros like `ASSERT_EQUAL`, that will display the value of the two arguments upon failure, but they don't have much more of them, because if they also want macros like `VERIFY_EQUAL`, they have to duplicate these.

The ModAssert library solves these two problems in the following way:

- ModAssert relies more on macros, so that if you want to break into the code you are directly in the function where the assertion macro is, for most of the macros. Therefore it uses a more traditional

approach, namely specifying the extra data (expressions, level, ...) as extra arguments, and define different macros that accept the different types of arguments. E.g. the above example would be written as

```
MOD_ASSERT_PG(foo() << a << b, ModAssert::Error, foo()<a);
```

There are similar macros `MOD_ASSERT`, `MOD_ASSERT_P` and `MOD_ASSERT_G`, and even more, that you use depending on which arguments you want to add. Another advantage of this approach with macros, is that assertions can be compiled out per level and group (if the compiler optimizes). ModAssert also has other assertions and checks that return a value, but breaking into the debugger will bring you in to another function, of which you have to jump out. So you can choose what is most important for you.

- The condition of an assertion macro in ModAssert can be a boolean expression, but also a Rich Boolean, which gives extra information if the expression fails. The ModAssert assertion macros process this extra information. Using a Rich Boolean, the above example could be rewritten as

  ```
  MOD_ASSERT_PG(b, ModAssert::Error, rbLESS(foo(), a));
  ```

  and the output would include the values of `foo()` and `a` in something like "(12) < (9) nok" (so `foo()` and `a` don't have to be displayed separately). This is just a simple Rich Boolean, there are nearly 60 Rich Booleans, varying from checking for equality to comparing containers of containers with a predicate per element. Every Rich Boolean gives you as much information on the failing condition as possible. And you can write Rich Booleans for your own specific needs.

Some assertion packages only have assertion macros that return a boolean, indicating whether the assertion succeeded. The most assertion macros in ModAssert don't return a value, because it would no longer be possible to break into the debugger where the function is, and you would have to jump out of the function every time, which becomes tiresome if you have to do it a lot. ModAssert has the macro `MOD_CHECK` (and variations) that have a failure action, to replace `if (!ASSERT(...))`. This makes it clear that this is a check for an error that is not due to a bug, and that the application can recover from the error by executing the failure action. Failing assertions on the other hand should ideally result in termination of the application, and the rest of your code should assume that the condition succeeded.

However, since version 2.0 ModAssert has macros that return a boolean, or even a value if you use a Rich Boolean. This is mainly to allow code like `MOD_VERIFY_V(rbDIFF(wnd, NULL))->Display();`, but you can also use it to recover from bugs, like `if (!MOD_VERIFY_B(wnd)) {...}`, if your application has to be very safe. It is best to still use `MOD_CHECK` for expected errors, and use the new macros for unexpected errors.

The available assertion macros

Statement assertion and check macros

The assertions macros in this section are statements, so you can't assign them to a left value.

MOD_ASSERT

The simplest assertion macro is `MOD_ASSERT`. It has one argument, a boolean expression or a Rich Boolean. If its argument evaluates to false, it will inform the user about this failure if there is an active responder, and log it if loggers were added.

```
MOD_ASSERT(n==0);
```

Its argument could also be a Rich Boolean:

```
MOD_ASSERT(rbEQUAL(n, 0));
```

This is equivalent to the first `MOD_ASSERT`, but will also display the value of `n` when the condition fails.

MOD_VERIFY

`MOD_VERIFY` is similar `MOD_ASSERT`, the only difference is that `MOD_VERIFY` will evaluate its condition when reporting of assertions is disabled, but if the condition fails it will not be displayed or logged. So unlike `MOD_ASSERT`, `MOD_VERIFY` is never compiled out completely, but its code size is still reduced if reporting of assertions is disabled. When reporting of assertions is enabled, it is equivalent to `MOD_ASSERT`. This can be important if you have to call a function, and check whether its return value is as expected, but don't need the return value later:

```
// foo() is still evaluated if reporting of assertions is disabled
MOD_VERIFY(foo()<10);
```

Note that well written Rich Booleans still evaluate their arguments in a `MOD_VERIFY` macro (and variations) when reporting of assertions is disabled, but without creating a `RichBool::Analysis` object. Even the condition is not tested, to save processing time.

```
// foo() is still evaluated if assertions are not reported,
// but the condition foo()<10 is not evaluated
MOD_VERIFY(rbLESS(foo(), 10));
```

MOD_FAIL

The macro `MOD_FAIL` doesn't have a condition argument, and always fails (so it is not modular, because it doesn't have a condition, but it has the prefix `MOD_` because it is part of the ModAssert package). This is useful if a certain part of your code shouldn't be reachable.

Example:

```
for (int i=0; i<n; ++i)
{
    if (a[i]>10)
        return a[i];
}
// at least one should be bigger than 10
MOD_FAIL;
```

MOD_CHECK

`MOD_CHECK` is similar to `MOD_VERIFY`, except that `MOD_CHECK` has a *failure action* as the last argument after the condition, which will be evaluated if the condition fails. `MOD_CHECK` is called a *check*, not an assertion, because it is not meant to test for bugs, but rather for other errors, such as invalid input by the user. Normally execution of the application will continue after the failure action is executed.

When reporting of checks is disabled, it will still evaluate its argument, test the condition (without creating an analysis if the condition is a Rich Boolean), and evaluate its failure action if the condition fails, but it

will not be displayed or logged (therefore `MOD_CHECK` can be thought of as `MOD_VERIFY` with a failure action). So `MOD_CHECK` is never compiled out completely, but its code size is still reduced if reporting of checks is disabled.

The failure action could e.g. throw an exception, return with an error value, break from a loop, or continue a loop to skip the rest of the loops body. Everywhere you have error handling in your code, you might consider using a `MOD_CHECK` macro (or a variation).

```
MOD_CHECK(p!=NULL, throw IllegalArgumentException());
```

Note that well written Rich Booleans still evaluate their arguments and condition in this case when reporting of checks is disabled, but without creating a `RichBool::Analysis` object.

```
// the condition foo()<10 is still evaluated
//if reporting of checks is disabled
MOD_CHECK(rbLESS(foo(), 10), throw IllegalArgumentException());
```

Note: responders and loggers are aware whether a failure comes from an assertion or a check, and might treat them differently. This is because these macros are meant for handling errors that are beyond the control of the programmer, whereas assertion macros are meant for bugs. See also the section called “Bugs versus other errors”

MOD_CHECK_FAIL

The macro `MOD_CHECK_FAIL(action)` is equivalent to `MOD_CHECK(false, action)`. It is useful when a part of your code should be unreachable under normal circumstances, and it is an error and not a bug if that part is reached.

Expression assertion and check macros

The assertions macros in this section are expressions, so you can assign them to a left value.

If you want to use a Rich Boolean with the macros in this section, you have to use one of a different kind, the ones that start with `rbv` instead of `rb`. The value that is returned by the `MOD_VERIFY_V` and `MOD_CHECK_V` macro is one of the arguments of the Rich Boolean macro, usually the first one. These Rich Booleans still evaluate their arguments when reporting of assertions is disabled, but without creating a `RichBool::Analysis` object. Even the condition is not tested, to save processing time.

MOD_VERIFY_V

`MOD_VERIFY_V` is the same as `MOD_VERIFY`, but it returns the value of the condition. Often the condition is a pointer returned by a call, that shouldn't be NULL.

```
// GetWidget() is still evaluated if reporting of assertions is disabled
// This macro returns a value
Widget *w = MOD_VERIFY_V(GetWidget());
```

Using a Rich Boolean you can perform tests on the returned value for a lot more than just not being NULL.

```
// Here MOD_VERIFY_V will return the value of foo()
// (most Rich Booleans return their first argument).
// foo() is still evaluated if assertions are not reported,
// but the condition foo()<10 is then not evaluated.
int f = MOD_VERIFY_V(rbvLESS(foo(), 10));
```

MOD_VERIFY_B

`MOD_VERIFY_B` is the same as `MOD_VERIFY_V`, except that it returns a `ModAssert::UseBool` object. See the section called “UseBool objects”.

```
if (!MOD_VERIFY_B(rbvLESS(foo(), 10)))
{
  ...
};
```

Note that you most likely won't need this, because if this part of your application works well the condition will always succeed. Only if your application should always make sure that it corrects errors, even errors that are a consequence of bugs, you should use this.

MOD_CHECK_V

`MOD_CHECK_V` is the same as `MOD_CHECK`, but it returns the condition, or, if you use a Rich Boolean, one of the arguments of the Rich Boolean. The failure action has to be an object on which `operator()()` is defined (the return value doesn't matter, and it can be `const` or not).

```
struct ThrowIllegalArgumentException
{
  void operator()() const
  {
    throw IllegalArgumentException();
  }
} throwIllegalArgumentException;

int *p = MOD_CHECK_V(foo(), throwIllegalArgumentException);
```

MOD_CHECK_B

`MOD_CHECK_B` is the same as `MOD_CHECK_V`, except that it returns a `ModAssert::UseBool` object, and doesn't have a failure action. See the section called “UseBool objects”.

```
// Ok.
if (!MOD_CHECK_B(rbvLESS(a, 10)))
{
  DecreaseValues();
  throw MyException();
}
```

```
// Not ok.
MOD_CHECK_B(rbvLESS(a, 10));


// Ok if caller checks the value.
ModAssert::UseBool foo()
{
  ...
  return MOD_CHECK_B(rbvLESS(a, 10));
}
```

Note that in the last example the function should not return bool, because then the `ModAssert::UseBool` will assume that the value is checked, but the caller of the function might ignore the return value.

```
// Not ok, but no assertion will fail
bool foo()
{
  ...
  return MOD_CHECK_B(rbvLESS(a, 10));
}
```

UseBool objects

`MOD_VERIFY_B` and `MOD_CHECK_B` return a `ModAssert::UseBool` object. Objects of this class can be converted to a boolean, so you can assign them to a boolean or use them as the condition in e.g. an if-statement or while-statement.

You should always use a `ModAssert::UseBool` by converting it to a boolean. If you don't, its destructor will cause an assertion to fail, to remind you that you ignored it. Note that this is not fool proof, one could assign it to a boolean but not use the boolean.

You can turn checking of the usage of `ModAssert::UseBool` objects on or off by calling `ModAssert::UseBool::SetCheck(bool b)`. If `NDEBUG` is defined, this is off by default, otherwise it is on. The checking is also affected by the setup functions.

`ModAssert::UseBool` objects have transfer semantics. This implies that if you assign a `ModAssert::UseBool` to another `ModAssert::UseBool` object, the original one will not check if it is used, but the new one still will. This means that you can use it as the return value of a function, if you want to leave the checking of the condition to the caller.

Note: if a Rich Boolean fails, it may be because the condition failed or because a bad value was given as a consequence of dereferencing an invalid pointer (see the documentation of Rich Booleans). If it happens because of a bad value, then:

- it will have a level Fatal, and no group, even if you used a group and or a level in the assertion or check
- in a `MOD_CHECK`, ModAssert will treat it as a failing assertion, because this is not an expected error

Adding expressions and messages

Adding the suffix `_P` (or `P` if it already has a suffix `_V`) allows to add expressions before the condition, e.g. `MOD_ASSERT_P`, `MOD_VERIFY_VP`. If the condition fails, these expressions will be evaluated, and shown and logged together with a failure message. This gives a programmer a better understanding of what is going on. They are a little different for different types of assertions and checks.

- For statement macros the first argument is a number of expressions separated by <<, and only if the assertion is reported, these will be evaluated and shown with their expressions. Here the number of expressions is not limited.
- For expression macros the first argument is a comma separated list of expressions embraced by parentheses, that are always evaluated if reporting of the macro is enabled, but only converted to a string if it is reported. Here the maximum number of expressions is 8. Note that they have to be enclosed by parentheses even if there is only one expression.

For both types it is recommended to use only expressions that don't have side effects.

```
int b = foo(a);
MOD_ASSERT_P(a, rbLESS(b, 10));
...
int n = nrSpaces+nrTabs;
MOD_ASSERT_P(nrSpaces << nrTabs, rbLESS_OR_EQUAL(n, str.size()));

int len = str.size();
int p = MOD_VERIFY_VP((nrSpaces, nrTabs), rbvLESS_OR_EQUAL(n, len));
```

Example output:

```
`b':<12> < `10':<10> nok
a: 17
...
`n':<8> <= `str.size()':<7> nok
nrSpaces: 5
nrTabs: 3
```

This macro also accepts literal strings, which will be shown as messages instead of as expressions.

```
int b = foo(a);
MOD_ASSERT_P(a, rbLESS(b, 10));
...
int n = nrSpaces+nrTabs;
MOD_ASSERT_P(
  "the number of spaces or the number of tabs was miscalculated"
      << nrSpaces << nrTabs,
  rbLESS_OR_EQUAL(n, str.size()));
```

Example output:

```
`n':<8> <= `str.size()':<7> nok
message: the number of spaces or the number of tabs was miscalculated
nrSpaces: 5
nrTabs: 3
```

Note that there is no use in adding the expressions in the rich boolean (`n` and `str.size()` in the example) to the parameters, since these will already be shown.

Adding a group

You can also specify a group in an assertion or a check. To do this, create an object of the class `ModAssert::Group<ModAssert::ReportFailure>`, add a suffix `G` to the assertion macro (after P if you add parameters), and give the object as argument before the condition, and after the parameters (if you add parameters).

```
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup("group 1");
...
MOD_ASSERT_G(modAssertGroup, rbLESS(n, str.size()));
```

`ModAssert::Group<ModAssert::ReportFailure>` has a constructor that takes a `const char*` as its argument, to give it a name. If such an assertion fails, its name will be shown and you'll have the option to stop displaying all assertions that are in this group.

If you want to disable reporting of assertions in that group at compile time, you can do so by changing the groups type to `ModAssert::Group<ModAssert::ReportNone>`:

```
ModAssert::Group<ModAssert::ReportNone> modAssertGroup("group 1");
...
MOD_ASSERT_G(modAssertGroup, rbLESS(n, str.size()));
```

This will reduce the executable size (if the compiler optimizes).

Usually group objects are declared inside a class, as a static variable or an instance variable (depending on whether you want to control assertions for a whole class at once or per object). If they are an instance variable, you should make them `mutable`, because assertions and checks need non-const groups. ModAssert could easily apply `const_cast`, but it is better to use `mutable` because this indicates that the group is not a part of the state of the object.

Adding a level

Another type of groups are level groups, which have an integer valued level. You can add a level group for the assertion, to indicate how serious the assertion is; you can choose from `Info`, `Warning`, `Error` or `Fatal` in the `ModAssert` namespace. These are objects of the class `ModAssert::Level<...>`.

Note: this has nothing to do with the levels in Rich Booleans.

The assertion in the following example has a level group:

```
MOD_ASSERT_G(ModAssert::Fatal, rbLESS(n, str.size()));
```

The assertion in the following example has an expression and a level group:

```
MOD_ASSERT_PG(a, ModAssert::Fatal, rbLESS(n, str.size()));
```

The advantage of adding a level, is that reporting of assertions can be controlled per level group at runtime, and even at compile time, which can reduce the executable size (see the section called “Disabling the reporting of Modular Assertions per level”).

Note: `ModAssert::Error` is implicitly added if a group is used that is not a level, or no group at all is used. This means that if displaying an assertion is turned off for that level, assertions with a levelless group will also not be displayed (just like assertions without a group or level). See also the section called “Disabling the reporting of Modular Assertions per level”.

Note: the levels remember whether to log or display a failure separately for assertions and checks. So e.g. if an assertion or check is reported, and you choose to stop displaying assertions of that level, checks with that level will still be displayed.

Combining groups

Groups that are not level groups can be combined with `operator&&`. An assertion will then only be displayed when they both have to be displayed. Likewise they will only be logged if they both have to be logged. Level groups can be added to a group that is not a level group, or a combination of such groups, with `operator%`, where the level group should be the second operand. If a level group is combined with another group, the combination is considered to have the same level as the level group. Groups of any kind cannot be added to a combination of a group and a level group (so only one level group per combination is allowed).

```
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup1("group 1"),
  modAssertGroup2("group 2");
...
MOD_ASSERT_G(modAssertGroup1 && modAssertGroup2, rbLESS(n, str.size()));
...
MOD_ASSERT_G(modAssertGroup1 % ModAssert::Warning, rbLESS(n, str.size()));
...
MOD_ASSERT_G((modAssertGroup1 && modAssertGroup2)% ModAssert::Warning,
  rbLESS(n, str.size()));
```

This also allows to disable reporting of assertions by turning off one of them, which can still reduce the executable size:

```
ModAssert::Group<ModAssert::ReportNone> modAssertGroup1("group 1");
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup2("group 2");
...
// this will be compiled out:
MOD_ASSERT_G(modAssertGroup1 && modAssertGroup2, rbLESS(n, str.size()));
...
// this will be compiled out:
MOD_ASSERT_G(modAssertGroup1 % ModAssert::Warning, rbLESS(n, str.size()));
```

The following will not compile:

```
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup1("group 1"),
  modAssertGroup2("group 2");
...
// error: use operator% to add a level group
MOD_ASSERT_G(modAssertGroup1 && ModAssert::Warning, rbLESS(n, str.size()));
...
// error: only one level group allowed per combination
```

```
MOD_ASSERT_G((modAssertGroup1 % ModAssert::Error) % ModAssert::Warning,
  rbLESS(n, str.size()));
...
// error: can't add group to a combination of a group and a level group
MOD_ASSERT_G((modAssertGroup1 % ModAssert::Warning) && modAssertGroup2,
  rbLESS(n, str.size()));
...
// error: level group should be the second operand
MOD_ASSERT_G((ModAssert::Warning % modAssertGroup1, rbLESS(n, str.size()));
```

You can also combine groups with `operator||`. In this case, a failed assertion will be reported when at least one should be reported. They will not be reported if they both aren't.

Note: if no level group is in the combination, `ModAssert::Error` is implicitly added. This means that if displaying an assertion is turned off for that level group, assertions with a group also will not be displayed (just like assertions without a group or level). See also the section called “Disabling the reporting of Modular Assertions per level”.

Displaying and logging an assertion if it succeeds

In certain cases you want to see the information of an assertion, even if the condition succeeds. The analysis and the expressions can be handy when debugging an application. You can do so by changing the groups type to `ModAssert::Group<ModAssert::ReportAll>`. This will cause all assertions with this group, to report the assertion, even if the condition succeeds. You can also do this for individual assertions by using `ModAssert::IfSuccess` as the group of an assertion.

```
ModAssert::Group<ModAssert::ReportAll> modAssertGroup("group 1");
...
MOD_ASSERT_G(modAssertGroup, rbLESS(n, str.size()));
MOD_ASSERT_G(ModAssert::IfSuccess, rbLESS(n, str.size()));
```

This poses a problem if you want to combine such a group object with another group object by using `operator&&`, because the other object returns `false` if the condition succeeds, so in that case there is no use in using them.

```
ModAssert::Group<ModAssert::ReportAll> modAssertGroup1("group 1");
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup2("group 2");
...
// next two will NOT be displayed or logged if the condition succeeds
MOD_ASSERT_G(modAssertGroup1 && modAssertGroup2, rbLESS(n, str.size()));
MOD_ASSERT_G(ModAssert::IfSuccess && modAssertGroup2, rbLESS(n, str.size()));
```

To solve this problem, group objects of the types `ModAssert::Group<ModAssert::ReportNone>`, `ModAssert::Group<ModAssert::ReportFailure>` and `ModAssert::Group<ModAssert::ReportAll>`, as well as the object `ModAssert::IfSuccess` have a template `operator()`, which takes one argument, that should be a group object. If the condition of the assertion succeeds, the first object is asked what to do when a condition succeeds, and the second (the argument) is asked what to do if a condition *fails* (for both displaying and logging, separately).

```
ModAssert::Group<ModAssert::ReportAll> modAssertGroup1("group 1");
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup2("group 2");
...
// next two will be displayed and logged if the condition succeeds
MOD_ASSERT_G(modAssertGroup1(modAssertGroup2), rbLESS(n, str.size()));
MOD_ASSERT_G(ModAssert::IfSuccess(modAssertGroup2), rbLESS(n, str.size()));
```

However, don't use them the other way around:

```
ModAssert::Group<ModAssert::ReportAll> modAssertGroup1("group 1");
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup2("group 2");
  ...
// next two will not be displayed or logged:
MOD_ASSERT_G(modAssertGroup2(modAssertGroup1), rbLESS(n, str.size()));
MOD_ASSERT_G(modAssertGroup2(ModAssert::IfSuccess), rbLESS(n, str.size()));
```

A more complicated, less legible, example:

```
ModAssert::Group<ModAssert::ReportAll> group1("group 1"),
  group2("group 2");
ModAssert::Group<ModAssert::ReportFailure> group3("group 3"),
  group4("group 4");
...
// next one will be displayed and logged if the condition succeeds
MOD_ASSERT_G((group1 && group2)(group3 || group4),
      rbLESS(n, str.size()));
```

Note that using `operator()` has no use for objects of the type `ModAssert::Group<ModAssert::ReportNone>` or `ModAssert::Group<ModAssert::ReportFailure>`, because then it is equivalent to using `operator&&`, but they can use this mechanism anyway, so one can easily switch between the three types.

Also note that this problem doesn't exist with `operator||`.

Optional actions

Sometimes it is desirable to be able to perform a certain action when an assertion or check is reported, especially when an assertion or check often fails, and we want to stop the execution of that part of the code. One might want to throw an exception, return false or break from a loop. This is possible with macros similar to the ones already discussed, with a suffix 'O', a capital letter o. (`MOD_ASSERT_O`, `MOD_ASSERT_PO`, `MOD_VERIFY_O`, `MOD_CHECK_O`, `MOD_FAIL_O`, ...). The suffix O should come after the suffixes P and/or G if you use these. These macros have two extra arguments before the condition (and after the parameters and group, if you use these): the optional action, and a description of the optional action. For expression assertion and checks, the optional action has to be an object on which `operator()()` is defined (the return value doesn't matter, and it can be `const` or not); if the user chooses to perform the optional action, `operator()()` is called. For the statement assertions and checks, it can be any expression, that is only performed it the user chooses to. Example:

```
MOD_ASSERT_O(throw std::invalid_argument(), "throw", p!=NULL);
```

```
...

void throw_invalid_argument()
{
  throw invalid_argument();
}

...

int *p = MOD_VERIFY_VO(throw_invalid_argument, "throw", foo());
```

Performance of expression assertions and checks

You should note that some variations of `MOD_VERIFY_V`, `MOD_CHECK_V` and `MOD_CHECK_B` are less performant than the ones that don't return a value or boolean.

- If they add expressions, the expressions are evaluated and a temporary object is created to hold references to the values of these expressions, regardless of whether the condition succeeds, but only if reporting is enabled at compile time (separately for assertions and checks). If there is only one expression, this causes no overhead. For two or more expressions, this causes a little overhead.
- If you use a Rich Boolean for the condition, a temporary object is created to hold a reference to every argument of the Rich Boolean that is not returned, regardless of whether reporting is enabled at compile time or not. Most Rich Booleans only have two arguments so the temporary object has a reference to just one argument, which most likely is easy to optimize by compilers.

Default parameters

Defining the macros `MOD_ASSERT_DEFAULT_PARAMETERS` and `MOD_ASSERT_DEFAULT_PARAMETERS_V` before including `modassert/assert.hpp` allows you to define default parameters for all assertion and check macros. You should define it to an expression that contains the parameters separated by `<<` for `MOD_ASSERT_DEFAULT_PARAMETERS` and by a comma with the whole enclosed by parentheses for `MOD_ASSERT_DEFAULT_PARAMETERS_V` (i.e. in the same way as you add parameters to an assertion or check macro). If an assertion or check macro has its own parameters to be shown, they will be added before the default ones. The maximum number of expressions you can give in `MOD_ASSERT_DEFAULT_PARAMETERS_V` is 8; so the maximum number of expressions that an assertion that returns a value can have, is 16: 8 default ones and 8 explicit ones. Note that you could use different expressions for `MOD_ASSERT_DEFAULT_PARAMETERS` and `MOD_ASSERT_DEFAULT_PARAMETERS_V`, but this is not recommended. It is even recommended to use both even if you need only one, because you may use assertions or checks of the other kind later, and forget to define its default parameters.

Therefore the listing

```
#include "modassert/assert.hpp"

void foo()
{
 MOD_ASSERT_P(a, rbEQUAL(b, c));
 ...
 int n = MOD_VERIFY_VP((a), rbEQUAL(b, c));
```

```
 ...
 MOD_CHECK_P(a, rbLESS(b, c), return);
 ...
 MOD_ASSERT(b << a, rbEQUAL(c, 10));
}
```

is equivalent to this listing:

```
#define MOD_ASSERT_DEFAULT_PARAMETERS a
#define MOD_ASSERT_DEFAULT_PARAMETERS_V (a)
#include "modassert/assert.hpp"

void foo()
{
 MOD_ASSERT(rbEQUAL(b, c));
 ...
 int n = MOD_VERIFY_V(rbEQUAL(b, c));
 ...
 MOD_CHECK(rbLESS(b, c), return);
 ...
 MOD_ASSERT_P(b, rbEQUAL(c, 10));
}
```

You can ignore the default parameters by inserting `NP` where you usually insert `P`. If you want to ignore the default parameters and use some specific parameters, insert `SP` instead.

```
#define MOD_ASSERT_DEFAULT_PARAMETERS a
#define MOD_ASSERT_DEFAULT_PARAMETERS_V (a)
#include "modassert/assert.hpp"

void foo()
{
 // shows parameter a:
 MOD_ASSERT(rbEQUAL(b, c));
 ...
 // shows parameters a and n:
 MOD_ASSERT_P(n, rbEQUAL(b, c));
 ...
 // shows no parameters:
 MOD_ASSERT_NP(rbEQUAL(b, c));
 ...
 // shows only parameter n:
 MOD_ASSERT_SP(n, rbEQUAL(b, c));
}
```

If `MOD_ASSERT_DEFAULT_PARAMETERS` and `MOD_ASSERT_DEFAULT_PARAMETERS_V` are not defined, macros with `NP` are equivalent to macros without it, and macros with `SP` are equivalent to macros with `P`. So defining and undefining `MOD_ASSERT_DEFAULT_PARAMETERS` and `MOD_ASSERT_DEFAULT_PARAMETERS_V` makes no difference to these macros.

Default groups

You can define a default group per source code file, for assertions and checks. For assertions, define the macro `MOD_ASSERT_DEFAULT_GROUP` before including `modassert/assert.hpp` to the group. For checks, do the same with the macro `MOD_CHECK_DEFAULT_GROUP`. They can be anything that can be given as a group to an assertion or check. Assertions and checks that specify a group ignore this.

```
// define these before including modassert/assert.hpp:
#define MOD_ASSERT_DEFAULT_GROUP group1
#define MOD_CHECK_DEFAULT_GROUP group2

#include "modassert/assert.hpp"

// these have to be declared after including modassert/assert.hpp:
ModAssert::Group<ModAssert::ReportFailure> group1("group1"),
    group2("group2"), group3("group3");
...
// this will have the group group1:
MOD_ASSERT(rbMORE(a, 0));

// this will have the group group2:
MOD_CHECK(rbMORE(a, 0), return false);

// this will have the group group3:
MOD_ASSERT_G(group3, rbMORE(a, 0));
```

Default optional action

You can define a default optional action per source code file, by defining the symbol `MOD_ASSERT_DEFAULT_OPTIONAL_ACTION` and/or `MOD_ASSERT_DEFAULT_OPTIONAL_ACTION_V` with the optional action before including `modassert/assert.hpp`. If you do this, you should also define `MOD_ASSERT_DEFAULT_OPTIONAL_ACTION_TEXT` to a literal string that will be used by the responder if it asks an action from the user. `MOD_ASSERT_DEFAULT_OPTIONAL_ACTION` is used for assertions and checks that don't return a value, and can be any expression, that will be evaluated if the user chooses to. `MOD_ASSERT_DEFAULT_OPTIONAL_ACTION_V` is used for assertions and checks that return a value or a boolean, and should be an object on which `operator()()` is defined, which will be called if the user chooses to. All assertions and checks without the suffix `O` will then use this optional action.

If you want another optional action for a particular assertion or check, just add the suffix `O` to it, and add the optional action and its description as usual.

Setting a global optional action

The function `ModAssert::SetGlobalOptionalAction` allows you to make a function available as an optional action in responders, in addition to the optional actions that can be given in individual assertion and check macros. This function takes two arguments. The first argument is the address of the function, which should take no arguments and return `void`. The second argument is a short description of the action, which should be a `const char*` (usually a string literal). This description will be shown by responders. You can get the global optional by calling `GetGlobalOptionalAction` and get the description by calling `GetGlobalOptionalActionDescription`.

Note that this has restrictions when compared to the optional actions that you give directly in statement assertion and check macros, because it is impossible to return with a value, or break or continue a loop.

However, the advantage of the global optional action is that it can be set and modified at runtime, and you only have to do it once.

What is allowed for the actions

The macro `MOD_CHECK` and its variations have failure actions, and the macros with a suffix `O` have optional actions (macros like `MOD_CHECK_O` even have both). For expression assertions and checks, this is rather limited. For statement assertions and checks, there are more possibilities. Because these actions are the argument of a macro, they have one limitation as a consequence of the C/C++ parsing rules: they can't have commas inside of them, unless the comma is between parentheses. This limitation can be circumvented by first defining a macro that contains the action with the comma, and use that macro as the action:

```
#define ACTION ++a, ++b
MOD_ASSERT_O(ACTION, "++a, ++b", rbMORE(a, 0));
```

However, in this example it would be better to use semicolons:

```
MOD_ASSERT_O(++a; ++b;, "++a, ++b", rbMORE(a, 0));
```

(the semicolon after `++b` actually isn't necessary) or add parentheses:

```
MOD_ASSERT_O( (++a, ++b), "++a, ++b", rbMORE(a, 0));
```

Note that all code that can be in a scope, is allowed for these actions, as long as it doesn't contain commas outside of an expression. So this is possible:

```
MOD_ASSERT_O(
    if (b)
    {
        int n=0;
        foo(n, 1); // fine, comma is between parentheses
    },
    "call foo",
    rbMORE(a, 0)
);
```

but the following is not possible:

```
MOD_ASSERT_O(
    if (b)
    {
        int n=0, p=1; // error: the compiler sees an extra macro argument
        foo(n, p);    // fine, comma is between parentheses
    },
    "call foo",
    rbMORE(a, 0)
);
```

Overview

Table 1. Overview of available assertion macros

MOD_VERIFY_BNPO	N	N	Y	N	Y	UseBool	Y	N
MOD_VERIFY_BSPO	Y	N	Y	N	Y	UseBool	Y	N
MOD_VERIFY_BNPGO	N	Y	Y	N	Y	UseBool	Y	N
MOD_VERIFY_BSPGO	Y	Y	Y	N	Y	UseBool	Y	N
MOD_CHECK_NP	N	N	N	Y	Y	void	Y	N
MOD_CHECK_SP	Y	N	N	Y	Y	void	Y	N
MOD_CHECK_NPG	N	Y	N	Y	Y	void	Y	N
MOD_CHECK_SPG	Y	Y	N	Y	Y	void	Y	N
MOD_CHECK_NPO	N	N	Y	Y	Y	void	Y	N
MOD_CHECK_SPO	Y	N	Y	Y	Y	void	Y	N
MOD_CHECK_NPGO	N	Y	Y	Y	Y	void	Y	N
MOD_CHECK_SPGO	Y	Y	Y	Y	Y	void	Y	N
MOD_CHECK_VNP	N	N	N	Y	Y	value	Y	N
MOD_CHECK_VSP	Y	N	N	Y	Y	value	Y	N
MOD_CHECK_VNPG	N	Y	N	Y	Y	value	Y	N
MOD_CHECK_VSPG	Y	Y	N	Y	Y	value	Y	N
MOD_CHECK_VNPO	N	N	Y	Y	Y	value	Y	N
MOD_CHECK_VSPO	Y	N	Y	Y	Y	value	Y	N
MOD_CHECK_VNPGO	N	Y	Y	Y	Y	value	Y	N
MOD_CHECK_VSPGO	Y	Y	Y	Y	Y	value	Y	N
MOD_CHECK_BNP	N	N	N	Y	Y	UseBool	Y	N
MOD_CHECK_BSP	Y	N	N	Y	Y	UseBool	Y	N
MOD_CHECK_BNPG	N	Y	N	Y	Y	UseBool	Y	N
MOD_CHECK_BSPG	Y	Y	N	Y	Y	UseBool	Y	N
MOD_CHECK_BNPO	N	N	Y	Y	Y	UseBool	Y	N
MOD_CHECK_BSPO	Y	N	Y	Y	Y	UseBool	Y	N
MOD_CHECK_BNPGO	N	Y	Y	Y	Y	UseBool	Y	N
MOD_CHECK_BSPGO	Y	Y	Y	Y	Y	UseBool	Y	N
MOD_FAIL_NP	N	N	N	N	N	void	-	N
MOD_FAIL_SP	Y	N	N	N	N	void	-	N
MOD_FAIL_NPG	N	Y	N	N	N	void	-	N
MOD_FAIL_SPG	Y	Y	N	N	N	void	-	N
MOD_FAIL_NPO	N	N	Y	N	N	void	-	N
MOD_FAIL_SPO	Y	N	Y	N	N	void	-	N
MOD_FAIL_NPGO	N	Y	Y	N	N	void	-	N
MOD_FAIL_SPGO	Y	Y	Y	N	N	void	-	N
MOD_CHECK_FAIL_NP	N	N	N	Y	N	void	-	N
MOD_CHECK_FAIL_SP	Y	N	N	Y	N	void	-	N
MOD_CHECK_FAIL_NPG	N	Y	N	Y	N	void	-	N
MOD_CHECK_FAIL_SPG	Y	Y	N	Y	N	void	-	N
MOD_CHECK_FAIL_NPO	N	N	Y	Y	N	void	-	N
MOD_CHECK_FAIL_SPO	Y	N	Y	Y	N	void	-	N
MOD_CHECK_FAIL_NPGO	N	Y	Y	Y	N	void	-	N
MOD_CHECK_FAIL_SPGO	Y	Y	Y	Y	N	void	-	N

Note that the arguments appear in the same order as the suffixes that indicate that the argument is present, and that this order is always the same. I.e. first come the parameters, then the groups, and then the optional action (for which there are two arguments). The condition is always the last argument, except for `MOD_CHECK` and its variations, where the failure action comes after the condition. Expression assertions and checks have a suffix V just before the other suffixes. Checks that return a boolean have a suffix B just before the other suffixes.

Bugs versus other errors

It is important to understand the following design decision in ModAssert: the assertion macros that begin with `MOD_ASSERT`, `MOD_VERIFY` and `MOD_FAIL` are meant for bugs in your application, while the macros that begin with `MOD_CHECK` are meant for other errors, i.e. errors that are not under the control of the programmers. Loggers and responders are aware of this distinction, and can act differently depending on this. E.g. to be safe, an application should shut down if a bug is encountered, otherwise worse things could happen (although ModAssert doesn't enforce this), unless it is a critical application that should recover from bugs. On the other hand, if an error occurs due to other causes, the failure action of the `MOD_CHECK` macro should be executed. The application developer decides what that failure action is. Typically it is throwing an exception, that is caught elsewhere, so execution of the application continues, but it can also be aborting the application.

Actually, the distinction also depends on what you want to happen. E.g. if your application detects that there is no more memory available, it might be wise to shut down the application, so you could use e.g. `MOD_ASSERT(p!=NULL)`, although `MOD_CHECK(p!=NULL, terminate())` is better.

Controlling displaying and logging of assertions and checks at runtime

Many functions in this section have a const reference to a `ModAssert::CategoryBase` object as an argument. This specifies whether you specify it for assertions or checks. `ModAssert::CategoryBase` is an abstract class. In this version the derived concrete classes `ModAssert::Category<ModAssert::Assertions>` and `ModAssert::Category<ModAssert::Checks>` are available. Use the former class for assertions, the latter for checks. `categoryAsserts` and `categoryChecks` are respectively objects of these classes (defined in `modassert/assert.hpp`, in the namespace ModAssert).

All

The methods `void SetDisplayAll(bool b, const ModAssert::CategoryBase &category)` and `void SetLogAll(bool b, const ModAssert::CategoryBase &category)` let you control displaying and logging of all assertions or checks at once.

Per file

The methods `void SetDisplayInFile(const char *file, bool b, const ModAssert::CategoryBase &category)` and `void SetLogInFile(const char *file, bool b, const ModAssert::CategoryBase &category)` let you control displaying and logging of assertions and checks per source code file (best used with __FILE__). The methods `bool GetDisplayInFile(const char *file, const`

`ModAssert::CategoryBase &category)` and `bool GetLogInFile(const char *file, const ModAssert::CategoryBase &category)` allow you to check whether assertions in a source code file will be displayed or logged.

Per level

`ModAssert::Info`, `ModAssert::Warning`, `ModAssert::Error` and `ModAssert::Fatal` are of the type `ModAssert::Level<...>`. They have the methods `void SetDisplay(bool b, const ModAssert::CategoryBase &category)` and `void SetLog(bool b, const ModAssert::CategoryBase &category)`, to control displaying and logging per level, and the methods `bool GetDisplay(bool success, const ModAssert::CategoryBase &category)` and `bool GetLog(bool success, const ModAssert::CategoryBase &category)`, to check if they are displayed or logged. Note that levels handle this separately for assertions and checks.

Per group

The template class `ModAssert::Group<T>` has the methods `void SetDisplay(bool b)` and `void SetLog(bool b)`, to control displaying and logging per group. The methods `bool GetDisplay(bool success)` and `bool GetLog(bool success)` allow you to check whether assertions with that group will be displayed or logged; the argument specifies whether it is an assertion that failed or that succeeded. These methods call the same methods on the object of type T that they contain.

Objects of the type `ModAssert::Group<ModAssert::ReportAll>` don't have the methods `void SetDisplay(bool b)` and `void SetLog(bool b)`, because with these objects you can specify this behaviour separately for failing and succeeding assertions. Therefore they have the methods `void SetDisplay(size_t n, bool b)` and `void SetLog(size_t n, bool b)`, where `n` is 0 for failing assertions, 1 for succeeding assertions. Note however that it is not possible to display or log only succeeding assertions. So if you call `SetDisplay(0, false)`, succeeding assertions will also not be displayed. If you call `SetDisplay(0, true)` after that, only failing assertions will be displayed. Call `SetDisplay(1, true)` to display succeeding assertions again. The same applies for `void SetLog(size_t n, bool b)`.

Objects of the type `ModAssert::Group<ModAssert::ReportNone>` and `ModAssert::Group<ModAssert::ReportFailure>` have besides the methods with one argument also the overloaded methods with two arguments, although the first argument is then ignored. They have these methods to allow easy switching between these types.

Note that group objects don't distinguish between assertions and checks, but you can make separate objects for each if necessary.

`ModAssert::Group<T>::SetDisplay`, `SetDisplayInFile` and `SetDisplayAll` are called when an assertion failed, and the user decides to stop displaying all assertions or checks, or just assertions or checks of a file or a group, but you can also call these functions yourself, e.g. when your application starts up. `ModAssert::Group<T>::SetLog`, `SetLogInFile` and `SetLogAll` are not called by ModAssert itself, you have to call them yourself.

Enabling and disabling the reporting of Modular Assertions at compile time

If reporting of assertions/checks is disabled, this means that assertion macros beginning with `MOD_ASSERT` and `MOD_FAIL` will be equivalent to an empty statement, while assertion macros beginning with `MOD_VERIFY` and `MOD_CHECK` evaluate their arguments, but don't display or log anything if their condition is false. Moreover macros that start with `MOD_CHECK` still execute their failure action if the condition failed. So `MOD_VERIFY` and `MOD_CHECK` cannot be compiled out completely, because they are intended to behave like that.

All at once

By default, reporting of assertions is enabled globally if the symbol NDEBUG is not defined, and is disabled if it is defined. However, if NDEBUG is defined and MOD_ASSERT_REPORT is also defined, reporting of assertions is enabled. This can be useful if you want to deliver an executable to a customer with assertions enabled. Likewise, if NDEBUG is not defined, you can disable reporting of assertions globally by defining MOD_ASSERT_DONT_REPORT.

By default, reporting of checks is also enabled globally if the symbol NDEBUG is not defined, and is disabled if it is defined. However, if NDEBUG is defined and MOD_CHECK_REPORT is also defined, reporting of checks is enabled. This can be useful if you want to deliver an executable to a customer with the reporting of checks enabled. Likewise, if NDEBUG is not defined, you can disable reporting of checks globally by defining MOD_CHECK_DONT_REPORT.

Disabling the reporting of Modular Assertions per source code file

You can disable the reporting of assertions per source code file, by defining MOD_ASSERT_DONT_REPORT_FILE before including `modassert/assert.hpp`. This overrides MOD_ASSERT_REPORT.

Likewise, if reporting of assertions is disabled globally, you can enable reporting of assertions per source code file, by defining MOD_ASSERT_REPORT_FILE before including `modassert/assert.hpp`. This overrides MOD_ASSERT_DONT_REPORT and NDEBUG.

You can do the same for checks with the symbols MOD_CHECK_DONT_REPORT_FILE and MOD_CHECK_REPORT_FILE

Disabling the reporting of Modular Assertions per level

Defining MOD_ASSERT_LEVEL to an integer allows you to disable the reporting of assertions per level. Assertions with a level will not be reported if `level < MOD_ASSERT_LEVEL`. If MOD_ASSERT_LEVEL is not defined, it is set to `0`.

- `ModAssert::Info` has a corresponding level 0
- `ModAssert::Warning` has a corresponding level 1
- `ModAssert::Error` has a corresponding level 2
- `ModAssert::Fatal` has a corresponding level 3

So `0` will disable the reporting of no assertions, `1` will only disable the reporting of assertions with `ModAssert::Info`, `2` will only disable the reporting of assertions with `ModAssert::Info` and `ModAssert::Warning`, and so on.

Likewise you can define MOD_CHECK_LEVEL to an integer to disable checks up to a level.

These symbols can be defined application wide at the commandline of your compiler (or equivalently in your project settings), or in a source code file before including `modassert/assert.hpp`.

Note that this also affects assertions without a level group, because by default `ModAssert::Error` is added. So if MOD_ASSERT_LEVEL is defined to be `3`, assertions like

```
MOD_ASSERT(rbEQUAL(a,5));
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup1("group 1");
MOD_ASSERT_G(modAssertGroup1, rbEQUAL(a,5));
```

will not be reported because their level is 2, but not assertions like

```
MOD_ASSERT_G(ModAssert::Fatal, rbEQUAL(a,5));
ModAssert::Group<ModAssert::ReportFailure> modAssertGroup1("group 1");
MOD_ASSERT_G(modAssertGroup1 % ModAssert::Fatal, rbEQUAL(a,5));
```

because these already have a level, so `ModAssert::Error` is not added to these, and their level is 3.

Note: MOD_ASSERT_REPORT, MOD_ASSERT_DONT_REPORT, MOD_ASSERT_REPORT_FILE, MOD_ASSERT_DONT_REPORT_FILE, MOD_ASSERT_LEVEL, MOD_CHECK_REPORT, MOD_CHECK_DONT_REPORT, MOD_CHECK_REPORT_FILE, MOD_CHECK_DONT_REPORT_FILE and MOD_CHECK_LEVEL should not be defined when building the ModAssert library. Theses symbols only affect code that includes `modassert/assert.hpp`. The ModAssert librrary doesn't have code that includes that file.

Disabling the reporting of Modular Assertions per group

Finally you can disable reporting of assertions per group, by changing the groups type from `ModAssert::Group<ModAssert::ReportFailure>` to `ModAssert::Group<ModAssert::ReportNone>`.

Knowing whether assertions or checks are reported

You can find out in your own code whether reporting of assertions is enabled at compile time, by checking the symbol `MOD_ASSERT_REPORTS`, which is non-zero if it is enabled, and `0` if it is not. This can be useful if you need some extra code to perform an assertion, which is otherwise not needed. This applies to both globally disabling reporting of assertions at compile time (see the section called “All at once”) and to disabling reporting of assertions in individual source code files at compile time (see the section called “Disabling the reporting of Modular Assertions per source code file”). You can do the same for checks with `MOD_CHECK_REPORTS`. There is no similar functionality for groups or levels that are disabled.

```
#if MOD_ASSERT_REPORTS
    int *array = ExpensiveCalculation();
    for (int i=0; i<100; ++i)
    {
        MOD_ASSERT(rbLESS(array[i], 5));
    }
    delete array;
```

```
#endif
```

You can also use it as the condition in an if-statement. This has the advantage that the code in the block will still be compiled if it is 0, so you can find compile errors sooner. Most compilers will then optimize the code away just as well as with `#if MOD_ASSERT_REPORTS`

You can also use the macro `IF_MOD_ASSERT_REPORTS`, which takes one argument, that expands to its argument if reporting of assertions is enabled at compile time, and to nothing otherwise, `IF_NOT_MOD_ASSERT_REPORTS`, which takes one argument, that expands to its argument if reporting of assertions is disabled at compile time, and to nothing otherwise, and `IF_MOD_ASSERT_REPORTS_ELSE`, which takes two arguments, that expands to its first argument if reporting of assertions is enabled at compile time, and to the second argument otherwise.

You can use the macros `IF_MOD_CHECK_REPORTS`, `IF_NOT_MOD_CHECK_REPORTS` and `IF_MOD_CHECK_REPORTS_ELSE` that are equivalent for checks.

This has the advantage that short statements such as declarations can remain oneliners. This is e.g. useful to compare a value to an old value in an assertion, that you want to optimize away if assertions are compiled out.

```
void Container::AddElement(int x)
{
    IF_MOD_ASSERT_REPORTS(int old_size = size();)
    ...
    MOD_ASSERT(rbEQUAL(old_size+1, size()));
}
```

Note: because these are macros, you can't have commas in it, unless they're inside an expression. So in the above example you can't define more than one variable in it, but of course you could e.g. have more than one `IF_MOD_ASSERT_REPORTS`. Another option is to define a macro and use that inside one of these macros:

```
#define DECLARATIONS int old_size=size(), old_a=a;
IF_MOD_ASSERT_REPORTS(DECLARATIONS)
...
MOD_ASSERT(rbEQUAL(old_size+1, size()));
MOD_ASSERT(rbEQUAL(old_a*2, a));
```

but this makes your code less readable.

The argument can even be more code with blocks etc., just like for optional actions and failure actions, but in those cases it's probably better to use `#if MOD_ASSERT_REPORTS`.

Note that the above example doesn't work with `MOD_VERIFY` and its variations, because when reporting of assertions is disabled, these are not compiled out completely, and the arguments are still evaluated, so the variable is needed.

```
void Container::AddElement(int x)
{
    IF_MOD_ASSERT_REPORTS(int old_size = size();)
    ...
    // error, old_size not defined when reporting of assertions is disabled:
    MOD_VERIFY(rbEQUAL(old_size+1, size()));
}
```

but they still can be used in parameter lists and optional actions, because these are compiled out when reporting of assertions is disabled (but not in failure actions because these are never compiled out).

```
void Container::AddElement(int x)
{
    IF_MOD_ASSERT_REPORTS(int old_size = size();)
    ...
    // ok, old_size not defined but not used when reporting of assertions is disab
    MOD_VERIFY_P(old_size, rbEQUAL(n, 10));
}
```

The same remark applies for `MOD_CHECK`, because its condition is always evaluated, so this doesn't compile if checks are not reported:

```
void Container::UpdateA(int x)
{
    IF_MOD_CHECK_REPORTS(int old_a = a;)
    a = f(a, x);

    // error, old_a not defined when reporting of checks is disabled:
    MOD_CHECK(rbLESS(old_a, a));
}
```

but you could use it in the parameterlist or the optional action:

```
void Container::UpdateA(int x)
{
    IF_MOD_CHECK_REPORTS(int old_a = a;)
    a = f(a);

    // ok, old_a not defined but not used when reporting of checks is disabled:
    MOD_CHECK_P(old_a, rbLESS(a, 10));
}
```

Using ModAssert with DLLs

When an application uses one or more DLLs, of which more than one use ModAssert, it may be preferable that all assertions and checks are processed by the same instance of ModAssert. This can be achieved with the functions `GetState` and `SetState`, which are declared in the include file `modassert/handler.hpp`. `GetState` returns a pointer to an object of the class `State`, that wraps the responder, all the loggers, hooks and other information that was set in the setup of ModAssert, as well as info about which assertions and checks should be ignored (except if determined at compile time). `SetState` forces an instance of ModAssert to use that state instead of its own state.

It is recommended to do the setup of ModAssert in the executable or one of the DLLs, get the `State` pointer with `GetState` in that instance (before or after setup) and call `SetState` with that pointer in the other DLLs and/or executable. This will require an exported function in every DLL that uses ModAssert to either set or get the state.

At the moment this does not work with Visual C++ 6 (this will probably not be fixed, unless there is a clear need for it).

Note that ModAssert and Rich Booleans still have to be linked in every executable and DLL that uses ModAssert.

There are examples of how to do this in the `demos` folder:

- The folder `demos\LibInDLL` contains the projects `DemoLibInDLL-<compiler>`, that builds a DLL where ModAssert is setup, and `DemoLibInDLL_AssertionInExe-<compiler>` that builds an executable with a failing assertion that is reported by ModAssert in the DLL.
- The folder `demos\LibInExe` contains the projects `DemoLibInExe-<compiler>`, that builds an executable where ModAssert is setup, and `DemoLibInExe_AssertionInDLL-<compiler>` that builds a DLL with a failing assertion that is reported by ModAssert in the executable.

Disabling condition text

If you define the symbol `MOD_ASSERT_NO_TEXT` before including `modassert/assert.hpp`, the text of the condition will not be passed on to the responders and loggers. A null pointer will be given instead. This has the advantage that it reduces the executable size. This is recommended for release builds if ModAssert reports, because the text of the condition is not so important. The number of failing assertions should be considerably lower by that time, and if a condition fails, you can still look up the condition in the source code (supposing that it is still on the same line).

If ModAssert doesn't report, defining this symbol has no effect.

Using the available responders and loggers

The assertion information is only useful if it is displayed or logged. You can derive classes from abstract classes to handle the information when an assertion or check is reported (see the section called “Processing assertions (advanced)”). But there are also a few implementations provided, that can be easily reused in your application, if it uses the same environment.

Windows 32

For Windows 32bit follow these steps:

- Build the project ModAssertWin32 in the configuration(s) that you need.
- Add `$(MODASSERT)/include` to your resources include path.
- Add `modassert/modassert.rc` to your resources. To do this in Visual Studio, right click your resources and choose 'Resource includes', and add the line `#include "modassert/modassert.rc"` under 'Compile-time directives'.
- If you want to run your application under Windows XP and you don't use MFC, add these lines to the initialization of your application:

```
INITCOMMONCONTROLSEX InitCtrls;
InitCtrls.dwSize = sizeof(InitCtrls);

InitCtrls.dwICC = ICC_WIN95_CLASSES;
InitCommonControlsEx(&InitCtrls);
```

Include `commctrl.h` to do this, and link to `comctl32.lib`. This is necessary to be able to use the listcontrol in the assertion dialog box. Otherwise the dialogbox simply won't be shown.

- Link to the appropriate STD version of ModAssertWin32, ModAssert and RichBool.
- Call `ModAssert::SetupForWin32(HINSTANCE h)` or `ModAssert::SetupForWin32(HINSTANCE h, ModAssert::Mode mode)` during the initialization of your application, where `h` is the instance handle of your application. See the section called “Overriding the default behaviour” for more info about the mode. Include the file `modassert/Win32Handler.hpp` to call this method.

Note: with Visual C++, the ModAssert, ModAssertWin32 and RichBool library are linked to automatically if you include the headerfile `modassert/Win32Handler.hpp`, so you can skip the fifth step above.

Note: with gcc, you should link to the modassertwin32 library before the modassert library, so use "-lmodassertwin32 -lmodassert".

See the project in `demos/Win32` or `demos/MFC` for an example.

An alternative that gives you more control is to include the headerfiles that start with Win32 in the `include/modassert` directory, instantiate the objects that you want yourself and give them to ModAssert.

If an assertion or check is reported, this will trace its information to the trace window. Furthermore, it does the following according to the mode:

- in testMode: if the assertion or check is reported from the main thread, it will display a dialog with all the available information, that gives you the choice of breaking into the debugger (if a debugger is attached to the process), abort, stop displaying the assertion, etc. Check all the responses that you want, or none if you want nothing to be done, and click the ok-button. If the assertion or check is reported from another thread, the application automatically breaks into the debugger (if a debugger is attached to the process, otherwise you will only know that there was an assertion from the trace window, or from another assertion logger that you added).
- In continueWithWarningOnFailure and terminateOnFailure mode, it will show a warning in a messagebox for the appropriate level, with the parameters of the assertion or check macro, or a messagebox that the application will end, at least if it happens on the main thread. It is recommended to give meaningful names to the variables that you provide in an assertion or check if a warning is given, because this is meant for users of your application.

Info providers: The dialog, the trace window and other loggers that you add, will show the date and time, the current directory, the thread id, the amount of free memory, the total and free space on all drives, and the value of `GetLastError()` if it was not 0, with the corresponding text. At the beginning of every logfile the OS version, the processor type and the processid will be given. Note: the OS version can't detect Windows Vista.

Before calling `ModAssert::SetupForWin32()` you can optionally call the following methods. They only have effect if they are called before `ModAssert::SetupForWin32()`.

- `void SetUseRemoteDrives(bool b)`: specifies whether remote drives should be used or not. By default remote drives are not used, because requesting info about them may be slow.
- `void SetUseDrive(char drive, bool b)`: specifies whether the drive with the given letter should be used or not. By default, drives a: and b: are not used, while all other drives will be used if they are present.

Getting the objects

You can get a reference to the displayer, the trace logger and the infoproviders that are used after calling `ModAssert::SetupForWin32()` with these functions that are declared in `modassert/Win32Handler.hpp`:

- `Responder& GetWin32Displayer()`: returns a reference to the object that displays the assertion info in a dialog box
- `Logger& GetWin32Trace()`: returns a reference to the object that logs the info using `OutputDebugString`
- `InfoProviders::InfoProvider& GetWin32ThreadIdInfoProvider()`: returns a reference to the object that gives the thread id
- `InfoProviders::InfoProvider& GetWin32LastErrorInfoProvider()`: returns a reference to the object that gives the last system error
- `InfoProviders::InfoProvider& GetWin32TimeInfoProvider()`: returns a reference to the object that gives the time and date
- `InfoProviders::InfoProvider& GetWin32ProcessIdInfoProvider()`: returns a reference to the object that gives the processid
- `InfoProviders::InfoProvider& GetWin32OSVersionInfoProvider()`: returns a reference to the object that gives a description of the operating system
- `InfoProviders::InfoProvider& GetWin32CurrentDirectoryInfoProvider()`: returns a reference to the object that gives the current directory
- `InfoProviders::InfoProvider& GetWin32ProcessorInfoProvider()`: returns a reference to the object that gives a description of the processor
- `InfoProviders::InfoProvider& GetWin32TotalMemoryInfoProvider()`: returns a reference to the object that tells how much RAM the computer has
- `InfoProviders::InfoProvider& GetWin32FreeMemoryInfoProvider()`: returns a reference to the object that tells how much RAM is available
- `InfoProviders::InfoProvider& size_t GetNrDetectedDrives()`: returns how many drives were detected on the computer, which is also the number of associated infoproviders that give info about the drive and how much free space there is on it
- `InfoProviders::Win32DiskInfoProvider& GetDiskInfoProvider(size_t idx)`: returns a reference to the object that gives information about the drive with the index `idx`. `idx` should be between 0 and `GetNrDetectedDrives()-1`.
- `InfoProviders::Win32FreeDiskSpaceInfoProvider& GetFreeDiskSpaceInfoProvider(size_t idx)`: returns a reference to the object that tells how much free space there is left on the drive with the index `idx`. `idx` should be between 0 and `GetNrDetectedDrives()-1`.
- `InfoProviders::Win32DiskInfoProvider& GetDiskInfoProvider(char drive)`: returns a reference to the object that gives information about the drive with the letter `drive`.

- `InfoProviders::Win32FreeDiskSpaceInfoProvider& GetFreeDiskSpaceInfoProvider(char drive)`: returns a reference to the object that tells how much free space there is left on the drive with the letter `drive`.

Note: The classes `InfoProviders::Win32DiskInfoProvider` and `InfoProviders::Win32FreeDiskSpaceInfoProvider` have a method `char GetDriveLetter() const`, that returns the drive letter.

wxWidgets

For wxWidgets, follow these steps:

- Build the project ModAssertWxGui in the configuration(s) that you need.
- Define the symbol RICHBOOL_USE_WX for your whole application
- Link to the wxWidgets version of ModAssertWxGui, ModAssert and RichBool
- Call `ModAssert::SetupForWxWidgets()` or `ModAssert::SetupForWxWidgets(ModAssert::Mode mode)` during the initialization of your application. See the section called “Overriding the default behaviour” for more info about the mode. To do this, include `modassert/wxhandler.hpp`.

Note: with Visual C++, the ModAssert, ModAssertWxGui and RichBool library are linked to automatically if you include the headerfile `modassert/wxhandler.hpp`, so you can skip the third step above.

See the project in `demos/Wx` for an example.

An alternative that gives you more control is to include the headerfiles that start with Wx in the `include/modassert` directory, instantiate the objects that you want yourself and give them to ModAssert.

If an assertion or check is reported, this will log the assertion information with `wxLogDebug`. Furthermore, it does the following according to the mode:

- in testMode: if the assertion or check is reported from the main thread, it will display a dialog with all the available information, that gives you the choice of breaking into the debugger (except if NDEBUG is defined), abort, permanently stop displaying the assertion, etc. Check all the responses that you want, or none if you want nothing to be done, and click the ok-button. If the assertion or check is reported from another thread, the application automatically breaks into the debugger (except if NDEBUG is defined, in that case you will only know that there was an assertion from the logs).
- In continueWithWarningOnFailure and terminateOnFailure mode, if an assertion or check is reported in the main thread, it will show a warning in a messagebox for the appropriate level, with the parameters of the assertion or check macro, or a message that the application will end (depending on the mode and the level). It is recommended to give meaningful names to the variables that you provide here, because this is meant for users of your application.

Info providers: the dialog, `wxLogDebug` and other loggers that you add, will show the date and time, the current directory, the amount of free memory, the return value of `wxSysErrorCode()` and `wxSysErrorMsg()`, and the thread id. At the beginning of every logfile the os version and the processid will be given.

Getting the objects

You can get a reference to the displayer, the trace logger and the infoproviders that are used after calling `ModAssert::SetupForWxWidgets()` with these functions that are declared in `modassert/wxhandler.hpp`:

- `Responder& GetWxDisplayer()`: returns a reference to the object that displays the assertion info in a dialog box
- `Logger& GetWxLogger()`: returns a reference to the object that logs the info using `wxLog`
- `InfoProviders::InfoProvider& GetWxThreadIdInfoProvider()`: returns a reference to the object that gives the thread id
- `InfoProviders::InfoProvider& GetWxTimeInfoProvider()`: returns a reference to the object that gives the time and date
- `InfoProviders::InfoProvider& GetWxProcessIdInfoProvider()`: returns a reference to the object that gives the processid
- `InfoProviders::InfoProvider& GetWxOsVersionInfoProvider()`: returns a reference to the object that gives a description of the operating system
- `InfoProviders::InfoProvider& GetWxCurrentDirectoryInfoProvider()`: returns a reference to the object that gives the current directory
- `InfoProviders::InfoProvider& GetWxFreeMemoryInfoProvider()`: returns a reference to the object that gives the amount of free memory
- `InfoProviders::InfoProvider& GetWxSysErrorInfoProvider()`: returns a reference to the object that gives information about the last system error

Note: if you get link errors, it may be necessary to put `libmodassertWxGui-*.lib` *before* `libmodassert-*.lib` in the link options.

Integrating with wxASSERT

If you use wxWidgets and your application uses a `wxApp` derived object, you could override `void OnAssert(const wxChar *file, int line, const wxChar *func, const wxChar *cond, const wxChar *msg)` (or `void OnAssert(const wxChar *file, int line, const wxChar *cond, const wxChar *msg)` if you use an old version of wxWidgets) to redirect the messages of existing `wxASSERT` and other wxWidgets assertion macros to the ModAssert framework, by calling `ModAssert::HandleAssert`. This way the assertion will be logged and shown to the user like the assertion macros of the ModAssert framework. In this case you can't use all the functionality of the ModAssert framework, like choosing to stop displaying an individual assertion, but at least you can redirect `wxASSERT` and other wxWidgets assertions in code that you can't change, to the same dialog and loggers. See the file `ModAssertWxApp.cpp` in the ModAssertDemoWx project for an example. One disadvantage that may be important, is that ModAssert doesn't know if the assertion comes from a `wxCHECK` kind of macro or not. In the example it assumes that it doesn't.

For a better alternative that requires adding an include file, see the section called “Redefining existing assertion macros”

Console applications

For console applications, follow these steps:

- Build the project ModAssertConsole in the configuration(s) that you need.
- Link to the STD version of ModAssertConsole, ModAssert and RichBool
- Next, call `ModAssert::SetupForConsole()` or `ModAssert::SetupForConsole(ModAssert::Mode mode)` during the initialization of your application. See the section called “Overriding the default behaviour” for more info about the mode. This function is declared in `modassert/consolehandler.hpp`.

Note: with Visual C++, the ModAssert, ModAsserConsole and RichBool library are linked to automatically if you include the headerfile `modassert/consolehandler.hpp`, so you can skip the second step above.

Note: with gcc, you should link to the modassertConsole library before the modassert library, so use "-lmodassertConsole -lmodassert".

See the project in `demos/Console` for an example.

If an assertion or check is reported, this will log the assertion information to `std::cerr`. Furthermore, it does the following according to the mode:

- in testMode: it will display the actions that you can choose from, like breaking into the debugger (except for a Release version), ignore the assertion, abort, stop displaying the assertion, etc (note: this hasn't been tested with a failing assertion in another thread). You can enter one or more responses by entering all their letters, or 'I' if no action should be taken.
- In continueWithWarningOnFailure and terminateOnFailure mode, it will show a warning on std::cout, for the appropriate level, with the parameters of the assertion or check macro. It is recommended to give meaningful names to the variables that you provide here, because this is meant for users of your application.

Info providers: This will also show the date and time on `std::cerr` and other loggers that you add.

Getting the objects

You can get a reference to the displayer, the logger and the infoproviders that are used after calling `ModAssert::SetupForConsole()` with these functions that are declared in `modassert/Win32Handler.hpp`:

- `Responder& GetConsoleDisplayer()`: returns a reference to the object that asks for a response
- `Logger& GetScreenLogger()`: returns a reference to the object that logs the info on `std::cerr`
- `InfoProviders::InfoProvider& GetConsoleTimeInfoProvider()`: returns a reference to the object that gives the time and date

Overriding the default behaviour

The functions `SetupForConsole`, `SetupForWin32` and `SetupForWxWidgets` have overloads that take an argument of the type `ModAssert::Mode` (for `SetupForWin32` after the `HINSTANCE` argument), that specifies what should happen if an assertion fails. Note that this only has an influence if reporting of assertions is enabled. The possible values of this argument are:

- `ModAssert::testMode`: shows information about the assertion. How it is shown depends on the function that you called. This is the default if `NDEBUG` is not defined.

- `ModAssert::continueSilentlyOnFailure`: continues silently if an assertion fails. This is the default if `NDEBUG` is defined.
- `ModAssert::continueWithWarningOnFailure`: this shows a warning to the user if an assertion fails and continues the application, if the level is `ModAssert::Warning` or higher. You can adjust this level by calling `ModAssert::SetMinimumWarningLevel` with the minimum level as the argument.
- `ModAssert::terminateOnFailure`: this terminates the application if an assertion fails, if the level is `ModAssert::Fatal` or higher. If the level of the assertion is at least `ModAssert::Warning` but less than `ModAssert::Fatal`, it shows a warning. You can adjust these levels by calling `ModAssert::SetMinimumTerminateLevel` and `ModAssert::SetMinimumWarningLevel` with the minimum level as the argument.

Calling the setup function with the argument `ModAssert::testMode`, is useful if an error only occurs in release mode, but you preferably should remove it again afterwards, because this is probably not the behaviour that the users of your application want. A failing check will not terminate the application or show a warning, and will only show debug info if `ModAssert::testMode` is used. In any case the loggers and hooks are still called.

After calling a setup function, you can call `ModAssert::GetWarningMessage()` that returns a pointer to a `WarningMessage` object, and `ModAssert::GetTerminateMessage()` that returns a pointer to a `TerminateMessage` object. These are called by ModAssert if a warning or a terminate message has to be shown. They both have a method `void SetText(const std::string &str)` to change their message, and `const std::string & GetText()` to retrieve it. Depending on the mode that you supplied to the setup function, these pointers may be NULL or not, so check their return value.

If reporting of assertions is disabled at compile time, assertions are ignored, and the application will not terminate, show a warning or show debug info due to a failing assertion.

It is recommended to define the symbol `MOD_ASSERT_REPORT` for the whole application, so that reporting of assertions is enabled, and call any of the functions `SetupForConsole`, `SetupForWin32` or `SetupForWxWidgets` with the appropriate argument if the default is not acceptable for your application. It is also recommended to have at least one logger, so you know why the application went wrong. In source files where performance is critical, you can define the symbol `MOD_ASSERT_DONT_REPORT_FILE` to disable assertions in that file, or give the assertions in the specific code a group argument that turns off reporting in release mode (by selecting its type with conditional compiling).

The optional logger

Optionally you can call `ModAssert::SetOptionalLogger(const ModAssert::Logger *logger, const char *text=0, bool extraInfo=false)` to give the user the option to log to this logger, if the responder allows this. This is useful if you want certain interesting assertions to be logged in another file. This method is defined in `modassert/handler.hpp`. If `extraInfo` is `true`, the logger also gets the begin and end message, like loggers that are added with `ModAssert::AddLogger(...)`. The responders in the libraries ModAssertWin32, ModAssertWxGui and ModAssertConsole all allow you to log to the optional logger if there is one. See the demos for examples.

If you write your own responder, and want to use the optional logger, you can retrieve it with `const Logger* GetOptionalLogger()`, and the corresponding text with `const char* GetOptionalLoggerText()`.

If you want to disable the use of the optional logger, call `ModAssert::SetOptionalLogger` with a NULL pointer for the logger.

Processing assertions (advanced)

This section explains how to process assertions if you don't use any of the provided setup functions described in the previous sections, e.g. if you use a different GUI. It is also of interest if you want to modify part of the behaviour a setup function described in the previous sections.

It also explains how to write your own loggers, which can be useful even if you use a setup function described in the previous sections.

The Responder object

If a modular assertion or check is reported, the information about it is passed to the virtual method

```
ModAssert::Response OnAssert(const Properties & properties,
                             const Result & result,
                             bool display);
```

of the class `ModAssert::Responder`, which has to be overridden by a derived class. See below for a description of the classes `Properties` and `Result`. `display` tells whether the assertion should be displayed. Responders that display an assertion to the user, should check this; if it is `false`, it should not display it. The value of this boolean is determined by ModAssert, and can be influenced with `ModAssert::SetDisplayAll(...)`, `ModAssert::SetDisplayInFile(...)`, the method `SetDisplay()` of the group and level objects, the filter that is set with `ModAssert::SetDisplayFilter(...)` and the boolean that every statement assertion and check keeps, which is set to `false` if the user selects 'stop displaying this assertion' (expression assertions and checks don't have such a boolean because of the impossibility to have a static variable in an expression).

To handle assertions with a responder, derive a class from `ModAssert::Responder` and override this method. Your class could e.g. show a dialog to the user, or print it out, and ask for an action; or it could decide on its own what response to return. Then create an object of your class, and pass it to `ModAssert::SetResponder` (defined in `modassert/handler.hpp`). Only one object of the type `ModAssert::Responder` can be active at a time. To disable displaying of assertions, call `ModAssert::SetResponder(NULL)`. Important: call this function only after `main` is entered (see the section called “When ModAssert is active”).

The return value is a `ModAssert::Response`, and can be `Abort`, `StopDisplayingThis`, `StopDisplayingFile`, `StopDisplayingAll`, `BreakIntoDebugger`, `Optional` or `StopDisplayingCustom<N>`, where N can be from 1 to 8 (all in the `ModAssert` namespace).

- Use `Abort` if you want the application to end by calling `abort()`.
- Use `StopDisplayingThis` if you no longer want this particular assertion displayed (this has no effect on expression assertions and checks).
- Use `StopDisplayingFile` if you no longer want assertions in the same file displayed.
- Use `StopDisplayingAll` if you no longer want any assertions displayed.
- Use `BreakIntoDebugger` if you want to break into the debugger. Note that this may crash your application if no debugger is attached.

- Use `Optional` if you want the optional action to be executed. If there is no optional action, this has no effect.
- Use `StopDisplayingCustom<N>` to stop displaying assertions of a given type, like `ModAssert::Error` or a custom group.

It can also be a combination of these, by or'ing them. If no action has to be taken, return `(ModAssert::Response)0`. The order in which they are processed after the responder returns, is as follows:

- First `StopDisplayingThis`, `StopDisplayingFile`, `StopDisplayingAll` and `StopDisplayingCustom<N>` are processed.
- Next `BreakIntoDebugger` is processed.
- Next `Optional` is processed.
- Next `Abort` is processed.

Example 1. Overriding `ModAssert::Responder::OnAssert`

```
ModAssert::Response MyAssertHandler::OnAssert(
  const Properties &properties,
  const Result &assertionResult,
  bool display)
{
 // if your responder needs the attention of the user,
 // check the value of display:
 if (!display)
  return (ModAssert::Response)0;

 // show information to the user
 ...

 // return a response:
 return ModAssert::BreakIntoDebugger | ModAssert::StopDisplayingCustom1;
}
```

The class WarningMessage

`WarningMessage` is a class that is defined in the file `modassert/warning_message.hpp` and is derived from `Responder`, that is called when an assertion fails and a setup function was called with the mode `continueWithWarningOnFailure` or `terminateOnFailure` (depending on the level of the assertion). It has the method `void SetText(const std::string & str)` to set the message to display, and `const std::string& GetText()` to retrieve that message.

If you need a custom warning message, derive your own class of `WarningMessage` and override `OnAssert` (of the class `Responder`). Preferably use the message that is returned by its parent method `GetText()`.

Then create an object of your class, and pass a pointer to it to `ModAssert::SetWarningMessage`.

The class TerminateMessage

`TerminateMessage` is a class that is defined in the file `modassert/terminate_message.hpp`, that is called when the application will terminate due to a failing assertion fails (i.e. if a setup function was called with the mode `terminateOnFailure` and the level of the assertion is high enough). It has the method `void SetText(const std::string & str)` to set the message to display, and `const std::string& GetText()` to retrieve that message.

If you need a custom terminate message, derive your own class of `TerminateMessage` and override `void OnAssert(const Properties &properties, const Result &result)`. Preferably use the message that is returned by its parent method `GetText()`.

Then create an object of your class, and pass a pointer to it to `ModAssert::SetTerminateMessage`.

The function Setup

Instead of calling `ModAssert::SetResponder`, `ModAssert::SetWarningMessage` and `ModAssert::SetTerminateMessage`, you can also call `void ModAssert::Setup(const ModAssert::Responder *responder)` which only sets the responder if `NDEBUG` is not defined, and calls `UseBool::SetCheck` with `true` if `NDEBUG` is not defined, `false` otherwise.

For more control, you can call the overloaded `void ModAssert::Setup(ModAssert::Mode mode, const ModAssert::Responder *responder, ModAssert::WarningMessage *warningMessage, ModAssert::TerminateMessage *terminateMessage)`, which ignores whether `NDEBUG` is defined, and takes care of setting up ModAssert to handle assertions according to the mode. It calls `UseBool::SetCheck` with `true` if `mode` is `ModAssert::testMode`, `false` otherwise.

The Logger objects

In addition, the information is also given to a number of objects that implement the interface `ModAssert::Logger`, by calling the pure virtual method

```
void OnAssert(const Properties & properties,
              const Result & result);
```

on them, which has to be implemented by derived classes. The arguments are the same as for `ModAssert::Responder::OnAssert`, except that `display` is not given. Note that this method returns `void`. See below for a description of the classes `Properties` and `Result`.

In addition, this class has another pure virtual method

```
void AddMessage(const std::string & message);
```

For now, this is only called (multiple times) when the logger is added to ModAssert and when it is removed or when the destructor of a `ModAssert::AutoShutdown` object is called, but it may be called at other times in the future. Derived classes should implement this method. Note that they should add a newline themselves if necessary.

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

To use loggers, derive one or more classes from `ModAssert::Logger`, implement `OnAssert` and `AddMessage`, and add them by calling `ModAssert::AddLogger` with pointers to the objects. This method returns an id of the type `size_t`, that is unique for every logger and at least 1. You can remove a logger again by calling `ModAssert::RemoveLogger` with a pointer to the object or its id. These

functions are defined in `modassert/handler.hpp`. ModAssert doesn't copy loggers, so the logger object must exist until it is removed (manually or by the destructor of a `ModAssert::AutoShutdown` object) or until the application ends. Important: call `ModAssert::AddLogger` only after `main` is entered (see the section called “When ModAssert is active”).

Note: the `ModAssert::Context` object in `properties` and the `ModAssert::ParameterList` object in `result` are temporay objects on the stack, so they no longer exist after `OnAssert` is called on these objects, so you can't store pointers to them for later usage. However, the `ModAssert::ParameterList` object can be cloned.

Hooks

In addition, the information is also given to a number of objects that implement the interface `ModAssert::Hook`, by calling the virtual method

```
void OnAssert(const Properties & properties,
              const Result & result);
```

on them, which has to be implemented by derived classes. These are called before any logger or responder is notified of the assertion. The arguments are the same as for `ModAssert::Logger::OnAssert`. See below for a description of the classes `Properties` and `Result`.

To use hooks, derive one or more classes from `ModAssert::Hook`, implement `OnAssert`, and add objects of those classes by calling `ModAssert::AddHook` with a pointer to the objects. You can remove a hook again by calling `ModAssert::RemoveHook` with a pointer to the object. These two functions are defined in `modassert/handler.hpp`. The hook object must exist until it is removed or until the application ends. Important: call `ModAssert::AddHook` only after `main` is entered (see the section called “When ModAssert is active”).

Note: the `ModAssert::Context` object in `properties` and the `ModAssert::ParameterList` object in `result` are temporay objects on the stack, so they no longer exist after `OnAssert` is called on these objects, so you can't store pointers to them for later usage. However, the `ModAssert::ParameterList` object can be cloned.

Hooks are only useful in some rare situations. One situation is storing the return value of `GetLastError()` in Windows. The return value of that function is altered if you log to a file or display a dialog box, so it has to be stored before any logger or the responder is called. See the Win32 demo for an example of how this is done. Another usage may be a 'rainy day fund'.

The class `ModAssert::Properties`

A `ModAssert::Properties` object represents all the attributes of an assertion or a check that are always the same. It has the following public methods:

- `const CategoryBase* GetCategory() const`: returns an object that describes the category of the assertion or check. The caller should not delete this.
- `bool IsUnconditional()`: returns whether the assertion or check is unconditional (e.g. `false` for `MOD_ASSERT`, `true` for `MOD_FAIL`)
- `const Context& GetContext() const`: returns the context
- `const char* GetCondition() const`: returns the condition as a character string. The caller should not delete this.

- `const GroupList* GetGroupList() const`: returns a list of all the groups that were added (or were added as a default). The caller should not delete this.
- `const char* GetOptional() const`: returns a character string with the optional action; this is `NULL` if no optional action was given. The caller should not delete this.
- `bool CanStopDisplayingThis() const`: returns whether the flag `StopDisplayingThis` in the return value of a responder will have any influence (e.g. it will have no influence on expression assertions and checks). If this returns false, your responder should not offer the possibility to stop displaying an assertion or check.

The class `ModAssert::Result`

A `ModAssert::Result` object represents all the attributes of an assertion or a check that can vary every time it is evaluated. It has the following public methods:

- `bool Succeeded() const`: returns `false` if the condition failed, `true` otherwise
- `const RichBool::SharedAnalysis GetAnalysis() const`: returns the analysis of the condition if a Rich Boolean was used
- `const ParameterList* GetParameterList() const`: returns the parameterlist if parameters were added; `NULL` otherwise. The caller should not delete this.

The class `ModAssert::CategoryBase`

A `ModAssert::CategoryBase` object represents the category of an assertion or a check. It has the following public methods:

- `bool IsAssertion() const`: returns `true` if it represents an assertion, `false` otherwise
- `bool IsCheck() const`: returns `true` if it represents a check, `false` otherwise
- `const char* GetName() const`: returns the name of the category it represents as a character string. The caller should not delete this. For now this can only be `"assert"` or `"check"`.

When ModAssert is active

It is important that `ModAssert::SetResponder`, `ModAssert::AddLogger` and `ModAssert::AddHook` should only be called after the `main` function has been entered and before it is left. I.e. it should not be called from the constructor or destructor of a global object. These functions are thread safe. Furthermore, calling one of these functions activates ModAssert and failing assertions are synchronized to be thread safe, but before `main` is entered, you can not be sure that the multithreading library of your platform is initialized, so this could lead to a crash.

Likewise, when the function `main` is exited, different libraries may shutdown. Also the multithreading libraries on many platforms. Since ModAssert synchronizes assertions, this may lead to a crash if an assertion is reported after your multithreading library has shut down. Assertions may be present in code called by destructors of global objects, so this danger is real.

Therefore ModAssert offers you a way of letting it know that it should shut down as well, and ignore assertions from that moment on. This is done in an exceptionsafe way, by declaring an object of the class `ModAssert::AutoShutdown`, which is declared in `modassert/handler.hpp`. Its destructor

sends a message to the loggers that ModAssert stops logging, removes all loggers and the responder. Therefore you should declare a non-static object of this class on the stack in your `main` (or another similar function), right before or after you initialize ModAssert by setting a responder or adding a logger. You should only create one such object in your application. So your `main` function may look like this:

```
#include "modassert/assert.hpp"
#include "modassert/ConsoleHandler.hpp"

int main(int argc, char* argv[])
{
 ModAssert::AutoShutdown modAssertAutoShutdown;

 // Assume we use the Console handler:
 ModAssert::SetupForConsole();

 ...
}
```

Note that loggers should be defined before the `ModAssert::AutoShutdown` object (or be global objects), unless you remove them yourself before its destructor is called. Otherwise the destructor of `ModAssert::AutoShutdown` will try to log the end message to loggers that no longer exist.

In some environments you can't change `main` (e.g. wxWidgets), but you can override an application objects methods to do the same. In that case, simply create a `ModAssert::AutoShutdown` object in a method that is called when the application ends.

```
bool MyApp::OnInit()
{
 // these objects are static, otherwise they would be destroyed
 // as soon as this method is left:
 static MyResponder myResponder;
 static MyLogger myLogger;

 ModAssert::SetResponder(&myResponder);
 ModAssert::AddLogger(&myLogger);

 ... // other setup

 return TRUE;
}

int MyApp::OnExit()
{
 // Add this line to make sure that ModAssert exits
 // gracefully when the application ends.
 // This will actually be done from the destructor of this object.
 ModAssert::AutoShutdown modAssertAutoShutdown;

 // other shut down code
 ...
}
```

Some programmers find it tempting to derive a class from `ModAssert::Responder`, `ModAssert::Logger` or `ModAssert::Hook`, and let the constructor automatically add it to

ModAssert, and let the destructor remove it again, so you only have to define them as a global object. But in the view of synchronization, this is clearly not a good idea.

Filters for the responder, the loggers, the hooks and the infoproviders

The functions `ModAssert::AddLogger`, `ModAssert::AddHook` and `ModAssert::AddInfoProvider` can take a second argument, a pointer to an object derived from `ModAssert::Filter`, that has a default value of `NULL`. These objects decide for their corresponding logger and hook whether it should be notified about an assertion.

You can also give a pointer to such an object to the method `ModAssert::SetDisplayFilter` (defined in `modassert/handler.hpp`), to let the responder decide whether it should display an assertion. Responders are always notified of an assertion, but have an extra argument `bool display`, that is determined by that filter (amongst other things), that tells whether an assertion should be displayed. If a `NULL` value is given, there is no influence.

`ModAssert::Filter` has two virtual methods:

```
bool Accept(const Properties & properties,
            const Result & result);

bool Accept(const LogType & logType);
```

which should return true if the filter accepts the assertion or logging type. In the base class they always return `true`. It also has a const non-virtual method:

```
const std::string& GetDescription();
```

which gives a description of the filter.

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

There are several classes derived from `Filter`, described below.

If you derive your own class of `Filter`, you should override at least one of the two `Accept` methods. Furthermore, the constructor should assign a value to the protected member `description`.

FilterIfFailed

`ModAssert::FilterIfFailed` will only accept assertions that failed, i.e. it will not pass succeeding assertions that should be reported. You can use the predefined object `ModAssert::filterIfFailed`, that is of this class, so you don't need to define an object yourself.

FilterIfAssert

`ModAssert::FilterIfAssert` will only accept assertions, i.e. that come from the macros that begin with `MOD_ASSERT`, `MOD_VERIFY` or `MOD_FAIL`. You can use the predefined object `ModAssert::filterIfAssert`, that is of this class, so you don't need to define an object yourself.

FilterIfCheck

`ModAssert::FilterIfCheck` will only accept checks, i.e. that come from the macros that begin with `MOD_CHECK`. You can use the predefined object `ModAssert::filterIfCheck`, that is of this class, so you don't need to define an object yourself.

FilterIfHasGroup

`ModAssert::FilterIfHasGroup` has a constructor that takes a `const char *`, the name of a group. It will only accept assertions that have that group.

FilterMinimumLevel

`ModAssert::FilterMinimumLevel` has a constructor that takes a `int` or a level object. It will only accept assertions that have at least that level.

Note: with Visual C++ 6.0 you can only pass a level object to the constructor.

FilterMaximumLevel

`ModAssert::FilterMaximumLevel` has a constructor that takes a `int` or a level object. It will only accept assertions that have at most that level.

Note: with Visual C++ 6.0 you can only pass a level object to the constructor.

FilterInFile

`ModAssert::FilterInFile` has a constructor that takes a `const char *`, the name of a file. It will only accept assertions that appear in that source file. It is recommended to use this with the predefined symbol `__FILE__`, so that moving source code doesn't change the applications behaviour.

FilterMessages

`ModAssert::FilterMessages` will only accept messages, i.e. no assertions and checks. It is a class without member data, so you can just use the instantiation `ModAssert::filterMessages`.

FilterLogType

`ModAssert::FilterLogType` has a constructor that takes four booleans. The first specifies whether logging of assertions and checks is accepted, the second specifies whether logging when a logger is added is accepted, the third specifies whether logging when a logger is removed is accepted, the fourth specifies whether logging data at other times (not assertions and checks) is accepted (this is not yet used).

Two objects of this type are already provided by ModAssert: `filterNoAssertionsAndChecks`, that doesn't allow assertions and checks, and `filterOnlyAssertionsAndChecks`, that only allows assertions and checks.

FilterNot

`ModAssert::FilterNot` has a constructor that takes a reference to a `ModAssert::Filter` object. It returns the negation of that filter. The filter that is given to the constructor should exist at least as long as the object itself exists.

Two objects of this class are predefined: `ModAssert::filterIfSuccess` that has `ModAssert::filterIfFailed` as an argument in its constructor, `ModAssert::filterIfNotAssert` that has `ModAssert::filterIfAssert` as an argument in its constructor, and `ModAssert::filterIfNotCheck` that has `ModAssert::filterIfCheck` as an argument in its constructor.

FilterAnd

`ModAssert::FilterAnd` has a constructor that takes two references to `ModAssert::Filter` objects. It will only accept assertions that those filters both accept. The filters that are given to the constructor should exist at least as long as the object itself exists.

FilterOr

`ModAssert::FilterOr` has a constructor that takes two references to `ModAssert::Filter` objects. It will only accept assertions that at least one of those filters accepts. The filters that are given to the constructor should exist at least as long as the object itself exists.

The ResponderSwitcher class

In `modassert/responder_switcher.hpp` the class `ModAssert::ResponderSwitcher` is defined. It is derived from `ModAssert::Responder`, but doesn't show the assertion information to the user like other responders. Instead its implementation of `OnAssert` decides on its own which response should be returned and/or what other `ModAssert::Responder` should be called, based on filters that you add to it.

The method `void AddFilter(ModAssert::Filter *filter, ModAssert::Response response)` lets you specify that if the filter accepts the assertion, `response` will be returned. The method `void AddFilter(ModAssert::Filter *filter, const ModAssert::Responder *responder)` lets you specify that if the filter accepts the assertion, `OnAssert` on `responder` will be called with the same arguments, and its return value is returned.

You can give as many filters as you want. The filters will be checked in the order that you added them. The response of the first matching filter will be returned by `OnAssert`. The method `void SetDefaultResponse(Response response)` lets you specify which value should be returned if no matching filter is found, or no filter was added. The method `void SetDefaultResponder(const Responder *responder)` lets you specify a responder that should be called if no matching filter is found, or no filter was added, to return its return value. Only one of these two can be in effect, so they override each others settings. If you don't call any of these, the default response is 0.

The method `bool HasDefaultResponse()` returns whether a const default response is set. The method `bool HasDefaultResponder()` returns whether a default responder is set. Only one of these two can return true. The method `Response GetDefaultResponse()` return the default response if one was set; only call this if `bool HasDefaultResponse()` returns true. `const Responder* GetDefaultResponder()` returns the default responder, if one was set; only call this if `bool HasDefaultResponder()` returns true.

Note: since `ModAssert::ResponderSwitcher` is derived from `ModAssert::Responder`, it is possible to have a circular reference if you add a `ModAssert::ResponderSwitcher` to another `ModAssert::ResponderSwitcher`, possibly resulting in an infinite loop. There is no checking for this, so some attention is needed here.

In the file `modassert/autoresponder.hpp` there is a typedef of `ModAssert::AutoResponder` to `ModAssert::ResponderSwitcher` for backward compatibility.

Providing extra information

It is often useful to add extra information to the loggers and the responder when an assertion failed. Examples are the thread id, time and date, the OS version, the version of your application, a backtrace, or the state of your application (e.g. the documents that are loaded and the active document). You can do this by implementing the interface `InfoProviders::InfoProvider` which is declared in the file `modassert/info.hpp`. It has a constructor that takes a `InfoProviders::Granularity` value, and three virtual methods

```
const std::string& GetType();

bool HasInfo();

std::string GetInfo();
```

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

The `InfoProviders::Granularity` value in the constructor specifies what type of info the infoprovider gives, so ModAssert knows when it should be used. It can be one of the following values:

- `volatileInfo`: this is used for info that is volatile, i.e. that can change from one call to the next in the same application. If it is, it will always be used. Examples are the time, the current directory and the amount of free memory.
- `thread`: this is used for info that is always the same in a thread. The trivial example is the thread id.
- `process`: this is used for info that is always the same in a process. The trivial example is the process id.
- `application`: this is used for info that is always the same in an application. An example is the application name. This is different from `process` because some info may be the same for different instantiations of the same application.
- `machine`: this is used for info that is always the same on a machine. Examples are CPU info and the OS version.

Note: there is also a granularity `customGranularity`. If you need to define other granularities, use values starting at this value. The maximum value is the number of bits in an `unsigned int` minus one.

InfoUser objects can specify for each of these granularities (except `volatileInfo`) if it will receive info from more than one InfoProvider object of that granularity or not. If it will, it will use the info every time an assertion or check is reported as well as when a logger is added or removed; otherwise only when a logger is added, because it will always be the same. This is mostly of concern for loggers, because they get various info when they are added or removed; responders are only called when an assertion or check is reported. E.g. by default, they will only use application specific and machine specific info at the begin, when a logger is added, because it has no use to repeat it every time info is logged to it. If you tell it it is for more than one application, e.g. a log file that is used by many applications, it will always use application specific info, so you know from which application it comes. If you tell it it is for more than one machine,

e.g. a log file that is on a shared network folder, it will always use machine specific info, so you know every time from which machine it comes.

Note: if you have a logfile for many applications on a single machine, the logger will contain machine specific info every time a logger for that logfile is added, so this info will be repeated many times. There is no mechanism in ModAssert to prevent this, but it is not a big problem.

`HasInfo` is implemented by `ModAssert::InfoProvider`, and returns `true`. Derived classes that don't always have useful information, should override this method. This prevents cluttered output if there are many InfoProviders that rarely have useful information. The other two virtual methods have to be implemented by derived classes. `GetType` should return a brief description of the information (e.g. "thread id"), and should always be the same (it is best to create a static const string object in the function, and return that). `GetInfo` should return the information as a string.

`InfoProvider` has one more method that tell what granularity the information has: `Granularity GetGranularity() const` (the value given in the constructor, see above).

Create an object of your derived class, and add it by giving a pointer to it to `ModAssert::AddInfoProvider`, which is declared in `modassert/infostore.hpp`. Optionally you can add a pointer to a filter as a second argument, to determine when the infoprovider should be used. You can remove an InfoProvider again with `ModAssert::RemoveInfoProvider`.

The objects that are added that way are called by any logger or responder that is provided in this package. It is recommended to do this also in loggers and responders that you create yourself. You can iterate over the active infoproviders by calling `ModAssert::beginInfoProviders(...)` and `ModAssert::endInfoProviders()`, also declared in `modassert/infostore.hpp`, which respectively return the iterator to the first and past the last infoprovider. `ModAssert::beginInfoProviders(...)` has two overloaded versions, of which the arguments specify which infoproviders should be ignored when you traverse the range, so you don't have to take into account whether an InfoProvider should be used. Infoproviders for which `HasInfo()` returns `false` are always ignored. `beginInfoProviders(const Properties &p, const Result &r, const InfoUser &iu)` should be used by Responders and Loggers when an assertion or a check is reported. The properties and the result are given to the method `OnAssert`, so you simply pass these. The classes `ModAssert::Responder` and `ModAssert::Logger` are both derived from `ModAssert::InfoUser`, so you can give `*this` for the last argument. `beginInfoProviders(const LogType &lt, const InfoUser &iu)` is used by the class `Logger`, you normally don't need it. The iterators are of the class `InfoProviderIterator`. You can also get the information of all the infoproviders at once in a string by calling `ModAssert::GetAllInfo`. There are two overloaded versions of this function, with the same arguments as `ModAssert::beginInfoProviders`, and an optional `const char *` argument that specifies the separator (default is `"\n"`).

The loggers and responders in ModAssert, as well as `ModAssert::GetAllInfo`, show the information in the format "<type>: <info>" for every InfoProvider that is not filtered out.

See the project Win32, Console or WxWidgets for examples of how all this is done.

Note: this class has nothing to do with the level `ModAssert::Info`.

The class TimeInfoProvider

In the file `modassert/timeinfo.hpp` there is the class `InfoProviders::TimeInfoProvider`, which is derived from `InfoProviders::InfoProvider`. Its implementation of `GetType` returns the string "time", and

its implementation of `GetInfo` returns a string with the time and date. It uses the function `ctime()` in the STD configuration. In the wxWidgets configuration it uses `now.FormatISODate()+' '+now.FormatISOTime()`.

The class ApplicationInfoProvider

In the file `modassert/appinfo.hpp` there is the class `InfoProviders::ApplicationInfoProvider`, which is derived from `InfoProviders::InfoProvider`. Its constructor takes a `std::string` (or `wxString` if you use wxWidgets) with the name (and perhaps the version) of the application. Its implementation of `GetType` returns the string "application", and its implementation of `GetInfo` returns the string that was passed in its constructor. Create an object of this class and add it if you want to use it.

The class InfoUser

The classes `ModAssert::Responder` and `ModAssert::Logger` are both derived from the class `InfoProviders::InfoUser`. This class has functionality to specify at which granularities it works. It can also overrule when certain InfoProviders should give information to it (note however that the filter that is given with an InfoProvider, can decide to not use an InfoProvider when the InfoUser and/or the InfoProvider want to, but not decide to use an InfoProvider when the InfoUser and/or the InfoProvider don't want to). This can be useful in certain situations, e.g. if a logfile is used by different applications, the processid is useful when an assertion is reported.

`ModAssert::InfoUser` has the following methods:

- `void AddGranularity(Granularity granularity);`

  This specifies that the object is meant for more than one item of the granularity, so it will use info of that granularity every time an assertion or check fails.

- `void RemoveGranularity(Granularity granularity);`

  This specifies that the object is not meant for more than one item of the granularity, so it will no longer use info of that granularity every time an assertion or check fails.

- `HasGranularity(Granularity granularity);`

  This tells whether the object is meant for more than one item of the granularity.

- ```
  void UseInfo(const InfoProvider & ip,
                When when);
  ```

  This specifies when the given infoprovider should, regardless of granularities. `When` can be `Never`, `Once` or `Always`. If it is `Never`, the infoprovider is never used by this object. If it is `Once`, it is only used when once, e.g. when a logger is added to ModAssert. If it is `Always`, it is always used, e.g. also for every assertion and check that is reported, and at the end. You usually don't need this method.

- ```
  bool ShouldUseInfo(const InfoProvider & ip,
                     When when);
  ```

  Returns whether this InfoUser object should use the info of the InfoProvider when logging of the given type is done. This takes into account what the InfoProvider object specified in its constructor, and whether it was given to `ShouldUseInfo`. `when` should be `Once` or `Always`. If it returns `true` when `when` is `Always`, it will also return `true` when `when` is `Once`.

The class LogType

The class `LogType` is used to specify what kind of logging is done. It has one member `type`, an enum defined inside `LogType`, that can have these values:

- `AtBegin`: used when a logger is added to ModAssert
- `AtEnd`: used when a logger is removed from ModAssert
- `InBetween`: not yet used

A note on the lifetime of objects given to ModAssert

ModAssert never copies objects that you pass to it. It neither takes ownership of them. This refers to objects of classes that are derived from `Logger`, `Hook`, `Filter` and `InfoProvider`. This was done to prevent the reporting of memory leaks.

Therefore it is best to create these objects in the main function, or make them global in an implementation file, so they are created on the stack and live until the end of the application. However, you should always give them to ModAssert after the main method is entered, otherwise the arrays to store them in may not yet be initialized, and the thread library of your platform may not yet be operable at that time. So don't use selfregistering objects to give objects to ModAssert.

Exceptions thrown by loggers and hooks

If an assertion is handled, the hooks are called first, then the loggers and then the responder. No exceptions are caught by ModAssert, so if a logger throws an exception, the loggers after it and the responder will not be called. Likewise, if a hook throws an exception, the hooks after it, the loggers and the responder will not be called. If the responder throws an exception, nothing is affected. In any case ModAssert will stay in a valid state.

The StreamLogger objects

An implementation of the interface `ModAssert::Logger` is already provided, the `ModAssert::StreamLogger` class. Its constructor takes a stream as an argument, and will write the debugging information to that stream if an assertion or check is reported. By default the streams are of the type `std::ostream`.

If `RICHBOOL_USE_WX` is defined, the stream can be of type `wxTextOutputStream` or `wxOutputStream`, but in that case it's actually better to use the logger class `ModAssertWxLog`, that is in the files `Wx/modassert/ModAssertWxLog.h` and `Wx/modassert/ModAssertWxLog.cpp`. This class uses `wxLogDebug` to output the information, so you can redirect this output to wherever you like using the methods that wxWidgets provides for that.

The AppendToFileLogger objects

Another implementation of the interface `ModAssert::Logger` is provided, the `ModAssert::AppendToFileLogger` class. Its constructor takes a filename as an argument, and will write the debugging information to that file if an assertion or check is reported. Every time an assertion or check is logged, the file is opened in append mode, the information is logged, and the file is closed again.

Note that this is not the same as passing a filestream to a `StreamLogger` object, because then the file is left open as long as the stream object exists (or you close the file yourself).

The class `ModAssert::GroupList`

`ModAssert::GroupList` represents the assertion groups that were passed in the macro (`ModAssert::Error` if none is given), e.g. `ModAssert::Error, modAssertGroup1`, `ModAssert::Error && modAssertGroup1` (if `modAssertGroup1` is of the class `ModAssert::Group<ModAssert::ReportFailure>`). `ModAssert::GroupList` has the following four methods of interest:

- `size_t GetSize()`: returns the number of embedded types; e.g. 1 for `ModAssert::Error`, 2 for `ModAssert::Error && modAssertGroup1` if `modAssertGroup1` is of type `ModAssert::Group<ModAssert::ReportFailure>`, 2 for `modAssertGroup1` (because `ModAssert::Error` is added implicitly)
- `const std::string& GetName(size_t idx)`: returns the name of the embedded type with index `idx`
- `int GetLevel()`: returns the level of the groups. Certain loggers might want to ignore assertions of certain levels.
- `bool Has(const char *groupName)`: returns whether a group with the given name is present in the group. Certain loggers might want to exclude or include certain groups.

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

It has other methods to determine if assertions of the type should be logged or displayed, and also for the embedded types. But these are not of interest here, because `ModAssert::Responder::OnAssert` and `ModAssert::Logger::OnAssert` will only be called if `GetDisplay()` and `GetLog()` return true respectively.

It has also other methods to stop displaying and/or logging for embedded types, but this is better done with the return value of `ModAssert::Responder::OnAssert`.

The class `ModAssert::ParameterList`

Objects of the class `ModAssert::ParameterList` hold the parameters that are given in assertion macros with a suffix `P`. They hold these in a linked list. Each node holds an expression and its value, or a message if it was a string literal. The class can distinguish between string literals and other expressions that return a `const char *` by parsing the stringized macro argument (this uses a simple, fast algorithm that passes once through the string and supposes that the string represents a correct set of streamed expressions - otherwise the code wouldn't have compiled).

Objects of this type can be streamed to a stream. If you need to display them in some other way (e.g. in a GUI), you can access the first data by its method `GetMessage()`, and the following ones with the method `GetNext()`. These are of the type `ModAssert::ParameterList::Message`. It has a method `virtual MessageType GetType() const`, that returns `ModAssert::ParameterList::eMessage` if it is a `ModAssert::ParameterList::Message`, and `ModAssert::ParameterList::eValue` if it is a `ModAssert::ParameterList::Value`, that is derived from `ModAssert::ParameterList::Message`. `ModAssert::ParameterList::Message` has

also a method `const ModAssert::SubString& GetMessage()`, which holds a substring of the stringized macro argument. `ModAssert::SubString` can be converted to `std::string`, and has the methods `const char* begin()`, `const char* end()` and `size_t size()`. `ModAssert::ParameterList::Value` has a method `const std::string& GetValue()` which returns the value of the expression as a string; the expression itself is obtained with `GetMessage()` of `ModAssert::ParameterList::Message`.

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

So code to process a parameterlist typically looks like

```
for (ModAssert::ParameterList::Message *node=pParameterList->GetFirst();
 node; node = node->GetNext())
{
 std::string str = node->GetMessage();
 if (node->GetType()==ModAssert::ParameterList::eMessage)
 {
     ...
 }
 else if (node->GetType()==ModAssert::ParameterList::eValue)
 {
  ModAssert::ParameterList::Value *nodeValue =
   (ModAssert::ParameterList::Value*)node;
  const std::string &value = nodeValue->GetValue();
  ...
 }
}
```

Redefining existing assertion macros

There are headerfiles in ModAssert that redefine other assertion macros to ModAssert assertions. This way you just have to include that headerfile instead of the one that has the original definitions to use ModAssert. If you can't prevent including of the headerfile with the original definitions(e.g. because it is already included through another headerfile), just make sure that you include the ModAssert headerfile after the original one.

assertd.hpp

The headerfile `modassert/assertd.hpp` redefines the macros `ASSERT` and `VERIFY` to `MOD_ASSERT` and `MOD_VERIFY` respectively.

verifyv.hpp

The headerfile `modassert/verifyv.hpp` redefines the macros `ASSERT` and `VERIFY` to `MOD_ASSERT` and `MOD_VERIFY_V` respectively.

wxassert.hpp

The headerfile `modassert/wxassert.hpp` redefines the macros `wxASSERT`, `wxASSERT_MSG`, `wxFAIL`, `wxFAIL_MSG` , `wxCHECK`, `wxCHECK_MSG`, `wxCHECK_RET`, `wxCHECK2` and `wxCHECK2_MSG` to ModAssert equivalents.

Thread safety

The ModAssert package is thread safe, if one of these three multithreading packages is available: Win32 threads, wxWidgets threads, or pThreads. pThreads is used if the symbol MODASSERT_USE_PTHREADS is defined when ModAssert is built. If it is not defined, but RICHBOOL_USE_WX is defined when ModAssert is built, wxWidgets threads is used. If that is neither the case, but the symbol _WIN32 (which is defined by default with most Windows compilers) is defined when ModAssert is built, Win32 threads is used. Otherwise, ModAssert is not threadsafe.

Note: if you use pthreads, MODASSERT_USE_PTHREADS only has to be defined when the ModAssert library is built. It has no use to define it when you build an application that uses ModAssert.

pthreads should be used on Linux with gcc, and with Cygwin gcc on Windows, but not for other Windows compilers that were tested.

Thread safety in ModAssert means that only one thread at a time can display and log an assertion, and let the responder change parameters. If you want to change parameters that ModAssert uses from your own code, and want to do so in a thread safe way, use the class `ModAssert::ScopedLock` in `modassert/threads.hpp` to automatically lock:

```
#include <modassert/threads.hpp>
...
{
    ModAssert::ScopedLock lock;
    ModAssert::Warning.SetLog(false);

} // lock is automatically released
```

This can also be used to temporarily prevent other threads from displaying and logging assertions, e.g. if you need to log to a stream that ModAssert also logs to.

`ModAssert::ScopedLock` has a constructor that takes a boolean as an argument, for which the default value is `true`; if that argument is `true`, the lock is locked in the constructor. The destructor unlocks the lock. It also has methods `Lock()` and `Unlock()` to lock and unlock the lock at any time. The method `bool IsLocked()` tells whether it is locked.

A well known advice in multithreaded programming is to not call virtual functions from inside a critical section, because a derived class beyond your call might enter a critical section, and cause a deadlock. Alas, ModAssert calls virtual functions from its critical section when it calls the hooks, the loggers and the responder. So it's best not to enter a critical section in a hook, logger or responder. If you have to, make sure you don't enter the critical section elsewhere in your application (a critical section that can only be entered in your derived class should be safe, at least if you don't use the hook, logger or responder directly).

On some platforms there are no recursive locks. If this is the case, a thread that wants to lock ModAssert while it already has the lock, will deadlock. This can happen if you acquire the lock yourself, and acquire it yourself a second time, as in the following example:

```
#include <modassert/threads.hpp>
```

```
void foo()
{
 ModAssert::ScopedLock lock;
 ModAssert::Warning.SetLog(false);
}

void bar()
{
 ModAssert::ScopedLock lock;
 foo(); // will cause a deadlock on some platforms!
}
```

or if you acquire the lock yourself and a modular assertion or check is reported, as in the following example:

```
#include <modassert/threads.hpp>

void foo()
{
 int a=0;
 ModAssert::ScopedLock lock;
 MOD_ASSERT(rbEQUAL(a, 1)); // will cause a deadlock on some platforms!
}
```

See also the section called “When ModAssert is active” for further considerations about thread safety in ModAssert.

Considerations related to macros

Some programmers avoid macros because of the problems that are associated with them. However, the macros in the ModAssert package are constructed with great care, to avoid these dangers.

- The arguments of the macros are evaluated at most once, except for the groups argument; however, using groups as an argument is very unlikely to have side effects. Note that `MOD_ASSERT` and variations of it are removed entirely if reporting of assertions is disabled at compile time, so the arguments are not evaluated. Consider using `MOD_VERIFY` or a variation of it if the condition has necessary side effects, and you can't easily move the side effect out of it. The condition of `MOD_CHECK` is always evaluated. Rich Booleans also evaluate their arguments only once. Therefore constructions like `MOD_VERIFY_GO(group1, ++c, "inc c", rbEQUAL(a++, b++))` are safe and do what you would expect, but of course `++c` is only evaluated if the user of the application chooses to. Also note that the parameters and the text of the optional action are only evaluated if the assertion or check is reported, so you should not write code in these arguments that has side effects. Of course the optional action is only evaluated if the user wants this to happen. The level or group argument can be evaluated up to three times, but this is usually a constant expression with no side effects.
- 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 ModAssert macros are wrapped in `if { ... } else 0` (a variation of `do { ... } while (0)`), so they can be used safely in a if-else construct without surrounding bracelets

Security

Please remember that with security sensitive data, you should not use Rich Booleans in assertions and checks. In these cases it is better to use a plain boolean condition. You should neither use such data in the parameterlist.

```
// don't do this!
MOD_CHECK(rbEQUAL(password, str), return false);

// don't do this!
MOD_CHECK_P(str, calc_md5(str)==md5, return false);

// safe:
MOD_CHECK(calc_md5(str)==md5, return false);
```

Warning levels

Where possible, ModAssert is 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 Visual C++.NET 2003 and 2005 on warning level 4, there is a warning about a constant condition in an if-statement for every assertion and check. This can be prevented by disabling warning number 4127.
- 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, ModAssert doesn'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.

Extending the context (advanced)

When an assertion or check is reported, the macro `MOD_ASSERT_CONTEXT` is called to construct an object of the type `ModAssert::Context`. This contains the filename and the linenumber where the assertion occurred, and if your compiler supports it, the name of the function where it occurred.

You can customize this mechanism by adding extra information. To do this, derive a class from `ModAssert::Context`, and redefine the macro `MOD_ASSERT_CONTEXT` to construct an object of your class with the necessary arguments. Override the method `streamout`, to have the information in your class displayed, and call `streamout` of the parent class.

The constructor of `ModAssert::Context` has three parameters: the filename, the linenumber and the functionname. Your derived class should at least have these three as constructor arguments and pass it to the constructor of `ModAssert::Context`, and your redefinition of the macro should give `__FILE__`, `__LINE__` and `MOD_ASSERT_FUNCTION` as values for these arguments of the constructor.

Then include the headerfile where you define your derived class and your redefinition of the macro, after the headerfile `modassert/assert.hpp`.

Note: the ModAssert library itself doesn't need to be rebuilt with this redefined macro `MOD_ASSERT_CONTEXT`. It uses references to the base class `ModAssert::Context`, and the extra information is obtained through the virtual method `streamout`.

This method was introduced to e.g. add the thread id, the time, a backtrace, or the state of your application. However, this can also be done with an `InfoProvider` derived object (see the section called “Providing extra information”), which is easier, since you don't need to add an include file to every source file where you want it. If some information is only interesting in certain sourcefiles, you should use default parameters.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W