Boost.Assert

Peter Dimov

The Boost.Assert library provides several configurable diagnostic macros similar in behavior and purpose to the standard macro `assert` from `<cassert>`.

Assertion Macros, <boost/assert.hpp>

BOOST_ASSERT

The header `<boost/assert.hpp>` defines the macro `BOOST_ASSERT`, which is similar to the standard `assert` macro defined in `<cassert>`. The macro is intended to be used in both Boost libraries and user code.

- By default, `BOOST_ASSERT(expr)` expands to `assert(expr)`.
- If the macro `BOOST_DISABLE_ASSERTS` is defined when `<boost/assert.hpp>` is included, `BOOST_ASSERT(expr)` expands to `((void)0)`, regardless of whether the macro `NDEBUG` is defined. This allows users to selectively disable `BOOST_ASSERT` without affecting the definition of the standard `assert`.
- If the macro `BOOST_ENABLE_ASSERT_HANDLER` is defined when `<boost/assert.hpp>` is included, `BOOST_ASSERT(expr)` expands to

  ```
  (BOOST_LIKELY(!!(expr))? ((void)0): ::boost::assertion_failed(#expr,
      BOOST_CURRENT_FUNCTION, __FILE__, __LINE__))
  ```

  That is, it evaluates `expr` and if it's false, calls `::boost::assertion_failed(#expr, BOOST_CURRENT_FUNCTION, __FILE__, __LINE__)`. This is true regardless of whether `NDEBUG` is defined.

  `boost::assertion_failed` is declared in `<boost/assert.hpp>` as

  ```
  namespace boost
  {
      void assertion_failed(char const * expr, char const * function,
          char const * file, long line);
  }
  ```

  but it is never defined. The user is expected to supply an appropriate definition.
- If the macro `BOOST_ENABLE_ASSERT_DEBUG_HANDLER` is defined when `<boost/assert.hpp>` is included, `BOOST_ASSERT(expr)` expands to `((void)0)` when `NDEBUG` is defined. Otherwise the behavior is as if `BOOST_ENABLE_ASSERT_HANDLER` has been defined.

As is the case with `<cassert>`, `<boost/assert.hpp>` can be included multiple times in a single translation unit. `BOOST_ASSERT` will be redefined each time as specified above.

BOOST_ASSERT_MSG

The macro `BOOST_ASSERT_MSG` is similar to `BOOST_ASSERT`, but it takes an additional argument, a character literal, supplying an error message.

- By default, `BOOST_ASSERT_MSG(expr,msg)` expands to `assert((expr)&&(msg))`.
- If the macro `BOOST_DISABLE_ASSERTS` is defined when `<boost/assert.hpp>` is included, `BOOST_ASSERT_MSG(expr,msg)` expands to `((void)0)`, regardless of whether the macro `NDEBUG` is defined.
- If the macro `BOOST_ENABLE_ASSERT_HANDLER` is defined when `<boost/assert.hpp>` is included, `BOOST_ASSERT_MSG(expr,msg)` expands to

  ```
  (BOOST_LIKELY(!!(expr))? ((void)0): ::boost::assertion_failed_msg(#expr,
      msg, BOOST_CURRENT_FUNCTION, __FILE__, __LINE__))
  ```

  This is true regardless of whether `NDEBUG` is defined.

  `boost::assertion_failed_msg` is declared in `<boost/assert.hpp>` as

  ```
  namespace boost
  {
      void assertion_failed_msg(char const * expr, char const * msg,
          char const * function, char const * file, long line);
  }
  ```

  but it is never defined. The user is expected to supply an appropriate definition.

- If the macro `BOOST_ENABLE_ASSERT_DEBUG_HANDLER` is defined when `<boost/assert.hpp>` is included, `BOOST_ASSERT_MSG(expr)` expands to `((void)0)` when `NDEBUG` is defined. Otherwise the behavior is as if `BOOST_ENABLE_ASSERT_HANDLER` has been defined.

As is the case with `<cassert>`, `<boost/assert.hpp>` can be included multiple times in a single translation unit. `BOOST_ASSERT_MSG` will be redefined each time as specified above.

BOOST_VERIFY

The macro `BOOST_VERIFY` has the same behavior as `BOOST_ASSERT`, except that the expression that is passed to `BOOST_VERIFY` is always evaluated. This is useful when the asserted expression has desirable side effects; it can also help suppress warnings about unused variables when the only use of the variable is inside an assertion.

- If the macro `BOOST_DISABLE_ASSERTS` is defined when `<boost/assert.hpp>` is included, `BOOST_VERIFY(expr)` expands to `((void)(expr))`.
- If the macro `BOOST_ENABLE_ASSERT_HANDLER` is defined when `<boost/assert.hpp>` is included, `BOOST_VERIFY(expr)` expands to `BOOST_ASSERT(expr)`.

- Otherwise, `BOOST_VERIFY(expr)` expands to `((void)(expr))` when `NDEBUG` is defined, to `BOOST_ASSERT(expr)` when it's not.

BOOST_VERIFY_MSG

The macro `BOOST_VERIFY_MSG` is similar to `BOOST_VERIFY`, with an additional parameter, an error message.

- If the macro `BOOST_DISABLE_ASSERTS` is defined when `<boost/assert.hpp>` is included, `BOOST_VERIFY_MSG(expr,msg)` expands to `((void)(expr))`.
- If the macro `BOOST_ENABLE_ASSERT_HANDLER` is defined when `<boost/assert.hpp>` is included, `BOOST_VERIFY_MSG(expr,msg)` expands to `BOOST_ASSERT_MSG(expr,msg)`.
- Otherwise, `BOOST_VERIFY_MSG(expr,msg)` expands to `((void)(expr))` when `NDEBUG` is defined, to `BOOST_ASSERT_MSG(expr,msg)` when it's not.

BOOST_ASSERT_IS_VOID

The macro `BOOST_ASSERT_IS_VOID` is defined when `BOOST_ASSERT` and `BOOST_ASSERT_MSG` are expanded to `((void)0)`. Its purpose is to avoid compiling and potentially running code that is only intended to prepare data to be used in the assertion.

```
void MyContainer::erase(iterator i)
{
// Some sanity checks, data must be ordered
#ifndef BOOST_ASSERT_IS_VOID

    if(i != c.begin()) {
        iterator prev = i;
        --prev;
        BOOST_ASSERT(*prev < *i);
    }
    else if(i != c.end()) {
        iterator next = i;
        ++next;
        BOOST_ASSERT(*i < *next);
    }

#endif

    this->erase_impl(i);
}
```

- By default, `BOOST_ASSERT_IS_VOID` is defined if `NDEBUG` is defined.
- If the macro `BOOST_DISABLE_ASSERTS` is defined, `BOOST_ASSERT_IS_VOID` is always defined.
- If the macro `BOOST_ENABLE_ASSERT_HANDLER` is defined, `BOOST_ASSERT_IS_VOID` is never defined.
- If the macro `BOOST_ENABLE_ASSERT_DEBUG_HANDLER` is defined, then `BOOST_ASSERT_IS_VOID` is defined

when `NDEBUG` is defined.

Current Function Macro, <boost/current_function.hpp>

BOOST_CURRENT_FUNCTION

The header `<boost/current_function.hpp>` defines a single macro, `BOOST_CURRENT_FUNCTION`, similar to the C99 predefined identifier `__func__`.

`BOOST_CURRENT_FUNCTION` expands to a string literal containing the (fully qualified, if possible) name of the enclosing function. If there is no enclosing function, the behavior is unspecified.

Some compilers do not provide a way to obtain the name of the current enclosing function. On such compilers, or when the macro `BOOST_DISABLE_CURRENT_FUNCTION` is defined, `BOOST_CURRENT_FUNCTION` expands to `"(unknown)"`.

`BOOST_DISABLE_CURRENT_FUNCTION` addresses a use case in which the programmer wishes to eliminate the string literals produced by `BOOST_CURRENT_FUNCTION` from the final executable for security reasons.

Appendix A: Copyright and License