Document number: N4154
Date: 2014–09–30
Reply to: David Krauss
(david_work at me dot com)

Operator `assert`

1. Abstract

The `assert` macro has never behaved much like a real function, and for the foreseeable future, it will look and smell like an operator. The way the macro is specified by C precludes it from providing optimization hints in production mode, but allows it to execute arbitrary side effects in debug mode. Adding `assert` as a keyword and built-in operator would have benefits but essentially no downside.

This will resolve LWG DR 2234 and 2413.

2. Proposal

Current macro-based implementations should remain nearly compliant with minimal adjustment, modulo behavior when no standard library headers are included and `#ifdef assert`, which are inconsequential, and tolerance of expressions with commas not surrounded by parentheses. The built-in operator should still discriminate on `NDEBUG`, which is examined each time `<assert.h>` is included. Instead of generating alternative definitions of the `assert` macro, it may invoke a `#pragma` to change the meaning of subsequent `assert` expressions.

Macro preprocessing tends to be more tolerant than any parser-based facility, but attributes set a useful precedent for somewhat-meaningful text in the *balanced-token-seq* production (N3969 §7.6.1 [dcl.attr.grammar]). This simply requires that parentheses, braces, and brackets balance. The preprocessor already requires balanced parentheses, and rejection of unbalanced braces and brackets should be an acceptable level of legacy breakage.

To satisfy as many users as possible, four levels of assertion are provided:

- Default: `assert` evaluates its condition and generates a diagnostic upon failure.
- `NDEBUG = strong`: `assert` has no side effects, but the implementation may use the condition, and if it would fail, the behavior is undefined. This provides optimal hints.
- `NDEBUG = strict`: The `assert` expression is fully parsed and semantically checked, but no evaluation occurs. The behavior is still defined even if it would evaluate as false, but this may be considered unlikely.
- `NDEBUG` defined as empty or an integer literal: The `assert` operands are syntactically a *balanced-token-seq*. Otherwise this is the same as `strict` mode.
- Other identifiers in the expansion of `NDEBUG` are reserved to the standard for future expansion, except for identifiers usually reserved to the library.

Some users prefer to define their own `assert` macro instead. This usage is not blessed by the C++ standard, if any standard library header is included. Following this proposal, custom macro definitions will continue to work as usual, along with any possible obnoxious effects to third-party code using the facility, under the guise of undefined behavior.

2.1. Semantics

By default, `assert` accepts a contextually convertible Boolean expression and an optional second operand, much as `static_assert` does. The condition is evaluated, and if it is true, there is no further effect. If the condition is false, the program produces a diagnostic and calls `abort()`. If the second operand is absent, the implementation prints the usual default message to `stderr`. If it is a string literal, that is printed to `stderr` instead. Otherwise, it is simply evaluated to produce the diagnostic.

postfix-expression:

`assert (` *assignment-expression* `)`

`assert (` *assignment-expression* `,` *string-literal* `)`

`assert (` *assignment-expression* `,` *assignment-expression* `)`

If `<assert.h>` observes `NDEBUG` to be defined to the tokens `strict` or `strong`, this grammar is also used. These tokens need to be reserved as macro names, which is also true of any identifier used in the standard library as e.g. an enumerator or class member name, so lowercase letters are appropriate. (As it happens, we already have `pointer_safety::strict`.)

For `strict`, both operands are unevaluated, for the sake of diagnosing ill-formed expressions. For `strong`, the implementation may treat any subexpression of the first operand as a constant expression, and furthermore assume that the entire first operand evaluates to `false`. (If the first operand would evaluate to `true`, the behavior is undefined.) It cannot access any objects, but values that happen to be in the local context are fair game.

When `NDEBUG` is defined such as it typically is, expanding to nothing or to an integer literal, `<assert.h>` sets `assert` to accept and ignore a *balanced-token-seq*.

`assert (` *balanced-token-seq* `)`

In this case as with `strict`, the implementation may attempt to extract some information from the operands, but it cannot assume that a false condition represents failure, because that is not how disabled assertions traditionally work.

These expressions have type `void`. An `assert` expression is not a constant expression if its first operand is not a constant expression (inclusive of the *balanced-token-seq* case). Otherwise, it is a constant expression if in `strict` mode or if the expression evaluates to true. This means that `assert` conditions in constant expression contexts or in `constexpr` function evaluation are always evaluated, but the result is discarded in `strict` mode. `assert(false)` is significant in `strong` mode because it produces undefined behavior, which renders an expression not constant.

2.2. Implementation

Here is what `assert.h` might look like.

```
#ifdef __cplusplus
#   define _ASSERT_MODE_strict 1
#   define _ASSERT_MODE_strong 2
#   define _CAT_LIT(A, B) A ## B
#   define _CAT_ASSERT_MODE(B) _CAT_LIT(_ASSERT_MODE_, B)
#   define _ASSERT_MODE _CAT_ASSERT_MODE( NDEBUG )

#   ifndef NDEBUG
#       pragma IMPL assert_mode debug
#   elif _ASSERT_MODE == _ASSERT_MODE_strong
#       pragma IMPL assert_mode strong
#   elif _ASSERT_MODE == _ASSERT_MODE_strict
#       pragma IMPL assert_mode strict
#   else
#       pragma IMPL assert_mode relaxed
#   endif
#else
// Not C++, handle C
```

2.3. Legacy support

The optional second operand can be implemented by a library for minimal, nonconforming functionality, without `strict` and `strong` modes. If an `assert` macro is invoked with only one operand, it can be mapped to a distinct expansion which adds the default string. If it invoked with two operands, the second operand can be used as an argument to an overloaded function. String literals can be reasonably discriminated by a template parameter of type `char const (&)[N]` for an integer-type template parameter `N`. This may not be good QOI, but it can serve as a lifeline to users who wish to apply proprietary `assert` macros to third-party or otherwise modern code.

Development environments may encourage users to migrate to `strict` or `strong` mode, but it could be inconvenient or dangerous to change the production-mode `NDEBUG` value quietly. Users should be informed of the implications of `strong` mode and allowed to opt in. Variables declared conditionally upon `#ifndef NDEBUG` and used in `assert` conditions will produce ill-formed expressions in `strict` mode, and this is not an uncommon construct.

3. Future work

This proposal has not yet been prototyped, and a formal specification needs to be drafted.

Assertions are part of the greater paradigm of precondition and postcondition specification, which relates to contract programming. The intent is to complement future work in this area, but no particular considerations have been made in this regard.