ISO/IEC JTC 1/SC 22/WG14
WG 21, SG 22

N2757 v0
P2390R0

June 7, 2021

Add annotations for unreachable control flow

proposal for addition to C23 and TS 6010

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

We propose features **unreachable**, **attest** and **testify** to specify paths in the control flow of a program that will never be reached. The aim is to provide means for the user to express guarantees about the effective control flow that will be executed by a program. Compilers may then apply aggressive optimizations that otherwise would not be possibly or that would rely on the detection of undefined behavior for certain input combinations. Additionally, in certain cases the **testify** macro allows to integrate the testing of runtime conditions into the interfaces of functions, and therefore helps to provide verification and debugging facilities for third party libraries for which source code is not available.

1. INTRODUCTION

Unfortunately, the C standard leaves handling of the detection of undefined behavior (UB) quite open. This often leads to a lot of misunderstandings and open-ended debates what assumptions about conditions leading to undefined behaviors (for example bounded, unbounded, race conditions, overflow) may be used in optimizations without alerting the programmer.

This status quo about the handling of undefined defined behavior becomes worrisome when we want to change the state of certain conditions at the margins. Currently the Memory Model Study Group (MMSG) of WG14 tries to establish a reformed model for access to storage. It has already proposed a model for provenance that has been accepted by WG14 in an upcoming technical specification TS 6010, see N2676, and is currently discussing a model for an internally consistent treatment of the access to objects that are not initialized (or only partially initialized) or that have unspecified byte representations (e.g padding).

It seems, that in some cases there is an implicit assumption that code that makes a potentially undefined access, for example, does so willingly; the fact that such an access is unprotected is interpreted as an assertion that the code will never be used in a way that makes that undefined access. Where such an assumption may be correct for highly specialized code written by top tier programmers that know their undefined behavior, we are convinced that the large majority of such cases are just plain bugs.

Our current paper for MMSG that gives a first model for uninitialized objects and unspecified values, see N2756, is therefore based, among others, on the following principle.

PRINCIPLE 3 (IMPLICIT REACHABILITY). *A path in the control flow that can only be reached if a preceding operation has undefined behavior shall not be skipped unless the program explicitly tags it as unreachable.*

Obviously, this principle is only a guideline and it would be difficult (semantically and socially) to formulate normatively. Instead, we propose to progress pragmatically and provide tools to express exactly and explicitly when, if and where an assumption of unreachability should be taken for granted by the compiler.

Currently the only tools to indicate non-reachability are

— undefined operations, such as a division by zero,
— a call to a **noreturn** function such as **abort** or **exit**,
— or extensions, such as platform specific attributes or builtins, or explicitly issuing undefined operations.

In the following we propose features that operate similar to a function **unreachable** as it already has been proposed to C++, see https://wg21.link/p0627.

```
namespace std {
  [[noreturn]] void unreachable(char const* message);
  [[noreturn]] void unreachable();
}
```

If a call to this pseudo-function is hit during execution, the behavior is undefined, and the intent is that the compiler may optimize the code aggressively under the assumption that this will not happen.
Observe that in C there is currently no tool that is designed explicitly to specify general input restrictions for function interfaces besides array sizes. Only with some stretch such conditions can be specified for array parameters by integrating them into the size expression.

```
int print(int argc, char* argv[static
                               /* test a necessary condition */
                              ((argc > 2 ? assert(argc > 2) : abort()),
                               argc + 1)
                             ]);
```

Such an expression is necessarily executed in the context of the called function and it is not immediately clear if it is possible to intercept it on the calling site and to completely avoid a call, if the condition is not met.
There is currently a C feature that *seems* to fulfill a similar task as **unreachable**, namely the **assert** macro. A call of that macro with an expression E as argument indeed indicates that E is expected to hold. In "development" mode, if E doesn't hold a runtime diagnostic is provided and the execution is aborted. So in this mode, after such a call to **assert** a compiler can assume that the code that is following is unreachable under $\neg E$ (because of the call to **abort**) and may optimize agressively. The possible definition of the macro **NDEBUG** is then intended to indicate a "production" compilation mode that removes the test for E and that unconditionally executes the code that comes after. But that specification is counter productive, because now the compiler may no longer assume that E holds and optimization opportunities that were present in development mode are removed.

```
1  int f(int const*const p) {
2    assert(p);
3    ...
4    if (p) return printf("value is %d\n", *p);
5    puts("invalid pointer value");
6    return 0;
7  }
```

Here in development mode the return can in fact be realized by the compiler into a tail call to **printf** that never returns to `f`. As soon as **NDEBUG** is defined, the picture changes completely. Instead of being tested at the beginning of the function, `p` is then tested at the end and both calls to **printf** and to **puts** have to be generated.
This behavior does not only systematically miss optimization opportunities, it also has the disadvantage that the behavior and generated code can be substantially different between development mode and production mode, and so debugging can be a real challenge.

2. DESIGN CHOICES

2.1. Naming

We have made some searches to see if the identifiers that we propose for the new features are already claimed in other parts of the community. This is obviously the case for **unreachable** which is a soon-to-be-integrated feature for C++. We want the two features be equivalent, so a reuse of this identifier is intentional.

In a first version we used the identifier `assume` for what is here **`attest`**. This was not well received because there is an ongoing proposal for so-called contracts in C++ that would use that identifier for a similar, but not equivalent feature.
Identifiers **`attest`** and **`testify`** seem to be rarely used and the probability of conflict with existing code should be low.[1] For these choices in particular, we hope that **`attest`** is putting the emphasis on the fact that the programmer is is responsibly for the assertion of the condition, and that **`testify`** conveys the message that an action can be taken by the implementation if the assertion does not hold.

2.2. Proposed features

For the first part we propose an addition to C that is similar to a function **`unreachable`** as it has been proposed to C++, see https://wg21.link/p0627. This proposal is based on a widely implemented practice for extensions in C and C++ namely a pseudo-function `__builtin_unreachable` and similar to the Rust feature `unreachable_unchecked`. Function calls with **`unreachable`** simply mark the whole branch of control flow in which they appear as unreachable under all circumstances that the code will ever be executed, and so the compiler may do any aggressive optimization they see fit.
Conceptually, a marker like **`unreachable`** might be hidden relatively deeply inside some sophisticated control flow and may thus not provide an easy overview of the prerequisites that a particular function has. Therefore we propose a second feature, called **`attest`**, that tests a scalar expression that is passed as an argument. It is based on a similar existing extension, `__assume`, of the MSVC compiler.
The difference between the two is really only on the surface, there is no profound conceptual difference. Both features are easily translated into each other.

```
#define attest(...)       ((__VA_ARGS__) ? (void)0 : unreachable())
#define unreachable(...) attest(false)
```

Only the emphasis for the programmer is a bit different, so we think the C standard should just provide both of them.
We also propose a third feature, **`testify`**, with the intent to fill the gap that is left by **`assert`**. It is in fact similar to **`assert`**, only that if the assertion fails in production mode the call it is not replaced by a `nop`, but by a call to **`unreachable`**. Thereby it ensures that the optimization opportunities still are the same as in development mode, and that the produced binary in both modes behaves similar if E is fulfilled. Again the feature itself is not very different from the previous two, for example an implementation could be as follows.

```
#undef testify
#ifdef NDEBUG
# define testify      attest
#else
# define testify      assert
#endif
```

A neat property of the **`testify`** feature is that in certain cases it allows to lift the test into the interface specification of a function. Due to that trick it is possible to advance the test (if in development mode) to the call side of such an annotated function. Even if the function itself is only available as a binary in a library, users of the function can then perform their own series of tests to see if the expected assumptions hold.

[1] The only usage of **`attest`** that we found is the "ESIM Simbolic Hardware Simulator" were it is protected by an **`ifndef`** and basically has the same role as the feature, here.

2.3. Specification method

There are different possible specifications for the proposed features:

— Add attributes.
— Provide a (or several) pseudo-function declaration.
— Provide an explicit or implicit definition of a macro.
— Add keywords.

Unfortunately attributes are too restrictive in this case because they would miss an important use case, namely that a "call" to the feature could be used in an expression. For example, we think that use as in the following would be important to note that a loop variable is not expected to overflow or wrap:

```
for (size_t i = some; stop_condition(i); testify(i < SIZE_MAX), ++i) {
   ...
}
```

Attributes are not allowed on expressions. The only possibility would be to introduce the new features as attributes for null statements, which would exclude the case above.
So the best way to convey the information is to add features to the language that behave like function calls. These must not necessarily be conventional functions. In the contrary conventional functions have the disadvantage that they constrain the optimizations, because specifying that a particular function call is undefined is different than to provide a function that then observes undefined behavior when called with certain arguments.
Similar to the existing **assert** macro, our choice has been to reclaim a behavior that is syntactically deduced from a function call, but without providing a pseudo-function interface. It makes no sense to have linker symbols with these functions (they have undefined behavior) and forcing implementations to provide linker symbols would add a possible incompatibility with legacy code.
To impede the least possible on existing code the next choice has been to provide the features via a header. This would have been useless if the features where specified with new keywords, so that possibility was also excluded. For the choice of the header itself `<assert.h>` seemed the most natural because it already provides **assert** with similar properties.

2.4. Function interface specification

As noted above, the prospected C++ feature **unreachable** currently is specified by explicitly providing function prototypes that include a `[[noreturn]]` attribute. We think that this is problematic for several reasons.
First, for C, there is no such thing as a **namespace** declaration, and so the names that we use would interact badly with existing code that already uses the same identifiers. We mitigate that problem by insisting that **unreachable** and **attest** should be pseudo-functions that, as specified here, will never have external definitions that could interact with TU that don't use the features.
So existing source that uses the identifiers **unreachable** or **attest** for other purposes may generally deal with this addition easily:

— If they don't include `<assert.h>` they don't have to change anything.
— If they also include `assert`, they'd have to ensure that before those parts that redfine **unreachable** or **attest**, macro names are undefined with something similar to

```
#undef unreachable
#undef attest
```

and that before any use of **unreachable** or **attest** as specified here `<assert.h>` is newly included.

This should very much limit the possibility of conflicts with existing code.
The feature **testify** is a bit different. Because it is modeled along the lines of the existing **assert** macro, in particular with its behavior with respect to **NDEBUG**, we didn't see much of an alternative to define it as a macro, too. So all the conflict potential for adding a macro to a standard header applies.

2.5. Undefined behavior versus noreturn function declarations

An important property of the proposed feature is that code stumbling upon them under certain conditions is plain and simply undefined. We do not want to restrict implementations in any way how they are going to take advantage of that knowledge.
For that reason we do not provide a pseudo declaration for the features, but give a normative description about the syntactical context in which they may occur. In particular a specification as **noreturn** function (or **[[noreturn]]** for C++ as in https://wg21.link/p0627) is not very helpful, because one of the important possibilities of code generation is indeed to fall through to other branches that lexically follow the call. We think that a "regular" function declaration would leave the false impression to users that the execution would stop at such a point.
Another reason to avoid function declarations is the misinterpretation (as it has happened for the **assert** macro) that such a pseudo-declaration would restrict the number of parameters that a macro might expect. By specifying the feature on the level of the syntax in translation phase 7, it should be clearer that the number of exposed commas is not relevant for the validity of a call, but whether the argument can be parsed as one valid expression.

2.6. Improved function interfaces

Generally, as they are function calls, all three features (much as **assert**) are void expressions and so in C they can appear in the size expression of an array parameter. For example we could have a declaration such as the following

```
int print(int argc, char const* argv[static (TEST(argc > 2), argc + 1)]);
```

where TEST could be any of **assert**, **attest** or **testify**. As the current rules go, such a specification would usually be ignored by any callers of print but only executed in the context of the definition. Since **attest** does not depend on the setting of **NDEBUG** the result of the test would be the same if performed in the context of the caller or the callee, so there is nothing to add to our specification.
For **testify** (and **assert**) the situation is different. The macro **NDEBUG** could be set to different values in the TU of the caller than in that of the function definition. Therefore we allow (but do not force) the implementation to advance the test into the context of the caller. Then, a user of print may run the test to see if the assumptions hold for them without having to recompile print.
There are only two situations to consider. If the assumption holds, execution continues normally and the only possible additional work is that the test might be evaluated. If it doesn't hold, either a diagnostic is printed and the program is aborted (if **NDEBUG** is not defined) so the call to print is not even entered, or the behavior is undefined (if **NDEBUG** is defined) and no guarantee about the call can be made whatsoever.
Technically, we just impose that a function that has an evaluation of **testify** in the interface must not be called when the assumption doesn't hold. Thereby we lift the resulting unreachability condition to the call site of the function, and we leave it up to the implementation to react to a violated assumption or not.

2.7. Side effects of the test

Since we are aiming for a general usability of the features that test an expression E, we want to ensure that the evaluation of E does not by itself constrains the optimization opportunities promoted by our new features. Therefore we enforce that E has no observable side effects, and thus its determination can be moved or completely removed by constant propagation or similar optimizations.

Note also that the features **`attest`** and **`testify`** evaluate their argument expression unconditionally. For **`testify`** in particular, we thereby avoid warnings when compiling in production mode of variables that are used within E, which for **`assert`** can be a bit annoying. If E has no observable side effects, modern optimizers are generally quite capable to remove the evaluation of E completely from **`attest`**`(`E`)`, because if it is not fulfilled, the behavior is undefined, anyhow. So the important aspect of this function is really to provide useful information about possibly values to the compiler. The same argument shows that the evaluation of E, if side effect free, can also be avoided when **`testify`** is used in production mode.

3. IMPACT

Other than using the previously unreserved identifiers **`unreachable`**, **`attest`** and **`testify`**, the proposed additions have no impact on existing code. Since there is no addition of function symbols to the C library, that impact is limited to code that includes the header `<assert.h>`. The pseudo-functions **`unreachable`** and **`attest`** could even be redefined in block scope if *e.g* some legacy code has them. The macro **`testify`** needs slightly more care because of the nature of macros in general. If that would be considered too intrusive the features could be proposed with their own new header, but we don't think that this is necessary.

4. PROPOSED CHANGES

We formulate the three features merely independent from one another such that WG14 may freely chose any or all of them. Also, the proposed text already provides a marginal integration with the lambda feature that we expect to go into C23. If that would not be the case, the text can easily be adapted.

If added to C23, features as described in the following should be added to TS 6010 as well to note the intended difference from C17 and to be able to formulate recommended practice that uses **`unreachable`**, **`attest`** or **`testify`**.

CHANGE 1. *Add a new clause to the* `<assert.h>` *library clause.*

> **7.2.2 Control flow assertions**
>
> 1 Interfaces in this clause are declared as identifiers of functions and have all the syntactical properties of function names. This property not withstanding, no internal or external definition for these identifiers shall be provided by implementations. If the `<assert.h>` header is not included, no conflict with function scope features with the same name as an identifier provided here shall be created.

4.1. The `unreachable` function

CHANGE 2. *Add* **`unreachable`** *as a function to the list in* `7.2 p1`.

CHANGE 3. *Add a new sub-clause to the new clause* `7.2.2`.

> **7.2.2.1 The `unreachable` function**
>
> **Constraint**

1 The **unreachable** function shall only be used as the function designation of a function call. Such a call shall receive zero or one arguments; if the call receives an argument, that argument shall be a string literal.

2 A translation unit that includes the header <assert.h> shall not otherwise declare the identifier **unreachable** in a file scope declaration or as having linkage.

Description

3 A function call using the **unreachable** function indicates that the particular flow of control that leads to the call will never be taken. Such a call is an expression of type **void**. The program execution shall not reach such a call, and any function body shall have one valid control flow that does not reach such a call.

Returns

4 If a function call with **unreachable** as the function designation is reached during execution the behavior is undefined.

5 NOTE 1 The **unreachable** function provides optimization opportunities. Its intended use is to annotate certain path in the control flow that can only be reached when the behavior is already undefined, or when an unintended use of the function containing the evaluation leads to the point of execution of **unreachable**.

6 NOTE 2 The **unreachable** function interface is quite particular as it specifies a function for which the program (and not the implementation) has to ensure that a function call is never reached. Executables that are the result of the translation of such a program will in general not be able to cope with a violation of that assumption and the behavior under such circumstances will be erratic. For example, parts of the control flow that come logically after a branch that is tagged with **unreachable** may then be executed unintendedly. Programs that do not provide the necessary guarantees themselves but simply seek to ensure that a specific erroneous branch of control flow is not executed have other tools at their disposition (such as **strerror** or **abort**) that are better suited and that leave the execution in a defined state.

Recommended practice

7 It is recommended that a diagnostic is issued if all executions of a function body will necessarily evaluate a call to **unreachable**.

8 It is recommended that implementations that operate under a debug or test mode for translation issue a diagnostic message and terminate the execution similar to **assert** whenever a call to **unreachable** is reached. If additionally a string literal is provided as an argument to a function call, it is expected to contain comprehensive information that indicates the reason why the particular path is considered to be never reached; under the above circumstances it is recommended that the string literal is added to the diagnostic message.

9 EXAMPLE The following program assumes that each execution is provided with at least two command line arguments. The behavior of an execution with less arguments is undefined.

```
1 #include <stdio.h>
2 #include <assert.h>
```

```
3
4  int main(int argc, char* argv[]) {
5    if (argc <= 2) unreachable("two arguments needed");
6    else return printf("we see %s\n", argv[2]);
7    return puts("this should never be reached");
8  }
```

Here, the possible effect of using the **unreachable** function is that the resulting executable never performs the comparison and unconditionally executes a tail call to **printf** that never returns to the **main** function. In particular, the call to **puts** can be omitted from the executable. Unless translated in a debugging or test mode, no diagnostic is expected.

4.2. The `attest` function

CHANGE 4. *Add* **attest** *as a function to the list in* `7.2 p1`.

CHANGE 5. *Add a new sub-clause to the new clause* `7.2.2`.

7.2.2.2 The attest function

Constraint

1 The **attest** function shall only be used as the function designation of a function call **attest**(*E*) with an argument expression *E* and such that *E* has scalar type.

2 A translation unit that includes the header `<assert.h>` shall not otherwise declare the identifier **attest** in a file scope declaration or as having linkage.

Description

3 A function call **attest**(*E*) with scalar expression *E* indicates that it is expected that the control flow will never reach the call where *E* evaluates to zero or null. Such a call is an expression of type **void**. *E* shall be such that, other than computing the result, no discrepancy in the state of the abstract machine before and after its evaluation is observable.

Returns

4 The **attest** function returns no value. If a call to the **attest** function with an argument expression that evaluates to zero or null is executed the behavior is undefined.

5 NOTE 1 The **attest** function interface is quite particular as it specifies a function for which the program (and not the implementation) has to ensure that no function call is reached where the argument expression evaluates to zero or null. Executables that are the result of the translation of such a program will in general not be able to cope with a violation of that assumption and the behavior under such circumstances will be erratic. For example, statements that come logically after a call to **attest** may then be executed unintendedly. Programs that do not provide the necessary guarantees themselves but simply seek to ensure that a specific erroneous branch of control flow is not executed have other tools at their disposition (such as **strerror** or **abort**) that are better suited and that leave the execution in a defined state.

6 NOTE 2 Similarly, if the program using the **attest** function is not able to guarantee that the evaluation of the argument expression has no side effects, the

behavior is undefined and it should not be relied upon that code following the call is not executed.

Recommended practice

7 Implementations are encouraged to diagnose expressions E that contain operations or C library calls that produce observable side effects.

8 EXAMPLE The following program assumes that each execution is provided with at least two command line arguments. The behavior of an execution with less arguments is undefined.

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 int main(int argc, char* argv[]) {
5   attest(argc > 2);
6   if (argc <= 2) return puts("this should never be reached");
7   return printf("we see %s\n", argv[2]);
8 }
```

The possible effect of using the **attest** function is that the resulting executable unconditionally executes a tail call to **printf** that never returns to the **main** function.

4.3. The testify macro

CHANGE 6. *Add the macro* **testify** *to the macros listed in* 7.2 p1 *and* p2.

CHANGE 7. *Add a new sub-clause to the existing clause* 7.2.1.

7.2.1.2 The testify macro

Constraint

1 A translation unit that includes the header <assert.h> shall otherwise only use the identifier **testify** as the function designation of a function call **testify**(E) with an argument expression E and such that E has a scalar type.

Description

2 A function call **testify**(E) with scalar expression E indicates that it is expected that the control flow will never reach the call where E evaluates to zero or null. Such a call is an expression of type **void**. E shall be such that, other than computing the result, no discrepancy in the state of the abstract machine before and after its evaluation is observable.

3 Similar to the **assert** macro, if the **NDEBUG** macro is not defined at the point of inclusion of <assert.h> and E is zero or null, a runtime diagnostic is issued to the standard error stream and **abort** is called; if the **NDEBUG** macro is defined, the program execution shall not reach such a call where E evaluates to zero or null.

4 If f is a declared function, a function pointer or a lambda value such that a call **testify**(E) is evaluated in one of its parameter declarations, any call to f shall be such that after an assignment of the arguments to their respective parameters, E is not zero or null.

Returns

5 The **testify** macro returns no value. If the **NDEBUG** macro is defined as indicated and the value of the argument expression evaluates to zero or null, the behavior is undefined.

6 NOTE 1 The **testify** macro is quite particular as it specifies a macro for which, if included with the **NDEBUG** macro defined, the program (and not the implementation) has to ensure that no call is reached where the argument expression evaluates to zero or null. Executables that are the result of the translation of such a program will in general not be able to cope with a violation of that assumption and the behavior under such circumstances will be erratic. For example, statements that come logically after a call to **testify** may then be executed unintendedly. Programs that do not provide the necessary guarantees themselves but seek to ensure that a specific erroneous branch of control flow is not executed should avoid to include `<asssert.h>` when **NDEBUG** is defined.

7 NOTE 2 Similarly, if the program using the **testify** function is not able to guarantee that the evaluation of the argument expression has no side effects, the behavior is undefined and it should not be relied upon that code following the call is not executed.

Recommended practice

8 Implementations are encouraged to diagnose expressions E that contain operations or C library calls that produce observable side effects.

9 If the **NDEBUG** macro is not defined as indicated in the context of the caller of `f` as above, it is recommended that prior to the call to `f` a call **testify**(E) is executed as if in the context of the called function body, but with the line number and source file information of the visible declaration (if `f` is a function or function pointer) or of the evaluation of the lambda expression. Thereby, the validity of calls to `f` is tested independently from the setting of the **NDEBUG** macro in the context of the definition (for functions) or lambda expression.

10 EXAMPLE 1 The following program assumes that each execution is provided with at least two command line arguments. If **NDEBUG** is defined at the point of inclusion of `<assert.h>`, the behavior of an execution with less arguments is undefined; otherwise a runtime diagnostic is printed and the execution is aborted.

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 int main(int argc, char* argv[]) {
5   testify(argc > 2);
6   if (argc <= 2) return puts("this␣should␣never␣be␣reached");
7   return printf("we␣see␣%s\n", argv[2]);
8 }
```

Regardless whether **NDEBUG** is defined, the possible effect of using the **testify** macro is that the resulting executable unconditionally executes a tail call to **printf** that never returns to the **main** function.

11 EXAMPLE 2 The following program is similar to the previous, only that here the test is integrated into a function interface.

```c
#include <assert.h>
int print(int argc,
          char* argv[static (testify(argc > 2), argc + 1)]);

int main(int argc, char* argv[]) {
  return print(argc, argv);
}
```

```c
// A different translation unit
#include <stdio.h>
#undef NDEBUG
#define NDEBUG 1
#include <assert.h>

int print(int argc,
          char* argv[static (testify(argc > 2), argc + 1)]) {
  return printf("we see %s\n", argv[2]);
}
```

Since **NDEBUG** is defined in the translation unit of `print`, the possible effect of using the **testify** macro is that the resulting executable never performs the test and unconditionally executes a tail call to **printf** that never returns to the `print` function; no diagnostic for this unit is expected at translation time. If **NDEBUG** is not defined in the translation unit of **main** and the program is executed with no or one commandline arguments, it is recommended that, instead of calling `print`, a diagnostic is printed to the error stream and that the execution is then aborted; the diagnostic is expected to use the line number information of the visible declaration (here line 3) and the name of the first source file.

5. QUESTION FOR WG14

QUESTION 1. *Does WG14 want an* **unreachable** *feature as described in N2757 for C23?*

QUESTION 2. *Does WG14 want an* **attest** *feature as described in N2757 for C23?*

QUESTION 3. *Does WG14 want a* **testify** *feature as described in N2757 for C23?*

QUESTION 4. *Does WG14 want to integrate changes 1, 2 and 3 as proposed in N2757 into C23?*

QUESTION 5. *Does WG14 want to integrate changes 1, 4 and 5 as proposed in N2757 into C23?*

QUESTION 6. *Does WG14 want to integrate changes 6 and 7 as proposed in N2757 into C23?*

QUESTION 7. *Shall the changes voted for C23 also be integrated into TS 6010?*

6. ACKNOWLEDGEMENTS

This paper is the result of fruitful discussions on the C and C++ liaison mail list and follow ups, in particular with Aaron Ballman, Jens Maurer, Martin Uecker, and Miguel Ojeda.