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

N2816 v1
P2390R1

September 17, 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 the feature **unreachable** to specify branches 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.

History: This was preceeded by a WG21 paper that introduces an equivalent feature to C++, https://wg21.link/p0627. It is likely that that feature will be adopted for C++23.
In its Oct. 2021 meeting, WG14 was much in favor (22-1-1) of adding this feature to C23.

Changes to v0:

— Removal of the **attest** and **testify** features.
— Removal of the optional string argument to **unreachable**.
— Allow functions where all control flow is unreachable.
— Reduction of the **unreachable** feature to the bare minimum as it is currently implemented by the **__builtin_unreachable** feature in a set of compilers.
— Propose variants as either a syntax construct or a pseudo-function.
— Add an optional change for a **noreturn** function specification.

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 branch 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

a tool 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 a feature that operates similar to a function **std::unreachable** as it already has been proposed to C++, see https://wg21.link/p0627.[1]

```
namespace std {
  [[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.
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.
This 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 identifier that we propose for the new feature are already claimed in other parts of the community. This is obviously the case for **unreachable** which is already a soon-to-be-integrated feature for C++. On the other hand, we found no use in C sources of that identifier that would not be compatible with our proposal.

2.2. Proposed feature

We now propose one single addition to C that is similar to a pseudo-function **std::unreachable** as it has been proposed to C++, see https://wg21.link/p0627. That proposal is based on a widely implemented practice for extensions in C and C++ namely as **__builtin_unreachable** and similar to a use of a standard function in Rust,

```
unsafe { std::hint::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.

[1] The variant that had an optional message argument has been removed from the proposal.

A previous version of this paper proposed derived features called **attest** and **testify** that would evaluate a logical expression which is expected to hold whenever a call is encountered. The fact of being based on the evaluation of an expression (which may itself be undefined or result in traps or other run-time exceptions) makes the specification of such features much more difficult.

We only found one existing extension in that direction, `__assume` of the MSVC compiler. Unfortunately the documentation for that extension lacks the description of most important properties (return type, behavior when called with invalid expressions) such that we were not able to provide a text for the integration under the given time constraints. What seems clear, though, is that the use as `__assume(false)` largely corresponds to a call **unreachable**`()` as proposed here.

Implementations or third party libraries that intend to provide extensions that would use **unreachable** subject to the evaluation of some expression and that would better integrate with debugging are invited to do so. It would be much helpful if they could document their choices thoroughly, to ease a future standardization effort.

2.3. Specification method

There are different possible specifications for the proposed feature. The case of attributes has already been ruled out for C++ because attributes would miss an important use case, namely that a "call" to the feature could be used in an expression and not only in a statement.

Similar to the existing **assert** macro, our choice has been to reclaim a behavior that is syntactically a function call. Then the semantic properties leave basically two choices: either the new feature is integrated as a proper syntax construct or it is proposed as a pseudo-function in the library. The syntax variant is much more concise and easier to read, so it is presented first.

For the function interface variant, to impede the least possible on existing code the next choice has been to provide the feature via a header. 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, C++ prospects the feature by explicitly providing a prototype for a function **std**`::`**unreachable** that includes a `[[`**noreturn**`]]` attribute. If we want an interface as a pseudo-function, for C we have to deviate from that specification (not from the semantics) for several reasons.

2.4.1. Visibility. First, for C, there is no such thing as a **namespace** declaration, and so the names that we use could interact badly with existing code that already uses the same identifiers. We mitigate that problem by insisting that **unreachable** should be a pseudo-function that, as specified here, will never have external definitions that could interact with TU that don't use the feature.

So existing TU that use the identifier **unreachable** for another purpose will generally not be affected by this addition if they don't include `<assert.h>`. This should very much limit the possibility of conflicts with existing binaries where sources for recompilation might not be available.

2.4.2. Lack of external definition. One point of debate has also been whether or not it should be possible to take the address of an **unreachable** function, or stated otherwise, if implementations would have to provide an external definition in their C library. There are several reasons why we think that this is not a good idea:

— There is no reference implementation for this. The **__builtin_unreachable** feature as currently implemented doesn't allow it.
— This would add a rarely used symbol to *all* C library implemenations.
— This may conflict with legacy code.
— Since C++ library function don't provide external symbols by default, the C feature could not be easily mapped onto an existing C++ feature.
— Application code that thinks that such an blackbox-UB function pointer is a good idea, could easily provide it in form of an 'inline' wrapper. There is no need to add that to the C library.
— The only case where a function pointer to **unreachable** could be advantegous is if the implementation is able to do a whole programm analysis and to prove that all control flow leads to a call always has the same function pointer. This would, again, hide optimization oportunities behind a complex deduction scheme involving UB, and would be a violation of Principle 3, above.

2.4.3. Undefined behavior versus **noreturn** *function declarations.* An important property of the proposed feature is that code that executes it 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. Neither do we want to insinuate any expectation for users of that feature. Therefore we think that 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 control flow branches that lexically follow the call.
Nevertheless, in case that WG14 thinks that a **noreturn** specification would be helpful we provide text for its addition as an optional change.

3. IMPACT

Other than using the previously unreserved identifier **unreachable** the proposed additions have no impact on existing code for both variants.
For the function variant, since there is no addition of function symbols to the C library, that impact is limited to code that includes the header `<assert.h>`. If that would be considered too intrusive the feature could be proposed with a new header, but we don't think that this is necessary.
For the syntax variant, impact for code that already uses the identifier **unreachable** is dependent of the choice for the keyword.

3.1. Compatibility with C++

The proposed C and C++ features are designed to be semantically equivalent. In particular, C++ can simply add code similar to

```
using std::unreachable;
#define unreachable unreachable
```

or

```
#define unreachable std::unreachable
```

to their compatibility header `<cassert>`.

4. PROPOSED CHANGES

4.1. Variant: syntax construct

In the following we use **UNREACHABLEKEYWORD** as a placeholder for the concrete choice of a keyword, see Question 4, below.

CHANGE 1. *Add a new keyword* **UNREACHABLEKEYWORD** *to the list of* 6.4.1

With that, we can anchor the new syntax at the same level as function calls, namely as postfix expression. It has the same precedence as a function call and is disambiguated from such calls because the use of a new keyword (and not an identifier).

CHANGE 2. *In* 6.5.2 *append a new alternative "unreachable-expression" to the* "postfix-expression" *production rule.*

The additional text that is needed can then be minimal.

CHANGE 3. *Add a new clause* 6.5.2.6.

> **6.5.2.6 Unreachable expressions**
>
> **Syntax**
>
> 1 *unreachable-expression*: **UNREACHABLEKEYWORD** ( )
>
> **Description**
>
> 2 An unreachable expression has type **void** and indicates that the particular flow of control that leads to it will never be taken. Such an expression shall not be evaluated.

Independent of WG14's choice for the keyword, we may add a macro to <assert.h> much as we previously did for **static_assert**.

CHANGE 4 (OPTIONAL). *Add* **unreachable** *to the list of macros of the* <assert.h> *header and add the following sentence were appropriate*

> The macro **unreachable** expands to **UNREACHABLEKEYWORD**.

Providing such a macro for any choice of the keyword, even for **unreachable**, would make it posssible for C++ to add a macro wrapper in their <cassert> compatibility header.

4.2. Variant: function

CHANGE 5. *Change the title of the* `<assert.h>` *library clause to*

> *7.2* ~~Diagnostics~~**Assertions** `<assert.h>`

CHANGE 6. *Add* **`unreachable`** *as a function to the list in* `7.2 p1`.

CHANGE 7. *Add a new clause to the* `<assert.h>` *library clause.*

> **7.2.2 Control flow assertions**

CHANGE 8. *Add a new sub-clause to the new clause* `7.2.2`.

> **7.2.2.1 The `unreachable` function**
>
> **Synopsis**

```
1  #include <assert.h>
   void unreachable(void);
```

> **Description**
>
> 2 A function call using the **`unreachable`** function indicates that the particular flow of control that leads to the call will never be taken. The function designator **`unreachable`** shall not be used other than in the token sequence
>
> **`unreachable ( )`**
>
> corresponding to such a call.[FNT1] The program execution shall not reach such a call.
>
> [FNT1] That is, constructs that intend to take the address of the function such as **`(unreachable)()`** or similar are undefined even if the execution does not reach a call to that pointer. The same holds for redeclaration of the identifier as an object or function with linkage, even if it declares a function that is compatible to the declaration given here.
>
> 3 A translation unit that includes the header `<assert.h>` shall not define or undefine a macro of the same name. The implementation shall not provide an external definition for the **`unreachable`** identifier.
>
> **Returns**
>
> 4 If a function call with **`unreachable`** as the function designation is reached during execution the behavior is undefined.

CHANGE 9 (OPTIONAL). *Use the following synopsis for the* **`unreachable`** *function instead of the one proposed in Change 8:*

```
#include <assert.h>
_Noreturn void unreachable(void);
```

4.3. Possible example

CHANGE 10 (OPTIONAL). *Add the following example*

EXAMPLE The following program assumes that each execution is provided with at least one command line arguments. The behavior of an execution with no arguments is undefined.

```
1  #include <assert.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[static argc+1]) {
5    if (argc <= 2) unreachable();
6    else return printf("%s: we see %s\n", argv[0], argv[1]);
7    return puts("this should never be reached");
8  }
```

Here, the **static** array size expression and the annotation of the control flow with **unreachable** indicates that the pointed-to parameter array `argv` will hold at least three elements, regardless of the circumstances. A possible optimization 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 entire call and reference to **puts** can be omitted from the executable. No diagnostic is expected.

4.4. TS 6010

If added to C23, the feature as described should also be added to TS 6010 such that the intended difference from C17 is noted and such that recommended practice that uses **unreachable** can be formulated within the scope of the TS.

5. QUESTIONS FOR WG14

QUESTION 1. *Could you live with the syntax variant?*

QUESTION 2. *Could you live with the function variant?*

QUESTION 3 (PREFERENCE). *Which variant do you prefer?*

(4.1) the syntax variant
(4.2) the function variant

5.1. Syntax variant

QUESTION 4. *Which spelling do you prefer for the keyword?*

(*1*) **unreachable**
(*2*) **_Unreachable**
(*3*) **__builtin_unreachable**

QUESTION 5. *Shall Changes 1 to 3 of N2816 (with a replacement of* **UNREACHABLEKEYWORD** *by the above choice) be integrated into C23?*

QUESTION 6. *Shall Change 4 of N2816 (with a replacement of* **UNREACHABLEKEYWORD** *by the above choice) be integrated into C23?*

5.2. Function variant

QUESTION 7. *Does WG14 want to integrate the unreachable feature as a function interface as described by Changes 5 to 8 in N2816 into C23?*

QUESTION 8. *Does WG14 want to apply optional Change 9 in N2816 for the inclusion of the* **unreachable** *function in C23?*

5.3. Example

QUESTION 9. *Does WG14 want to integrate an example as described by Change 10 in N2816 (position to be determined by the editors) into C23?*

5.4. TS 6010

QUESTION 10. *Shall the changes that are 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, the WG14 plenary and follow ups, in particular with Aaron Ballman, Eskil Steenberg, Jens Maurer, Joseph Myers, Martin Uecker, and Miguel Ojeda.