Title:	Function to mark unreachable code
Document Number:	P0627R5
Date:	2021-09-15
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	WG21 – Library Working Group (LWG)
Reply-to:	Jens Maurer <jens.maurer@gmx.net>
Original author:	Melissa Mears

1. Introduction

This proposal introduces a new standard library function, `std::unreachable`, for marking locations in code execution as being known by the programmer to be unreachable.

2. Document History

2017-03-14 – Revision 0, first published release.

2017-06-12 – Revision 1:

- Removed “Open Questions” section, since these questions have been resolved.
- Updated references to the Standard to refer to the N4659 draft instead of N4618.
- Improved the proposed Standardese considerably on advice from Jens Maurer.
- Added a Qt-like example of how `__assume` could be simulated using `[[unreachable]]`.
- Minor fixes throughout the document.

2017-07-15 – Revision 2:

- Added message parameter.
- Added notes about concerns from the EWG meeting.

2018-10-08 – Revision 3:

- Switched to being a function instead of an attribute.
- Added Toronto 2017 EWG straw poll information and retargeted for LEWG.

2019-07-14 – Revision 4:

- Integrated requested changes from Kona February 2019.

2021-09-09 – Revision 5:

- Integrated requested changes from LWG review in Prague 2020
- Remove overload with “message” parameter.

3. Motivation and Scope

Compilers cannot know every situation in which code may execute, thanks to the Halting Problem. There will always exist programs in which a compiler cannot determine that a situation is impossible.

When the programmer knows that a situation is impossible, but it is not obvious to the compiler, it is helpful to be able to tell the compiler to avoid runtime checking for a case that is impossible.

For example, a common situation is that a `switch` statement handles all possible situations, but it's not obvious to the compiler. Given this example `switch` statement:

```
void do_something(int number_that_is_only_0_1_2_or_3)
{
	switch (number_that_is_only_0_1_2_or_3)
	{
	case 0:
	case 2:
		handle_0_or_2();
		break;
	case 1:
		handle_1();
		break;
	case 3:
		handle_3();
		break;
	}
}
```

…a compiler might generate object code like this (using Intel-syntax x86-64 as an example):

```
cmp eax, 4
jae skip_switch
lea rcx, [jump_table]
jmp qword [rcx + rax*8]
```

If, however, we had a way to tell the compiler that no other value is possible, the compiler could omit the first two instructions, the ones checking for a value that is not `0` `1` `2` or `3`.

Another case in which it would be nice to tell a compiler that something cannot happen is with non-obvious cases of a function never returning. An example from POSIX could be the following:

```
[[noreturn]] void kill_self()
{
	kill(getpid(), SIGKILL);
}
```

Such code cannot fail or return, but generally, a compiler will issue a warning that `kill_self` might return despite its `[[noreturn]]` attribute.

3.1 Existing implementations

3.1.1 POSIX world

GCC, Clang and Intel C++ all support a directive function named `__builtin_unreachable()`. Calling this “function” tells these compilers that that location in the source code cannot be reached. Thus, the `do_something` and `kill_self` functions from section **2** above would appear as follows:

```
void do_something(int number_that_is_only_0_1_2_or_3)
{
      switch (number_that_is_only_0_1_2_or_3)
      {
      case 0:
      case 2:
            handle_0_or_2();
            break;
      case 1:
            handle_1();
            break;
      case 3:
            handle_3();
            break;
      default:
            __builtin_unreachable();
      }
}

[[noreturn]] void kill_self()
{
      kill(getpid(), SIGKILL);
      __builtin_unreachable();
}
```

3.1.2 Windows world

Microsoft Visual C++ doesn't have such a directive, but it has an alternative that can produce the same effect. Visual C++ has `__assume(E)`, which directs the compiler to assume that arbitrary Boolean expression `E` is true when execution reaches that location.

If one uses the contradictory statement `__assume(false)`, Visual C++ assumes that execution cannot reach this point, much like the behavior of `__builtin_unreachable()`.

3.1.3 Other implementations

In some implementations, it is possible to accomplish unreachability assumptions by intentionally causing undefined behavior, such as intentionally dividing by zero in the unreachable case. However, the author feels that that should not be encouraged. Instead, we'll make something that directly is undefined behavior in a standard manner.

4. Design Decisions

4.1 Form of the directive

The major compilers all support a similar useful feature, and it would be nice to have a standard way.

The author sees the `__assume(E)` design (`__assume(false)`) and the similar Contract-Based Programming proposal as undesirable for few reasons that are covered in the next section.

Additionally, the proposed `std::unreachable()` could easily be used to provide the functionality of `__assume`, using something like this (the idea coming from Qt's source code):

```
#define ASSUME(...) if (__VA_ARGS__); else std::unreachable()
```

An EWG straw poll at Toronto 2017 indicated that a function—say, `std::unreachable()`—would be preferred to a new attribute `[[unreachable]]`. One reason brought up at EWG that suggested that a library function is preferred is that it allows changing the internal implementation later.

4.2 Diagnostic string

EWG expressed in the Toronto 2017 straw poll the recommendation that `std::unreachable` take an optional message parameter. This parameter would be used by implementations—and certain modes of implementations—that trap in response to `std::unreachable()` executing, providing a runtime diagnostic message.

This overload was present in R3 of this paper, but was removed again in R5, for the following reasons:

- Existing practice in implementations does not allow for a message.
- Given the function interface, there is no way to guarantee that the message is a compile-time string.
- WG14 is intending to add "unreachable" to C23, but only without the message parameter.

4.3 Comparison with Contract-Based Programming proposal (P0380/P0542)

A proposal for adding contract-based programming to C++ is proposed by paper P0380 and formalized by paper P0542. The contracts proposal turns out to be a superset of this proposal; the closest contracts equivalent to this proposal's `std::unreachable()` is `[[assert axiom: false]]`.

`[[assert: E]]` means that the given expression `E` must be true at a certain point. The `axiom` "checking level" means that no runtime checking is to be done. `[[assert axiom: E]]` is thus very similar to the Visual C++ extension `__assume(E)`.

The author believes that utilizing the contract-based programming proposal, or Visual C++'s `__assume`, instead of a separate `std::unreachable()` is undesirable for the following reasons:

- The Contracts specification is incomplete, and it will be a while before it is ready.
- `[[assert axiom: false]]` doesn't convey to programmers that that statement is unreachable like `std::unreachable()` does. It looks like, and is, a logical contradiction, and so is awkward to comprehend. `std::unreachable()` is more clear in its meaning.
- The most problematic aspect the author sees is that the Contracts proposal does not state that the effect of failing an `[[assert axiom]]` is undefined behavior. `std::unreachable()` denoting undefined behavior at the point is a desirable feature (see next section).
- `std::unreachable()` can be implemented in terms of Contracts in the future if desired. Its behavior could then be defined as a contract failure without breaking backward compatibility. (Because the behavior would previously be undefined, programs would not expect anything.)

4.4 Definition

What is the best way to define the attribute's effect? The author feels that the best way is to make the behavior of `std::unreachable()` be undefined. There are several reasons:

- `std::unreachable()` causing undefined behavior means that the Standard would not prescribe any particular action, leaving open many possible implementation actions.
- Some compilers already associate being unreachable to undefined behavior. Clang's documentation states that `__builtin_unreachable()` "has completely undefined behavior".
- Optimizing under the assumption that a statement is unreachable, and thus having unpredictable behavior if the statement is in fact reachable, falls naturally under "undefined behavior".
- An alternative would be to issue a trap if `std::unreachable()` is executed. This could be used in "debug builds", for example. Such a trap falls under "undefined behavior".
- Being undefined behavior implies what happens if a `constexpr` function calls `std::unreachable()`: it's not a constant-expression, by (N4713) **[expr.const]/2.6**.

4.5 Concerns over proliferation of undefined behavior

A concern brought up at EWG was that this feature would result in further proliferation of undefined behavior. The author acknowledges that this indeed introduces an additional case of undefined behavior, but does not believe this to be a large problem.

In the author's opinion, it is not undefined behavior itself that is the problem, but rather *unexpected* undefined behavior. C++ will always have undefined behavior; this is unavoidable. Situations in which undefined behavior occurs unexpectedly are problematic for programmers, because they often lead to subtle bugs, or unexpected compiler optimizations. However, `std::unreachable()` is *expected* undefined behavior: a programmer using it knows that undefined behavior will occur there and is making a calculated risk in using it.

5. Example Implementation

Compilers with magic for inducing undefined behavior would just use those.

Even without compiler magic, a compliant implementation is simple:

```
namespace std {
      [[noreturn]] void unreachable() { }
}
```

6. Impact on Existing Implementations

The major implementations already have support for this feature in a different form, so modifications to implementations of the Standard Library to support `std::unreachable()` should be simple.

7. Proposed Wording

The proposed Standardese wording is below. All portions are additions.

Add a new prototype to the `<utility>` synopsis in **[utility.syn]**:

X.X.X Header `<utility>` synopsis **[utility.syn]**

```
#include <initializer_list>      // see X.X.X

namespace std {
  …
  // X.X.X, unreachable
  [[noreturn]] void unreachable();
}
```

Add a new section to **[utility]**:

X.X.X Function unreachable [utility.unreachable]

```
[[noreturn]] void unreachable();
```

1 *Preconditions:* `false` is `true`. [Note: This precondition cannot be satisfied, thus the behavior of calling `unreachable` is undefined. —end note]

2 [*Example:*

```
int f(int x) {
  switch (x) {
  case 0:
  case 1:
    return x;
  default:
    std::unreachable();
  }
}

int a = f(1);  // OK; a has value 1
int b = f(3);  // undefined behavior
```

— end example]

Add a new entry to the **[tab:support.ft]** table in **[support.limits.general]**:

Table X: Standard library feature-test macros [tab:support.ft]

Macro name	Value	Header(s)
...		
`__cpp_lib_unreachable`	*XXXXYY*L	`<utility>`

8. Straw Polls

8.1 EWG Straw Poll (Toronto 2017)

At Toronto 2017, the EWG straw poll voted to send this proposal with the optional message addition to LEWG. SF=4, F=10, N=2, A=1, SA=1

Additionally, a straw poll asked whether to recommend to LEWG that this functionality be available without a header. This aspect has not been incorporated above, because I do not know how the wording would function. SF=2, F=7, N=8, A=1, SA=1

8.2 LEWG Straw Poll (San Diego 2018)

In San Diego 2018, LEWG voted to forward to LWG unchanged:

SF=1, F=9, N=1, A=0, SA=0

9. References

- N4713 Working Draft, Standard for Programming Language C++: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf
- November 2015 std-proposals mailing list discussion thread started by Nicol Bolas: https://groups.google.com/a/isocpp.org/forum/#!searchin/std-proposals/unreachable/std-proposals/f1G45z3dMp0/04qGH9X0FQAJ
- “A Contract Design” proposal for defining contractual programming in C++: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0380r1.pdf http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0542r0.html
- GCC documentation on `__builtin_unreachable`: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html#index-_005f_005fbuiltin_005funreachable
- Clang documentation on `__builtin_unreachable`: https://clang.llvm.org/docs/LanguageExtensions.html#builtin-unreachable
- Visual C++ documentation on `__assume`: https://msdn.microsoft.com/en-us/library/1b3fsfxw.aspx

10. Acknowledgements

The original author of this proposal is Melissa Mears, shepherding it through most of the process. All the thanks should go to her, all the errors are mine.

The author would like to recognize the contributions of the members of the std-proposals discussion group on the subject: Nicol Bolas for starting the thread with the idea, Richard Smith for pointing out how ignoring the attribute is a correct implementation, and Thiago Macieira for noting other possible compiler reactions. And more recently, James Touton for helping with presenting this to the Committee.