Title: Attribute to mark unreachable code
Document Number: P0627R1
Date: 2017-06-12
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: Evolution Working Group (EWG)
Reply-to: Melissa Mears <myriachan@gmail.com>

1. Introduction

This proposal introduces a new standard attribute, `[[unreachable]]`, for marking statements 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.

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.

2.1 Existing implementations

2.1.1 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();
}
```

2.1.2 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()`.

2.1.3 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.

3. Design Decisions

3.1 Form of the directive

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

One possibility is creating a function named `std::unreachable()`, but this seems awkward, and has disadvantages compared to using an attribute. It would have to use compiler magic to achieve the desired warning suppression from the `kill_self` example.

An attribute is an interesting solution: `[[unreachable]]`. Since being unreachable is an aspect of code flow, being attached to a statement seems the most appropriate. Naturally, in practice, `[[unreachable]]` would be attached to a null statement, which this proposal requires.

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 `[[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 [[unreachable]]
```

3.2 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's `[[unreachable]]`: the closest contracts equivalent to `[[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 `[[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 `[[unreachable]]` does. It looks like, and is, a logical contradiction, and so is awkward to comprehend. `[[unreachable]]` is clear in its meaning.
- `__assume`-like functionality can be implemented in terms of `[[unreachable]]` if desired.

- Finally, 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. `[[unreachable]]` denoting undefined behavior at the point is a desirable feature (see next section).

3.3 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 a statement with `[[unreachable]]` be undefined. There are several reasons:

- `[[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 for implementations would be to issue a trap if an `[[unreachable]]` statement is executed. This could be used in "debug builds", for example. Such a trap falls under "undefined behavior".

- `[[unreachable]]` is ignorable. An implementation that does not understand `[[unreachable]]` and thus ignores it (as required by N4659 **[dcl.attr.grammar]/6**) would actually be a correct implementation of `[[unreachable]]`, because "do nothing" also falls under the purview of "undefined behavior".

- Being undefined behavior implies the answer to the question of what happens if a `constexpr` function executes an `[[unreachable]]` statement: it's not a constant-expression, by (N4659) **[expr.const]/2.6**.

4. Impact on the Standard

This proposal is purely a new attribute. Existing code does not use this attribute. This proposal has no impact on the Standard Library.

As noted above in **3.3**, if an implementation does not recognize the new attribute `[[unreachable]]`, and ignores it as required by the existing Standard, it is in fact already a correct implementation.

5. Impact on Existing Implementations

The major implementations already have support for this feature in a different form, so modifications to support `[[unreachable]]` should be simple. Alternatives include continuing to ignore `[[unreachable]]` or to turn it into a trap.

6. Proposed Wording

The proposed Standardese wording below is relative to N4659. All portions are additions.

Add a new section to 10.6 (**[dcl.attr]**):

10.6.*X* Unreachable attribute [dcl.attr.unreachable]

1 The *attribute-token* `unreachable` may be applied to a null statement (**[stmt.expr]**); such a statement is called an *unreachable statement*. The *attribute-token* `unreachable` shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present.

2 The behavior of executing an unreachable statement is undefined. [*Note:* The `unreachable` attribute is intended as a hint to the implementation that execution cannot reach the statement so marked. An implementation may use this hint to optimize under this assumption or to trap if the statement indeed executes, among other alternatives. — *end note*]

3 [*Note:* Implementations should not issue a warning regarding undefined behavior along the execution path of an unreachable statement, such as from reaching the end of a function returning other than `void` (**[stmt.return]**) or the end of a function with the `[[noreturn]]` attribute (**[dcl.attr.noreturn]**). — *end note*]

4 [*Example:*

```
int f(int x) {
  switch (x) {
  case 0:
  case 1:
    return x;
  case 2:
    [[unreachable]] return x;   // error: not a null statement
  default:
    [[unreachable]];
  }
  // implementations should not emit a warning about a
  // missing return statement at the end of this function
}

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

— *end example*]

7. Feature-Testing Macro

For the purposes of SG10 SD-6, `__has_cpp_attribute(unreachable)` suffices.

8. References

- N4659 Working Draft, Standard for Programming Language C++: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.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

9. Acknowledgements

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, Thiago Macieira for noting other possible compiler reactions, and Jens Maurer for improving the Standardese.