Proposal for C2x
WG14 N2408

Title:	The `fallthrough` attribute
Author, affiliation:	Aaron Ballman, GrammaTech
Date:	2019-08-11
Proposal category:	New features
Target audience:	General developers, compiler/tooling developers

Abstract: A lesser-used idiom within a switch statement is the ability to "fallthrough" from a one case block into another case block. This can create expensive logic errors when the fallthrough behavior is unintentional because the secondary case functionality may appear to be reasonable at first glance, causing the incorrect behavior to be discovered elsewhere in the code base. Compilers cannot easily diagnose fallthrough behavior because there is no way to distinguish programmer intent: sometimes fallthrough behavior is expected and desirable, other times it is a bug. This paper proposes a new attribute, `fallthrough`, to convey whether this behavior is expected by the programmer.

Prior art: C++ has this feature using the same syntax.

The fallthrough attribute

Reply-to: Aaron Ballman (aaron@aaronballman.com; aballman@grammatech.com)
Document No: N2408
Revises Document No: N2268
Date: 2018-08-11

Summary of Changes

N2408

- Added an example of how to annotate Duff's Device with `[[fallthrough]]`.
- Added rationale for why it is a constraint violation for the attribute to not be followed by a label.

N2268

- Changed a mention of "warning" to be "diagnostic" instead.
- Rearranged and renumbered paragraphs; the [Note] is now a Recommended Practice section.

N2216

- Added the appropriate cover page.
- Rebased on top of N2165, the latest attributes syntax proposal

N2052

- Original proposal.

Introduction

A lesser-used idiom within a switch statement is the ability to "fallthrough" from a one case block into another case block. This can create expensive logic errors when the fallthrough behavior is unintentional because the secondary case functionality may appear to be reasonable at first glance, causing the incorrect behavior to be discovered elsewhere in the code base. Compilers cannot easily diagnose fallthrough behavior because there is no way to distinguish programmer intent: sometimes fallthrough behavior is expected and desirable, other times it is a bug.

Rationale

The `[[fallthrough]]` attribute has real-world use, being implemented by Clang and GCC as C++ vendor-specific attributes, and was standardized under the name `[[fallthrough]]` by WG21. This attribute cannot be implemented via the `__declspec` or `__attribute__` vendor-specific extensions because it is an attribute that appertains to a statement instead of a declaration or a type.

Proposal

This document proposes the `[[fallthrough]]` attribute as a way for a programmer to specify that fallthrough behavior is desirable under the assumption that silently fallthrough is an accidental omission of a break statement. This allows programmers to specify their intent explicitly, giving an implementation the opportunity to diagnose fallthrough behavior in the absence of explicit marking.

Consider:

```
enum E { Zero, One, Two, Three };
 size_t calculate_required_buffer_size(enum E e) {
   size_t Ret = 0;
   switch (some_value) {
   case Zero: // (0)
   case One:
     Ret = 100;
     break;
   case Two:
     Ret = 200; // (1)
   case Three:
     Ret = 20; // (2)
     break;
   default:
     assert(0 && "Unknown enumeration value");
   }
   return Ret;
 }
```

If the enumerator `Two` is passed to `calculate_required_buffer_size()`, the `Ret` variable will store the value `200` at (1). However, because there is no break statement following the assignment, execution "falls through" to the next statement, overwriting `Ret` with the value `20` at (2). This results in a likely unexpected value being returned from `calculate_required_buffer_size()` that could, e.g., lead to the caller allocating an insufficient amount of memory for an object, resulting in an exploitable buffer overrun. Without an annotation, the compiler has insufficient information to determine whether falling through from one case to another is intended behavior. Thus, the fallthrough into (2) can be diagnosed by a compiler. If the fallthrough is desired, the programmer can instead write:

```
// ...
case Two:
  Ret = 200;
  [[fallthrough]];
case Three:
  // ...
```

The fallthrough at (0) is left to QoI as to whether a diagnostic is desired or not. Common practice is for (0) to not diagnose fallthrough because there is only a single newline between the case labels.

The `[[fallthrough]]` attribute can only be applied to a null statement that precedes a case or default label statement within a switch statement. It is a constraint violation to use the `[[fallthrough]]` attribute when fallthrough to another label is impossible. This is compatible with the behavior specified in C++ and is desirable because such a situation signifies programmer confusion (while the purpose to the attribute is to clarify programmer intent) and the code can always be written to properly position the attribute. For instance, at the London 2019 meeting, it was asked whether you could use the `[[fallthrough]]` attribute when a subsequent `case` or `default` label is controlled by a macro. You can, but you must be cognizant of this rule regarding `[[fallthrough]]` attribute being immediately followed by a label to fall through to. The incorrect code unconditionally uses the `[[fallthrough]]` attribute, which results in a constraint violation when `ON` is not defined. The correct pattern is to introduce the attribute within the `FROBBLE` macro definition directly.

Incorrect	Correct

`#ifdef ON`
`#define FROBBLE case 1: return 2`
`#else`
`#define FROBBLE break`
`#endif`
 	
`int func(int i) {`
`  int j = 0;`
`  switch (i) {`
`  case 0:`
`  j = 10;`
`  [[fallthrough]];`
`  FROBBLE;`
`  }`
`  return j;`
`}`
	`#ifdef ON`
	`#define FROBBLE [[fallthrough]]; \`
	`  case 1: return 2`
	`#else`
	`#define FROBBLE break`
	`#endif`
	
	`int func(int i) {`
	`  int j = 0;`
	`  switch (i) {`
	`  case 0:`
	`  j = 10;`
	`  FROBBLE;`
	`  }`
	`  return j;`
	`}`

The London 2019 meeting also raised the question as to how this attribute will work with Duff's Device or other, less idiomatic switch statements. Below is an example demonstrating how you would annotate such a switch statement. A live example demonstrating how Clang behaves with and without the attribute when enabling diagnostics for implicit switch fallthrough can be viewed here: https://godbolt.org/z/eKFkxo

```
void send(int *to, int *from, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++; [[fallthrough]];
    case 7:      *to = *from++; [[fallthrough]];
    case 6:      *to = *from++; [[fallthrough]];
    case 5:      *to = *from++; [[fallthrough]];
    case 4:      *to = *from++; [[fallthrough]];
    case 3:      *to = *from++; [[fallthrough]];
    case 2:      *to = *from++; [[fallthrough]];
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

Proposed Wording

This proposed wording currently uses placeholder terms of art and it references a new subclause from WG14 N2269, 6.7.11, Attributes that describes the referenced grammar terms. The [Note] in paragraph 1 of the semantics is intended to convey informative guidance rather than normative requirements.

6.7.11.2 Fallthrough attribute

Constraints

1 The *attribute-token* `fallthrough` shall be applied to a null statement (6.8.3); such a statement is a fallthrough statement. The *attribute-token* `fallthrough` shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. A fallthrough statement may only appear within an enclosing `switch` statement (6.8.4.2). The next statement that would be executed after a

fallthrough statement shall be a labeled statement whose label is a case label or default label for the same `switch` statement.

Recommended Practice

2 The use of a fallthrough statement is intended to suppress a diagnostic that an implementation might otherwise issue for a case or default label that is reachable from another case or default label along some path of execution. Implementations are encouraged to issue a diagnostic if a fallthrough statement is not dynamically reachable.

3 EXAMPLE

```
void f(int n) {
  void g(void), h(void), i(void);
  switch (n) {
  case 1: /* diagnostic on fallthrough discouraged */
  case 2:
    g();
    [[fallthrough]];
  case 3: /* diagnostic on fallthrough discouraged */
    h();
  case 4: /* fallthrough diagnostic encouraged */
    i();
    [[fallthrough]]; /* constraint violation */
  }
}
```

Acknowledgements

I would like to recognize the following people for their help in this work: David Keaton, David Svoboda, and Andrew Tomazos.

References

[N2269]
Attributes in C. Aaron Ballman. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2269.pdf

[P0068R0]
Proposal of [[unused]], [[nodiscard]] and [[fallthrough]] attributes. Andrew Tomazos. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0068r0.pdf

[P0188R1]
Wording for [[fallthrough]] attribute. Andrew Tomazos. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0188r1.pdf