Proposal of [[unused]], [[nodiscard]] and [[fallthrough]] attributes.

Document No.: P0068R0
Project: Programming Language C++ - Evolution
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2015-09-03

Summary

The attributes [[unused]], [[nodiscard]] and [[fallthrough]] are proposed. [[unused]] silences an unused entity warning. [[nodiscard]] creates a warning if the return value of a function call is discarded. [[fallthrough]] silences an implicit fallthrough warning in a switch. The three attributes have heavy use in existing practice, and we seek standardization of them to provide them with portable names across implementations.

Background

There are currently 3 standard attributes in C++. They are:

```
[[noreturn]]
[[carries_dependency]]
[[deprecated]]
```

They are specified, respectively, in [dcl.attr.noreturn], [dcl.attr.depend] and [dcl.attr.deprecated].

In addition to the standard attributes, most implementations offers a set of non-standard attributes.

Proposal

We propose the standardization of 3 more attributes. They are based on some of the most common and useful existing non-standard attributes. They are:

```
[[unused]]
[[nodiscard]]
[[fallthrough]]
```

We offer a rough informative-only description here of what they do. (For a formal description of what is proposed, see the Wording section below).

An entity marked [[unused]] means that the entity may appear to be unused for some reason. If an implementation would have otherwise warned that the entity isn't used, the [[unused]] attribute suppresses the warning:

```
int x; // WARNING: x is unused
[[unused]] int x; // OK
```

If [[nodiscard]] is marked on a function, it stresses that the return value is not intended to be discarded. If the return value is discarded, a warning is issued:

```
[[nodiscard]] int f();
void g() {
  f(); // WARNING: return value of nodiscard function discarded.
}
```

If [[nodiscard]] is marked on a type, it makes it so that all functions that return that type are implicitly [[nodiscard]].

```
[[nodiscard]] struct S { ... };
```

```
S f();
void g() {
  f(); // WARNING: return value of nodiscard type discarded.
}
```

The [[fallthrough]] attribute is used like a statement we call a *fallthrough statement*:

```
[[fallthrough]];
```

A fallthrough statement is placed just before a case label in a switch. It serves as a hint that the feature of execution “falling through” into the following case-labelled statement is intentional (not accidental). Once a codebase is annotated with [[fallthrough]], the implementation can be configured to issue a warning if it isn’t present:

```
switch (c) {
case ‘a’:
  f(); // WARNING: implicit fallthrough
case ‘b’:
  g();
  [[fallthrough]]; // OK
case ‘c’:
  h();
}
```

Motivation

The three proposed attributes share a common theme of all being about capturing the intent of the programmer to improve the accuracy of diagnostic generation.

In many cases an unused entity can be a symptom of a logic error. For this reason, most implementations can be configured to issue warnings about certain kinds of entities when they are not sufficiently used (by some implementation-defined definition). Such a diagnostic is usually part of the default set (-Wall). In cases where such a warning is emitted but determined to be a false alarm, some technique needs to be used to individually suppress the warning. There are a variety of non-standard techniques:

```
__attribute__((“unused”))
 [[gnu::unused]]
__pragma(warning(suppress:4100))
#pragma foo diagnostic unused-bar ignored
void f(int /*arg*/);
#define UNUSED(x)  (void)(x)
void f(int /*unused*/);
template <typename T> void Unused(T&&) {}
#ifdef COMPILER1
  #define UNUSED something
#elseif COMPILER2
  #define UNUSED something else
etc
```

By standardizing the [[unused]] attribute we would provide a common, portable and more readable way to express this intent.

A value-returning function can be called purely for its side-effects, and the return value discarded. For some functions, the return value is an essential component for its proper operation, and should generally not be discarded. Some types are designed to be used as the return value of such functions. It is a common logic error to accidentally discard a return value of such a function or of such a type. Similar to [[unused]], there are a variety of non-standard techniques to express such a property, such as [[gnu::warn_unused_result]], which provokes a warning when such a return value is discarded. The [[nodiscard]] attribute allows this intent to be expressed in a common, portable way.

An extremely expensive logic error is accidental use of the little-used feature of switch statements whereby the path of execution can implicitly flow from one case block to the one after it. The logic error that ensues from such accidental fall through is usually quite insidious because the side effects of executing a second "wrong" case block generally appear as quite reasonable things to happen locally without considering the big picture of the system architecture. The system misbehaviour is usually caught some distance from the origin, and can take a long-time to diagnose. For this reason, many programming languages offer a fallthrough statement to explicitly denote this intent, and will fail to compile if it is not used on such an execution path. Some C++ implementations are have developed a popular diagnostic and accompanying [[fallthrough]] attribute statement. We propose the standardization of the attribute to provide the feature in a common portable way to C++.

Examples

Example 1

Compiled with an unused variables warning enabled in a release build (NDEBUG):

```
std::pair<int, int> flatten(int x, int y, int z) {
  // WARNING: y unused
  assert(y == 0);
  return {x, z};
}

std::pair<int, int> flatten(int x, int y [[unused]], int z) {
  // OK
  assert(y == 0);
  return {x, z};
}
```

Example 2

When compiled with USE_IMPL1 defined:

```
static void impl1() { ... }
static void impl2() { ... } // warning: impl2 unused

void iface() {
#ifdef USE_IMPL1
  impl1();
#elif USE_IMPL2
  impl2();
#else
  #error set an implementation
#endif
}

[[unused]] static void impl1() { ... }
[[unused]] static void impl2() { ... } // OK

void iface() {
#ifdef USE_IMPL1
  impl1(); // OK
#elif USE_IMPL2
  impl2(); // OK
#else
  #error set an implementation
#endif
}
```

Example 3

```
template< class Function, class... Args>
[[nodiscard]] future async( Function&& f, Args&&... args );

int main() {
  async( []{ f(); } ); // WARNING: return value discarded
  async( []{ g(); } ); // WARNING: return value discarded
}
```

Example 4

```
namespace example {
```

```
// Provides information about the success or
// failure of operations in the example api.
[[nodiscard]] struct Disposition { ... };

Disposition MoveUp();
Disposition MoveDown();

} // namespace example

int main() {
  example::MoveUp(); // WARNING: nodiscard return value discarded
  example::MoveDown(); //  WARNING: nodiscard return value discarded
}
```

Example 5

Compiled with an implementation-defined implicit fallthrough warning enabled:

```
switch (n) {
case 22:
case 33:  // OK: no statements between case labels
  f();
case 44:  // WARNING: no fallthrough statement
  g();
  [[fallthrough]];
case 55:  // OK
  if (x) {
    h();
    break;
  }
  else {
    i();
    [[fallthrough]];
  }
case 66:  // OK
  p();
  [[fallthrough]]; // WARNING: fallthrough statement out-of-place
  q();
case 77:  // WARNING: no fallthrough statement
  r();
}
```

Existing Practice

There are literally millions of examples of the non-standard attributes on which [[unused]], [[nodiscard]] and [[fallthrough]] are based (and of the families of related non-standard techniques) being used in the wild. One must merely perform a GitHub code search at github.com/search, or an OpenHUB code.openhub.net search to see these. (Keywords “unused”, “warn_unused_result”, “fallthrough”.)

Design Philosophy

We are not proposing standardization or normalization of “warnings”. As per the existing attributes, all three proposed attributes merely serve as possible hints to an implementation, with a little non-normative guidance on what they intend. There is nothing novel in their proposal, and they are mainly non-prescriptive standardization of extensive existing practices.

We have intentionally erred on the side of implementation freedom. Ultimately, the purpose of warnings is to catch mistakes, but there are many times when an implementation cannot definitively determine whether a piece of code is intentional or accidental. The algorithms implementations use to determine when it is worth issuing a warning in such cases are arbitrarily complex, so we would advocate, in general, to leave them as a QoI issue. The goal of proposing these popular attributes is simply to provide some portable vocabulary to express intents - that can then be leveraged in an implementation-defined fashion by these algorithms.

Wording

7.6.6 Unused attribute [dcl.attr.unused]

1. The attribute-token **`unused`** can be used to mark various names, entities and expression statements that may be intentionally not used.   [Note: If an implementation would have otherwise emitted a warning about an entity, so marked, not being used, they are encouraged not to. -- end note] [Note: Implementations are discouraged from emitting a warning if an entity marked **`unused`**, is used. -- end note] It shall appear at most once in each attribute-list, with no attribute-argument-clause.
2. The attribute may be applied to the declaration of a class, a typedef-name, a variable, a non-static data member, a function, an enumeration, a template specialization, or a non-null expression statement.
3. A name or entity declared without the **`unused`** attribute can later be re-declared with the attribute and visa-versa. An entity is considered marked after the first declaration that is marked is analyzed, and for the remainder of translation of the current translation unit.
4. When an expression statement is marked **`unused,`** it indicates that the expression is intentionally a discarded-value expression. [Note: If an implementation would have otherwise emitted a warning due to a **`nodiscard`** attribute, they are encouraged not to. -- end note]

7.6.7 Nodiscard attribute [dcl.attr.nodiscard]

1. The attribute-token **`nodiscard`** can be used to mark a function, a function template specialization or a type.

2. A *nodiscard call* is a function call expression, other than an assignment or compound assignment, that:
    a. is to a function marked **`nodiscard`**, or
    b. is to an instantiation of a function template specialization marked **`nodiscard`**, or
    c. returns a type marked **`nodiscard`**.
3. Appearance of a nodiscard call as a discarded-value expression is discouraged. [Note: This is typically because discarding the return value of a nodiscard call has surprising consequences. Implementations are encouraged to issue a warning, unless the nodiscard call is marked **`unused.`** --end note]

7.6.8 Fallthrough attribute [dcl.attr.fallthrough]

1. A null statement marked with the attribute-token **`fallthrough`**, is a *fallthrough statement*. The fallthrough attribute-token shall appear at most once in each attribute-list, with no attribute-argument-clause.
2. A fallthrough statement may appear within an enclosing switch statement, on some path of execution immediately between a preceding statement and a succeeding case-labeled statement.
3. [Note: If an implementation would have otherwise issued a warning about implicit fall through on a path of execution immediately after a fallthrough statement, they are encouraged not to. -- end note]

FAQ

1. Why does [[fallthrough]] need a trailing semi-colon? Why doesn’t it annotate the case label?

On 2015–09–09, at 7:41 AM, Richard Smith <richard@metafoo.co.uk> wrote:

> The argument when we designed [[fallthrough]] was that [[fallthrough]] doesn't notionally appertain to the label -- it appertains to the *preceding* sequence of labelled statements. Note that when you have a sequence of case labels with no intervening statements, it allows fallthrough through all of them, so it doesn't meaningfully apply to just one label. Also observe Example 3, where the fallthrough within the 'case 55:' block is not even immediately lexically preceding a case label.
>
> We viewed [[fallthrough]] as being more of a flow control keyword (being provided as an extension) than a source annotation, and from that perspective it made sense for it to be a new kind of statement (like a break statement or continue statement). (This also allows source compatibility with existing systems that already have such a keyword -- see for instance the "__fallthrough;" statement provided by MS SAL, which can be implemented with this proposal as "#define __fallthrough [[fallthrough]]", but cannot be implemented with a label attribute.)

On 2015–09–09, at 8:48 AM, Andrew Tomazos <andrewtomazos@gmail.com> wrote:

Consider the following:

```
switch (n) {
case 2:
  if (c1) {
    f();
    break;
  } else if (c2) {
    g();  // WARNING: no fallthrough statement
  } else if (c3) {
    h();
    break;
  } else if (c4) {
    g();
    h();
    [[fallthrough]];
  }
case 3:
  h();
}
```

This can be addressed with this:

```
switch (n) {
case 2:
  if (c1) {
    f();
    break;
  } else if (c2) {
    g();
    break; // <---- bug fixed
  } else if (c3) {
    h();
    break;
  } else if (c4) {
    g();
    h();
    [[fallthrough]];
  }
case 3:
  h();
}
```

or this:

```
switch (n) {
  case 2:
    if (c1) {
      f();
      break;
    } else if (c2) {
      g();
      [[fallthrough]]; // <--- nope, intentional
    } else if (c3) {
      h();
      break;
    } else if (c4) {
      g();
      h();
      [[fallthrough]];
    }
  case 3:
    h();
  }
```

Had we specified fallthrough as appertaining to the label, and not as statements, this bug would have been missed. Or rather, the programmer would not be able to express the fallthrough intent of each block individually within the if-else chain as shown above.

fallthrough appertains to points on one or more paths of execution. It can be thought of as an assertion statement that the next thing that will happen at run-time is the next case block will be executed. (This assertion is checked statically at compile-time.)

2. Why is [[nodiscard]] being proposed as an attribute and not a context-sensitive-keyword? Why doesn't nodiscard make the program ill-formed?

We have considered three different options in the design process of nodiscard:

(1) A [[nodiscard]] attribute that generates a warning, like [[deprecated]]:

```
[[nodiscard]] int f();
[[nodiscard]] struct S { ... }
S g();

int main() {
  f(); // WARNING
  g(); // WARNING
}
```

(2) A [[nodiscard]] attribute that causes ill-formed, no diagnistic required, like [[noreturn]]

```
[[nodiscard]] int f();
[[nodiscard]] struct S { ... };
S g();

int main() {
  f(); // UNDEFINED BEHAVIOUR
  g(); // UNDEFINED BEHAVIOUR
}
```

(Note that “ill-formed no diagnostic required” and “undefined behaviour” are normatively synonyms, they both place no requirements on the implementation with respect to the enclosing program.)

(3) A nodiscard context-sensitive keyword that causes ill-formed, diagnostic required - like override:

```
int f() nodiscard;
struct S nodiscard { ... };
S g();

int main() {
  f(); // ERROR
  g(); // ERROR
}
```

After careful deliberation we decided on proposing 1 with the following rationale:

The existing practice demonstrates there are cases when the programmer intentionally wants to discard the result of a nodiscard function, even though in most cases they do not. The existing nodiscard is a hint from the function designer to the function user, that immediately destroying the result is most likely not what you want, but it isn’t a straight-jacket and isn’t used as such.

In the intentional case, under option 1, the implementation is encouraged to emit a warning, but the semantics of the program remain untouched. The return value is destroyed at the end of the statement in well-defined order.

In the intentional case, under option 2, the program could potentially have arbitrary unexpected consequences. Undefined behaviour is not allowed in many codebases. Some consider undefined behaviour a semantic effect and not in spirit with the intended use of attributes.

In the intentional case, under option 3, the program is ill-formed and won't compile. The programmer is strictly denied what they want to do. The [[unused]] attribute could not be used to strictly suppress it, as that would be a semantic effect.

3. What constitutes an entity being used?

As per the existing appearance of the term "used" in [dcl.attr.deprecated], this is unspecified and hence left as a quality of implementation issue. There are a spectrum of increasingly complex algorithms an implementation could use to statically analyze a little-used entity in a given program, and to take an educated guess as to whether it likely enough to be indicative of an logic error to issue a warning. We feel it would be onerous and unnecessarily restrictive on implementations to strictly specify a particular algorithm.

4. Why do you discourage implementations from emitting a warning if an [[unused]] entity is used?

This is in line with existing practice. It shows that many times the [[unused]] annotation is used where a certain set of defines leads to an entity appearing unused for one preprocessed translation unit, as typified by the assert-NDEBUG case. If the implementation warned when an [[unused]]-marked entity was used, this would trigger warnings when the other set of defines were used:

ie If an implementation warned about an [[unused]]-marked entity being used, then without NDEBUG the following would generate a warning:

```
std::pair<int, int> flatten(int x, int y [[unused]], int z) {
  assert(y == 0); // WARNING: y used ???
  return {x, z};
}
```

This would of course defeat the purpose. The semantic of [[unused]] is that the entity MAY appear unused, not that it MUST be unused.

It would be very difficult for an implementation to statically analyze a translation unit under all possible results of preprocessing.

(Please note that there are millions of examples of unused. Even if this category typifies 90% of them, we are still left with 100,000+ examples.)