Wording for [[nodiscard]] attribute.

Document No.: P0189R0
Revises: P0068R0 In Part
Project: Programming Language C++ - Evolution
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2016-01-03

Summary

A wording for the [[nodiscard]] attribute described in P0068R0 is proposed for application to the C++17 working draft, with modifications based on Kona EWG feedback. [[nodiscard]] marks functions and return types where discarding the return value has surprising consequences. It has heavy use in existing practice. Kona EWG voted SF=10, F=6, N=0, A=2, SA=0 in favor of [[nodiscard]] from P0068R0. See P0068R0 for detailed motivation/rationale.

Changes From P0068R0

As per Kona EWG change request, we replaced a facility where a warning about a nodiscard call could be suppressed with [[unused]] to suppressing it with the existing practice of explicitly casting to void.

Wording

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. It shall appear at most once in each attribute-list, with no attribute-argument-clause.
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 if it is not explicitly cast to void. [Note: Implementations are encouraged to issue a warning in such cases. This is typically because discarding the return value of a nodiscard call has surprising consequences. --end note]

Example

```
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
}
```

FAQ

1. 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 diagnostic 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 revoke any and all 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.

This design decision was reviewed and voted on at Kona EWG, and the decision was strongly upheld.