

A nice placeholder with no name

Document #: P2169R0
Date: 2020-05-17
Project: Programming Language C++
Audience: EWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>
Michael Park <mcypark@gmail.com>

Abstract

In [P1110][?] Jeffrey Yesskin and JF Bastien explore the design space of placeholders with the goal of not naming entities for which a name would provide no additional information. In this paper, we propose a concrete solution for the use of `_` (U+005F _ LOW LINE) as such placeholder for variable declarations and pattern matching textbf in a fully backward compatible manner.

Tony tables

Before	After
<pre>std::lock_guard namingIsHard(mutex); // Structured binding [[maybe_unused]] auto [x, y, iDontCare] = f(); // Pattern matching inspect(foo) { __ => bar; };</pre>	<pre>std::lock_guard _(mutex); // Structured binding auto [x, y, _] = f(); // Pattern matching inspect(foo) { _ => bar; };</pre>

Motivation

Both [P1110][?], [P1469][?] and [P0577][?] go over the motivation both for a placeholder syntax in general and the single underscore `_`:

- Some variable, like locks, `scope_guard` and other RAII objects are only used for their side effects.

```
std::lock_guard lock(mutex);
```

- Some structured bindings may not be used, and we would like a syntax to specify that this is the case. Introducing names for variables that are not used to convey less meaning than `_`, which convey the "I don't care meaning". It is furthermore not possible to apply attributes to structured bindings and as such, it is not possible to carry the developer's intent to the compiler. The current wording advises:

Recommended practice: For an entity marked `[[maybe_unused]]`, implementations should not emit a warning that the entity or its structured bindings (if any) are used or unused. For a structured binding declaration not marked `[[maybe_unused]]`, implementations should not emit such a warning unless all of its structured bindings are unused.

- We need a wildcard token for pattern matching. The proposed `_` is, as explained by [P1469][?] an industry wide-standard and we believe having a single token `_` for both wildcards and placeholders contribute to make C++ more consistent with itself and with other languages (making easier to teach, etc)

Design considerations

The design as presented in this paper takes the following design parameters into consideration

- `_` is recognized as some kind of placeholder in many existing languages, including go, python, rust, C#.
- `_` is used in existing code and popular frameworks.
- `_` might be defined as a macro function in some files depending on the `gettext` library.
- `_` already carries the meaning of "I don't want to use this variable". Variables that developers care about have more meaningful names.
- `_` unnamed variables cannot have linkage.
- `_` and `__` are similar enough that we should not give meaning to both.
- Tokens are rare, using a different token for placeholder may needlessly restrict the design space for more important features.
- For consistency, we would like to use the same token as the pattern matching placeholder.

Proposal

We propose that when `_` is used as the identifier for the declaration of a variable, function parameter name, lambda capture or structured binding. the introduced name is implicitly given the `[[maybe_unused]]` attribute.

[*Example:*

```
auto _ = foo(); // equivalent to [[maybe_unused]] auto _ = f();
```

— *end example*]

When a variable, function parameter structured binding or lambda capture is introduced by the identifier `_` at function scope, or at namespace scope within the purview of a module implementation unit, if it would redefine an existing declaration, [basic.scope.declarative], it

instead defines a variable with an implementation-defined name which does not participate in name lookup

[*Example:*

```
namespace a {
    auto _ = f(); // Ok, declare a variable "_"
    auto _ = f(); // error: "_" is already defined in this namespace scope
}
void f() {
    auto _ = 42; // Ok, declare a variable "_"
    auto _ = 0; // Ok, declare a variable with no name
    {
        auto _ = 1; // Ok, shadowing
        assert( _ == 1 ); // Ok
    }
    assert( _ == 42 ); // Ok
}
```

— *end example*]

In contexts where the grammar expects a pattern matching pattern, `_` represents the wildcard pattern.

```
inspect(i) {
    0 => 0;
    _ => 1; // wildcard pattern
};
```

Alternative Design

Tony van Eerd suggested that **use** of a variable named `_` could be made ill-formed after the introduction of a placeholder variable in the same scope. This has been implemented as a warning [Compiler Explorer](#).

Pattern matching

Citing [P1371R0][?], `_` can be used as a pattern matching wildcard without ambiguities

Even though `_` is a valid identifier, it does not introduce a name as doing so would result in redeclaration errors in the case where multiple wildcard `_` identifiers are used. It is possible for users to have already introduced `_` as a type or variable name in the same scope where an `inspect` statement is used. As a highly-visible example, the authors are aware of the use of `_` as a name in the popular “Google Mock” library. Idiomatically this is accessed by introducing `_` into the current namespace or block scope with a `using` declaration. Using the wildcard pattern in cases like this is unambiguous since the expression pattern requires a `^` introducer for primary expressions. `_` will always match against an existing name and `_` will always represent the wildcard pattern. An existing `_` name can be used without ambiguity in the matched statement to which control is passed. Naturally, the impact of

defining `_` in the pre-processor cannot be predicted or controlled by this paper and is thus liable to result in an ill-formed program.

Impact on P2011 and other pipeline rewrite musing

Placeholders in pipeline rewrite have a "Bind to this" semantic rather than a "I don't care" semantic. As such, these proposals could use a different syntax.

It might also be possible to have `_` and `_1`, ... contextually be used as bind placeholders in pipeline require contexts.

Impact on existing code

Google Mock

Google Mock appear to constitute the most common use of the `identifier`. With this proposal, Google Mock continues to work. Variables named `_` may shadow the `testing::_` global variable introduced by google mock:

```
namespace testing {  
    const internal::AnythingMatcher _ = {};  
}
```

Because the proposal limits the use of placeholder to function scope (because of linker considerations), there is little risk that increased use of `_` causes shadowing issues for users of this framework.

It is possible to use the feature proposed in this paper along with google mock as long as the using `namespace testing;` appears before any declaration of `_` in that code.

[Compiler explorer link](#)

Gettext

Some projects using Gettext define `_` as a function-like macro.

```
# define _(msgid) gettext (msgid)
```

It is important to know that this is not provided in any gettext header. It is also technically undefined as the `_` is reserved at global scope. But, regardless, defining this macro would not prevent the use of the proposed syntax, as `_` is only expanded by the preprocessor in this case if followed by parentheses:

```
constexpr const char* gettext(int) { return nullptr;}  
#define _(msgid) gettext (msgid)  
  
int main() {  
    constexpr auto _ = _(42);  
    auto _ = 42;  
}
```

```
static_assert(_ == nullptr);  
}
```

[Compiler explorer link](#)

Implementation

The proposal has been implemented in Clang and the implementation is available on [Compiler Explorer](#).

The implementation consists of ignoring existing `_` variables in the same scope when declaring placeholder variables. Note that name lookup is not affected by this proposal, some variables are simply skipped over based on their name. The variables otherwise conserve their name for diagnostics purposes.

It is possible to warn about uses of variables called `_` [Compiler Explorer](#).

Wording

// TODO

Rebuttal some wilder ideas explored by P1110

[P1110] explores the different places where a placeholder may be used.

Enums

The code on the left (P1110) can be written in standard C++ in a way that expresses the intent as clearly, if not more.

```
enum MmapBits {  
    Shared,  
    Private,  
    —,  
    —,  
    Fixed,  
    Rename,  
    //...  
};
```

```
enum MmapBits {  
    Shared,  
    Private,  
    Fixed = Private + 2,  
    Rename,  
    //...  
};
```

Concept-constrained declarations

The following example from P1110 is a historical artifact as C++20 uniformizes `auto` for this purpose.

```
template<Integral __ N> class integral_constant { ... };
Numeric multiplyAdd(Numeric __ x, Numeric __ y, Numeric __ z) {
    Numeric __ multiplied = x * y;
    return multiplied + z;
}
```

Here is the C++20 equivalent

```
template<Integral auto N> class integral_constant { ... };
Numeric multiplyAdd(Numeric auto x, Numeric auto y, Numeric auto z) {
    Numeric auto multiplied = x * y;
    return multiplied + z;
}
```

Other explored areas, including placeholders for function names, type declaration, using and concepts are not useful, either because C++ does already provide the desired facilities such as anonymous type, or because the entity needs a name (concepts, functions and using declarations without a name cannot be used)

Acknowledgments

Thanks to Miro Knejp and Tony Van Eerd for providing valuable feedback on very short notice!

References

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++*
<https://wg21.link/N4861>