Title:	This Variable Should Not Be Named
Document number:	<unassigned>
Date:	Yyyy-mm-dd
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	Evolution Working Group (EWG)
Reply-to:	Alberto Barbati <ganesh@barbati.net>

1 Introduction

The special identifier `__` (double underscore) is proposed as a universal identifier for all variables that need not have a name or that the programmer deliberately don't want to name.

2 Motivation and Scope

There are at least two scenarios where a universal identifier is useful: RAII idioms and decomposition declarations.

2.1 RAII idiom

The typical RAII idiom requires the programmer to provide a name for a local variable that is never used elsewhere in the lexical scope where it is defined. For example:

```
void f()
{
    unique_lock lock {mutex};
    // ... name "lock" not used here
}
```

If multiple resources are acquired in the same lexical scope, the programmer has to invent different names for each variable needed and sometimes good names are hard to find… Consider this example taken from paper P0052R3, simplified by replacing the `make_scope_xxx` factory functions with deduced constructors:

```
void demo_scope_exit_fail_success()
{
    std::ostringstream out {};
    auto lam = [&]{out << "called ";};
    try{
        scope_exit    v {[&]{out << "always ";}};
        scope_success w {[&]{out << "not ";}};
        scope_fail    x {lam};
        throw 42;
    } catch(...) {
        scope_fail    y {[&]{out << "not ";}};
        scope_success z {[&]{out << "handled";}};
    }
    ASSERT_EQUAL("called always handled", out.str());
}
```

Notice how the presence of the unnecessary variable names is unnecessary and even distracting. This is the same example, rewritten with this proposal:

```
void demo_scope_exit_fail_success()
{
    std::ostringstream out {};
    auto lam = [&]{out << "called ";};
    try{
        scope_exit    __ {[&]{out << "always ";}};
        scope_success __ {[&]{out << "not ";}};
        scope_fail    __ {lam};
        throw 42;
    } catch(...) {
        scope_fail    __ {[&]{out << "not ";}};
        scope_success __ {[&]{out << "handled";}};
    }
    ASSERT_EQUAL("called always handled", out.str());
}
```

The essential part of this proposal is that each occurrence of the special identifier `__` is considered as if it were a new unique identifier.

2.2 Decomposition declarations

When using a decomposition declaration, the programmer is currently required to provide an identifier for every element of the initializer. If, for example, some elements are not used in the lexical scope, having the programmer specify unique names is a waste of creativity, especially in case there's a risk to create duplicates.

However, it's more that just a matter of choosing names. A bigger problem is that the compiler has all rights to complain that the unwanted element is actually never used[1]. By using a special identifier, we get two birds with a stone: not only the `__` identifier can be safely and even repeatedly used (it could be used even more than once in the same decomposition declaration!), but the special identifier is *per se* a valid hint[2] that the compiler can safely apply the "maybe unused" semantic even in absence of an explicit attribute.

3 Impact On the Standard

This proposal is a language extension. It provides a special meaning for an identifier whose use is already reserved to the implementation for all uses (see [lex.name]), but is not otherwise used by the current Standard.

Although the proposal should not break user code, the change might still affect library code, however the impact is believed to be extremely limited. A search of the identifier `__` in libc++ 4.0 reveals that it's used only two times in the header `<future>`: in both cases it's an instance of the RAII idiom where the variable is not subsequently used, exactly as in the first of our motivating examples! A search for the identifier in the Visual Studio 2015 standard headers shows no uses at all.

4 Design Decisions

The only case in the C++ language where a variable name can be omitted is in the declaration of function parameters. In the case of regular non-decomposition declarators, the grammar requires the presence of an identifier and omitting the identifier produces a valid code with a different meaning:

---

[1] It is not currently possible to specify attributes for the individual elements of a decomposition declaration, thus it's not possible to use the `[[maybe_unused]]` attribute to silence the compiler.

[2] In fact, since all occurrences of the `__` identifiers are considered unique, the variable cannot effectively be referenced in any way after the point of declaration.

```
Type x { value }; // declaration of x with initializer
Type { value }; // functional-style cast, whose value is discarded
```

As we cannot simply omit the name, we have only two choices: either we provide a special identifier or we introduce a new syntax. Such a syntax is actually been proposed in paper P0577R0.

In the case of decomposition declarations, we might probably amend the syntax to allow omitting the unwanted identifiers entirely. However, the author believes that allowing multiple consecutive commas would make the code less readable:

```
auto [x, , z] = expr; // not proposed: double comma is easy to miss
auto [x, __, z] = expr; // proposed: ignored element is explicit
```

The identifier `__` has been chosen since:

a) it is a valid identifier in the current grammar;

b) it is already reserved for all uses and it's likely that implementations are not using it extensively;

c) it's short and inconspicuous, yet not totally "invisible";

d) the absence of alphanumeric characters makes it clear it is not "a name".

5 Technical Specifications

tdb

6 Open Questions

6.1 Is the feature implementable?

The author believes the feature is easily implementable. Actually, a naive implementation based on the C processor might be provided for every compilers featuring a pre-defined macro like `__COUNTER__` (among others, Visual Studio and Clang have it). However, such an implementation has the following drawbacks:

- it might introduce ODR violations in case the special identifier is used inside an inline function;
- it won't apply the "maybe unused" semantic.

7 References

P0052R3 Generic Scope Guard and RAII Wrapper for the Standard Library

P0577R0 Keep that Temporary!

8 Acknowledgements