Doc. No. P4348R0
Date: 2026-08-12
Audience: EWG, SG23
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

Types and Attributes

Bjarne Stroustrup (www.stroustrup.com)
Columbia University

Abstract

Attributes are not part of types. There are solid reasons for that, notably that ABIs would have to change. There are also disadvantages, notably some code is more complex than if an attribute was part of the type to which it was applied. This choice is important for the implementation and use of attributes used in profiles to guarantee type safety (e.g., to correctly distinguish between properly initialized objects and uninitialized memory). This paper shows that we can manage either way, using **[[uninit]]** and **[[ref_to_uninit]]** from the initialization profile as examples. Therefore, we recommend "no change" unless more compelling examples are found.

Introduction

The most important examples involve function calls that pass a pointer to initialized or uninitialized memory. Marking an argument with **[[ref_to_uninit]]** means that that argument must refer to uninitialized memory. If not, the initialization profile requires that that an argument refers to an initialized object. Consider

```
int x [[uninit]];
using T = decltype(x);          // does T accept a pointer to uninitialized memory?
int f(T* [[ref_to_uninit]]);
int f(T*);
f(&x);                          // which f() is called?
```

Is **T** acknowledged to refer to uninitialized memory or not?

- If the answer is "no", **f(&x)** will **call f(T*)**. As far as I can tell, that's status quo.
- If the answer is "yes", **(&x)** will **call f(T* [[ref_to_uninit]])**, and we need to redefine the type system to accommodate this.

In either case, if **[[ref_to_uninit]]** is ignored (because it is an unknown attribute) or suppressed by a profile, both declarations of **f()** will refer to the same function.

Note that a declaration like **int f(int [[uninit]])** is meaningless because its argument are copied.

In either case, we must modify the ABIs to reflect the use of attributes on function arguments. This implies that a program relying on **[[ref_to_uninit]]** (or any other attribute used similarly) to resolve overloading cannot be linked to a program compiled with an older compiler. That doesn't affect older code not knowing about **[[uninit]]** or code for which **[[uninit]]** is suppressed.

In [dcl.type] the standard says

> The optional *attribute-specifier-seq* in a *type-specifier-seq* or a *defining-type-specifier-seq* appertains to the type denoted by the preceding *type-specifier*s or *defining-type-specifier*s ([dcl.meaning]). The *attribute-specifier-seq* affects the type only for the declaration it appears in, not other declarations involving the same type.

That seems to be a clear "no" but current implementations diverge from that in their own attribute systems (e.g., Attributes in Clang — Clang 24.0.0git documentation).

For compatibility, it seems clear that the answer must be "no", but that implies some inconveniences (problem/surprises) when used. Otherwise, there would be no point in discussing these alternatives. Below, I will present the implication of each alternative followed by the conclusion: "no change."

Assume that an attribute cannot be part of a type

Because an attribute is not part of a type

- An alias containing an attribute doesn't transmit the attribute to its users

```
using UI = int [[uninit]];
UI x;          // an error if the programmer thought UI implied [[uninit]]
UI y = 7;      // an error if the programmer thought UI implied [[uninit]]
```

Instead, we must use an attribute, such as **[[uninit]],** consistently. If we do that, we can

- specify the information at both the call and implementation point for overloaded functions:

```
// somewhere:
int f(int x [[ref_to_uninit]]) { x = 7; /* … */ }
int f(int x) { int y = x; /* … */ }
```

```
// call environment:
int f(int [[ref_to_uninit]]);
int f(int);

int x = 7;
f(&x);  // call the f() specified to assume its argument to be initialized
int y [[uninit]];
f(x&y); // call the f() specified to assume its argument to be uninitialized
```

- specify the information at both the call and implementation point for template functions

```
// somewhere:
template<class T> T f(T* p) { auto y = *p; /* … */ }
template<class T> T f(T* p [[ref_to_uninit]]) { construct_at(p,y); /* … */ }

// call environment:
 template<class T> T f(T*);
template<class T> T f(T* [[ref_to_uninit]]);
int x [[uninit]];
f(&x);  // call the f() specified to assume its argument to be uninitialized
x = 7;
f(&x);  // call the f() specified to assume its argument to be initialized
```

For this to work, a compiler must

- Remember the attribute locally (in the scope of the function in which it is used).
- Maintain the usual link information for functions not using an attribute like **[[ref_to_uninit]]**.
- Use different link information for functions using an attribute, like **[[ref_to_uninit]]**.

Error handling

It is a principle that “a program that passes a profile will compile with the same meaning if compiled with the profile suppressed.” That is a slight simplification. We have to add: The program with the profile suppressed will compile only if the support for the profile is still present otherwise the program will fail to compile.

Now consider the case of an implementation that doesn’t enforce the **[[uninit]]** and **[[ref_to_uninit]]** attributes, such as an older compiler. The result of ignoring such attributes are:

1. The exact same code being generated, or
2. A compile-time error, or

3. A linker-time error

If there are no calls of overload resolution (incl. function templates using concept) that use **[[ref_to_uninit]]** we have [1].

To get different semantics, there must be two different functions of the same name invoked, one with **[[ref_to_uninit]]** and one without. Suppress **[[ref_to_uninit]]** and we get a double definition. To use their arguments correctly, those two functions must be different.

If the function definitions are in the same scope, we get a compile-time double definition error [2].

If the function definitions are in separate TUs, we get a link-time error [3].

Assume that an attribute can be part of a type

The examples in the previous section, didn't (and couldn't under the assumption there) rely on an alias containing an attribute, so they have exactly the same meaning under the assumption here.

If an attribute becomes part of a type, we can write some code more elegantly. For example, in addition to the examples above, we could write:

```
using UI = int [[uninit]];

// somewhere:
int f(int* p) { auto y = *p; /* … */ }
int  f(UI* p) { construct_at(p,y); /* … */ }

// call environment:
template<class T> T f(T);
int x = 7;
 f(&x);  // call the f() specified to assume its argument to be initialized
UI y;    // UI means that we can leave y uninitialized
f(&y);   // call the f() specified to assume its argument to be uninitialized
```

However, we don't get anything we couldn't achieve without embedding **[[uninit]]** in a type. This implies that we could adopt the current meaning for now and expand use of attributes later if we find the need. Since I expect the use of **[[uninit]]** and **[[ref_to_uninit]]** to be rather limited and specialized (compared to the immense amount of C++ code). I expect we can manage for quite a while, so that the decision about whether to embed attributes in types can be

postponed until we have experience with more attributes. So, I am not proposing to make attributes part of types.

WP wording

Here we assume that an attribute is not part of the type.

This text needs review by a wording expert.

A function enforcing **[[ref_to_uninit]]** or **[[must_init]]** for an argument is a different function from an otherwise identical function ignoring or suppressing the **[[ref_to_uninit]]** or **[[must_init]]**.

When enforcing **[[ref_to_uninit]]** or **[[must_init]]**

- A function argument marked **[[ref_to_uninit]]** or **[[must_init]]** can accept only an argument referring to something marked **[[uninit]]**.
- A function argument not marked **[[ref_to_uninit]]** or **[[must_init]]** cannot accept an argument referring to something not marked **[[uninit]]**.

Acknowledgement

Thanks to Gabriel Dos Reis for consistently pointing to this problem with attributes and types.

References

Bjarne Stroustrup: An initialization profile (R1). P4222R1. 2026-08-12.