Proposed resolution for **CWG1223**

P2915R0
Corentin Jabot, 13 June 2023
CWG

Because the restriction that a trailing-return-type can appear only in a declaration with “the single type-specifier auto” is a semantic, not a syntactic restriction, it does not influence disambiguation, which is “purely syntactic”. Consequently, some previously unambiguous expressions are now ambiguous.

This problem is made more apparent by P0849R8 "auto(x) decay copy in the language", which can make auto(x) a valid expression.

The proposed wording specifies that something that looks like a declaration with a trailing return type is only a declaration if it starts with auto.

**Wording**

Modify [stmt.ambig]/p1 as follow

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a `. In those cases the statement is a declaration, except as specified below.

Add at the end of [stmt.ambig] after Example 3

A syntactically ambiguous statement that can syntactically be a declaration with an outermost declarator with a trailing-return-type is a declaration only if it starts with auto.

**Example:**

```c
struct M;
struct S {
    S* operator()();
    int N;
    int M;

    void mem(S s) {
        auto(s)→M; // S::M hides ::M, this is an expression
```
The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in [stmt.ambig] can also occur in the context of a declaration. In that context, the choice is between an object declaration with a function-style cast as the initializer and a declaration involving a function declarator with a redundant set of parentheses around a parameter name. Just as for the ambiguities mentioned in [stmt.ambig], the resolution is to consider any construct, such as the potential parameter declaration, that could possibly be a declaration to be a declaration. However, a construct that can syntactically be a declaration whose outermost declarator would match the grammar of a declarator with a trailing-return-type is a declaration only if it starts with auto.

[Example 1:

```c
struct S {
  S(int);
};

typedef struct BB { int C[2]; } *B, C;
```

```c
void foo(double a) {
  S w(int(a));       // function declaration
  S x(int());        // function declaration
  S y((int(a)));     // object declaration
  S y((int)a);       // object declaration
  S z = int(a);      // object declaration
  S a(B())->C);     // object declaration
  S b(auto())->C);  // function declaration
```
An ambiguity can arise from the similarity between a function-style cast and a type-id. The resolution is that any construct that could possibly be a type-id in its syntactic context shall be considered a type-id. However, a construct that can syntactically be a type-id whose outermost abstract-declarator would match the grammar of an abstract-declarator with a trailing-return-type is a type-id only if it starts with auto.

```cpp
template <class T> struct X {};  
template <int N> struct Y {};  
X<int()> a;  // type-id  
X<int(1)> b;  // expression (ill-formed)  
Y<int()> c;  // type-id (ill-formed)  
Y<int(1)> d;  // expression

void foo(signed char a) {  
  sizeof(int());  // type-id (ill-formed)  
  sizeof(int(a));  // expression  
  sizeof(int(unsigned(a))));  // type-id (ill-formed)  

  (int())+1;  // type-id (ill-formed)  
  (int(a))+1;  // expression  
  (int(unsigned(a)))+1;  // type-id (ill-formed)
}
typedef struct BB { int C[2]; } *B, C;  
void g() {  
  sizeof(B ()→C[1]); // OK, sizeof(expression)  
  sizeof(auto () → C[1]); // error: sizeof of a function returning an array
}
```