Aggregates Having Default Member Initializers

C++14 enables the use of **aggregate initialization** with classes employing default member initializers.

Description

~~Prior to~~ C++~~14~~, classes that used default member initializers, i.e., initializers that appear directly within the scope of the class (see Section 2.1."Default Member Init" on page 318), were not considered **aggregate** types:

```
struct S                // aggregate type in C++14 but not C++11
{
    int i;
    bool b = false;     // uses default member initializer
};

struct A                // aggregate type in C++11 and C++14
{
    int  i;
    bool b;             // does not use default member initializer
};
```

Because `A` but not `S` is considered an aggregate in C++11, instances of `A` can be created via aggregate initialization, whereas instances of `S` cannot:

```
A a={100, true};  // OK, in both C++11 and C++14
S s={100, true};  // Error, in C++11; OK, in C++14
```

Note that since C++11, direct list initialization can be used to perform aggregate initialization; see Section 2.1."Braced Init" on page 215:

```
A a{100, true};  // OK in both C++11 and C++14 but not in C++03
```

As of C++14, the requirements for a type to be categorized as an aggregate are relaxed, allowing classes employing default member initializers to be considered as such; hence, both `A` and `S` are considered aggregates in C++14 and eligible for aggregate initialization:

```
void f()
{
    S s0{100, true};        // OK in C++14 but not in C++11
    assert(s0.i == 100);    // set via explicit aggregate initialization
    assert(s0.b == true);   // set via explicit aggregate initialization

    S s1{456};              // OK in C++14 but not in C++11
    assert(s1.i == 456);    // set via explicit aggregate initialization
    assert(s1.b == false);  // set via default member initializer
}
```