Integrating default member initialization with braced initialization

Another new feature for C++11 is *default member initializers* for data members in a class; see Section 2.1."Default Member Init" on page 318. This new syntax supports both copy list initialization and direct list initialization. However, initialization with parentheses is not permitted in this context:

```
struct S
{
    int i = { 13 };

    S() { }                        // OK, i == 13.
    explicit S(int x) : i(x) { }  // OK, i == x.
};


struct W
{
    S a{};        // OK, by default i.i == 13.
    S b{42};      // OK, by default i.i == 42.
    S c = {42};   // Error, constructor for S is explicit.
    S d = S{42};  // OK, direct initialization of temporary for initializer
    S e(42);      // Error, fails to parse as a function declaration
    S f();        // OK, declares member function f
};
```

List initialization where the list itself is a single argument to a constructor

Another new form of initialization for C++11 is list initialization with a braced list of arguments to populate a container; see Section 2.1."`initializer_list`" on page 553. If a braced list contains arguments that are all of the same type, then the compiler will look for a constructor taking an argument of type `std::initializer_list<T>`, where `T` is that common type. Similarly, if a braced list of values can be implicitly converted to a common type, then a constructor for an `std::initializer_list` of that type will be preferred. When initializing from a nonempty braced-initializer list, a matching initializer list constructor always wins overload resolution. However, *value* initializing from a pair of empty braces will prefer a default constructor:

```
#include <initializer_list>  // std::initializer_list

struct S
{
  S() {}
  S(std::initializer_list<int>) {}
  S(int, int);
};
```