For any member m that has a default member initializer, constructors that don't initialize m in their member initializer list will implicitly initialize m by using the default member initializer value:

```
struct S2
{
    int d_i = 1;
    int d_j = 1;

    S2() { }                          // Initialize d_i with 1, d_j with 1.
    S2(int) : d_i(2) { }              // Initialize d_i with 2, d_j with 1.
    S2(int, int) : d_i(2), d_j(3) { } // Initialize d_i with 2, d_j with 3.
};
```

Note that initialization of all data members including those using the default member initializers happen in the order in which they are **declared** in the class **definition**. Accordingly, previously initialized nonstatic data members can be used in subsequent initializer **expressions**:

```
struct S4 {
    const char* d_s{"hello"};
    int         d_i{2};
    char        d_c{d_s[1]};  // OK, d_c initialized to d_s's second character

    S4()                      { }
    S4(const char* s) : d_s(s) { }
};

S4 s4d;             // OK, s4d.d_c initialized to 'e'
S4 s4v("goodbye");  // OK, s4v.d_c initialized to 'o'
```

The default member initializer, just like **member function** bodies and member initialization lists, ~~executes~~ in a **complete-class context**. Since the initializer sees its enclosing class as a **complete type**, it can therefore reference the size of the enclosing type and invoke member functions that have not yet been seen:

```
struct S5
{
    int d_a[4];
    int d_i = sizeof(S5) + seenBelow();  // OK
    int seenBelow();
};
```

Name lookup in default member initializers will find members of the enclosing class and its bases before looking in namespace scope: