```
    constexpr D1() { }  // Error, B has nonconstexpr default constructor
};

struct D2 : B  // public derivation
{
    int d_i;  // nonstatic, scalar data member
    constexpr D2(int i) : B(i) { }  // Error, doesn't initialize d_i
};

          int f1() { return 5; }  // nonconstexpr function
constexpr int f2() { return 5; }  //    constexpr function

struct D3 : C  // public derivation
{
    int d_i = f1();  // initialization using nonconstexpr function
    int d_j = f2();  // initialization using    constexpr function

    constexpr D3() { }  // Error, d_i not constant initialized

    constexpr D3(int i) : d_i(i) { }  // OK, d_i set from init list
};
```

The example code above illustrates various ways in which a base class or nonstatic data member might fail to be initialized by a constructor that is explicitly declared **constexpr**. In the final derived class, `D3`, we note that there are two data members, `d_i` and `d_j`, having member initializers that use a non**constexpr** function, `f1`, and a **constexpr** function, `f2`, respectively. The implementation of the **constexpr default constructor**, `D3()`, is erroneous because data member `d_i` would be initialized by the non**constexpr** function `f1` at run time. On the other hand, the implementation of the **value constructor**, `D3(int)`, is fine because the data member `d_i` is set in the member-initializer list, thereby enabling compile-time evaluation.

3. Defining a constructor to be **constexpr** requires that the class have no **virtual** base classes[5]:

```
struct B { constexpr B();  /*...*/ };  // some arbitrary base class

struct D : virtual B
{
    constexpr D(int) { }   // Error, class D has virtual base class B.
};
```

4. A constructor that is explicitly declared to be **constexpr** can always be suppressed using **= delete** (see Section 1.1."Deleted Functions" on page 53). Deleting a function

[5] C++20 removes the restriction that a constructor cannot be **constexpr** if the class has any virtual base classes.