```
struct D0 : B0       // D0 inherits publicly from B0.
{
    void f();        // OK, overrides void B0::f()
    void g();        // Error, void B0::g() is final.
    void g() const;  // OK, void B0::g() const is not final.
};
```

As the simple example above illustrates, decorating a virtual member function — e.g., `B::g()` — with **final** precludes overriding only that specific function signature. Note that when redeclaring a **final** function outside the class definition (e.g., to define the function), the **final** specifier is not permitted:

```
void B0::g() final { }  // Error, final not permitted outside class definition
void B0::g() { }        // OK
```

final on destructors

The use of **final** on a virtual destructor precludes inheritance entirely, as any derived class must have either an implicit or explicit destructor that will attempt to override the **final** base class destructor:

```
struct B1
{
    virtual ~B1() final;
};

struct D1a : B1 { };    // Error, implicitly tries to override B1::~B1()

struct D1b : B1
{
    virtual ~D1b() { }  // Error, explicitly tries to override B1::~B1()
};
```

Any attempt to suppress the destructor in the derived class, e.g., using `=`**delete** (see Section 1.1.“Deleted Functions” on page 53), will be in vain. If the intent is to suppress derivation entirely, a direct way would be to declare the type itself **final**; see ***final*** *user-defined types* on page 1011.

final pure virtual functions

Although declaring a **pure virtual function final** is allowed, doing so makes the type an **abstract class** and also prevents making any derived type a **concrete class**: