```
class Container { /*...*/ };

class ContainerIterator
{
    friend class Contianer;  // Compiles but wrong: ia should have been ai.
    // ...
};
```

The code above will compile and appear to be correct until `ContainerIterator` attempts to access a **`private`** or **`protected`** member of `Container`. At that point, the compiler will surprisingly produce an error. As of C++11, we have the option of preventing this mistake by using extended **`friend`** declarations:

```
class Container { /*...*/ };

class ContainerIterator
{
    friend Contianer;  // Error, Contianer not found
    // ...
};
```

Befriending a type alias used as a customization point

In C++03, the only option for friendship was to specify a particular **`class`** or **`struct`** when granting private access. Let's begin by considering a scenario in which we have an **in-process value-semantic type (VST)** that serves as a *handle* to a platform-specific object, such as a `Window` in a graphical application. (When used to qualify a VST, the term in-process, also called *in-core*, refers to a type that has typical value-type–like operations but does not refer to a value that is meaningful outside of the current process.[2]) Large parts of a codebase might seek to interact with `Window` objects without needing or obtaining access to the internal representation.

A small part of the codebase that handles platform-specific window management, however, needs privileged access to the internal representation of `Window`. One way to achieve this goal is to make the platform-specific `WindowManager` a **`friend`** of the `Window` class; however, see *Potential Pitfalls — Long-distance friendship* on page 1041:

```
class WindowManager;  // forward declaration enabling extended friend syntax

class Window
{
private:
```

[2]A discussion of this topic is planned for **lakos2a**, section 4.2.