```
struct P : E<S>  // Oops! should have been E(P) -- a serious latent defect
{
    int d_x;
    int d_y;
};

void test2()
{
    P p1; p1.d_x = 10; p1.d_y = 15;
    P p2; p2.d_x = 10; p2.d_y = 20;

    assert( !(p1 == p2) );  // Oops! This fails because of E(S) above.
}
```

Again, thanks to C++11’s extended **friend** syntax, we can defend against these defects at compile time simply by making the CRTP base class’s default constructor *private* and befriending its template parameter:

```
template <typename D>
class E
{
     E() = default;
     friend D;
};
```

Note that the goal here is not security but simply guarding against accidental typos, copy-paste errors, and other occasional human errors. By making this change, we will soon realize that there is no **operator**< defined for P.

Compile-time polymorphism using the curiously recurring template pattern Object-oriented programming provides certain flexibility that at times might be supererogatory. Here we will exploit the familiar domain of abstract/concrete shapes to demonstrate a mapping between runtime polymorphism using virtual functions and compile-time polymorphism using the CRTP. We begin with a simple abstract Shape class that implements a single, pure, virtual draw function:

```
class Shape
{
public:
    virtual void draw() const = 0;  // abstract draw function (interface)
};
```

From this abstract Shape class, we now derive two concrete shape types, Circle and Rectangle, each implementing the *abstract* draw function: