```
        int d_side2;
        int d_side3;

    public:
        Triangle(int side1, int side2, int side3)
            : d_side1(side1), d_side2(side2), d_side3(side3) { }

        void draw() const
        {
            std::cout << "Triangle(side1 = " << d_side1 << ", "
                                  "side2 = " << d_side2 << ", "
                                  "side3 = " << d_side3 << ")\n";
        }
    };
```

Unfortunately, we forgot to change the base-class type parameter when we copy-pasted from `Rectangle`.

Let's now create a new test that exercises all three and see what happens on our platform:

```
void test2()
{
    print(Circle(1));          // prints: Circle(radius = 1)
    print(Rectangle(2, 3));    // prints: Rectangle(length = 2, width = 3)
    print(Triangle(4, 5, 6));  // prints: Rectangle(length = 4, width = 5) ?!
    Shape<int> bug;            // Compiles?!
}
```

As should by now be clear, a defect in our `Triangle` implementation results in *hard* **undefined behavior** that could have been prevented at compile time by using the extended **`friend`** syntax. Had we defined the CRTP base-class template's default constructor to be *private* and made its type parameter a **`friend`**, we could have prevented the copy-paste error with `Triangle` and suppressed the ability to create a `Shape` object without deriving from it (e.g., see `bug` in the previous code snippet):

```
template <typename T>
class Shape
{
    Shape() = default;  // Default the default constructor to be private.
    friend T;           // Ensure only a type derived from T has access.
};
```

Generally, whenever we are using the CRTP, making just the default constructor of the base-class template **`private`** and having it befriend its type parameter is typically a trivial local change, is helpful in avoiding various forms of accidental misuse and is unlikely to