```
    A(const A&&) { ++s_count; }
    ~A()         { --s_count; }

    A& operator=(A&)  = default;  // see special members
    A& operator=(A&&) = default;  //  "     "       "
    // ...
};

int A::s_count;  // definition (in .cpp file)

class B { /* similar to A (above) */ };
// ...

void test()
{          // A::s_count = 0, B::s_count = 0
    A a1;  // A::s_count = 1, B::s_count = 0
    B b1;  // A::s_count = 1, B::s_count = 1
    A a2;  // A::s_count = 2, B::s_count = 1
}          // A::s_count = 0, B::s_count = 0
```

In this example, we have multiple classes, each repeating the same common machinery. Let's now explore how we might refactor this example using the CRTP:

```
template <typename T>
class InstanceCounter
{
protected:
    static int s_count;  // declaration

public:
    static int count() { return s_count; }
};

template <typename T>
int InstanceCounter<T>::s_count;  // definition (in same file as declaration)

struct A : InstanceCounter<A>
{
    A()          { ++s_count; }
    A(const A&)  { ++s_count; }
    A(const A&&) { ++s_count; }
    ~A()         { --s_count; }

    A& operator=(const A&) = default;
    A& operator=(A&&)      = default;
    // ...
};
```