spatial inefficiencies can become problematic at scale — especially when cache-friendliness is at a premium.

A solution to avoid this issue is to partially specialize `Tuple1` for one element in addition to the two existing specializations:

```
template <typename T>
class Tuple1<T>              // (3) specialization for one element
{
    T first;
    // ...
};
```

With this addition, `Tuple1<>` uses the full specialization (1), `Tuple1<int>` uses the partial specialization (3), and all instantiations with two or more types use the partial specialization (2). For example, `Tuple1<int, long, double>` instantiates specialization (2), which uses `Tuple1<long, double>` as a member, which in turn uses the partial specialization (3) for member `rest` of type `Tuple1<double>`.

The disadvantage of the design above is that it requires similar code in the `Tuple1<T>` partial specialization and the general definition, leading to a subtle form of code duplication. Having to write some redundant code might not seem especially problematic, and yet a good tuple API typically has considerable scaffold: `std::tuple`, for example, has 25 member functions.

Let's address `Tuple1`'s problem by using inheritance instead of composition, thus benefitting from an old and well-implemented C++ layout optimization known as the **empty-base optimization**. When a base of a class has no state, that base is allowed, under certain circumstances, to occupy no room at all in the derived class. Let's design a `Tuple2` variadic class template that takes advantage of the empty-base optimization:

```
template <typename... Ts>
class Tuple2;                // incomplete declaration

template <>
class Tuple2<>               // specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple2<T, Ts...>       // specialization for one or more elements
    : public Tuple2<Ts...>   // recurses in inheritance
{
    T first;
    //...
};
```

If we assess the size of `Tuple2<int>` with virtually any contemporary compiler, it is the same as `sizeof(int)`, so the base does not, in fact, add to the size of the complete object. One awkwardness with `Tuple2` is that with most compilers the types specified appear in the memory layout in reverse order; for example, in an object of type