```
    ImmutableString insert(size_type pos, const ImmutableString& s) const
    {
        std::string dataCopy(asStdString());    // Copy string from this object.
        dataCopy.insert(pos, s.asStdString());  // Do insert.
        return std::move(dataCopy);             // Move into return value.
    }

    const std::string& asStdString() const
    {
        return d_dataPtr ? *d_dataPtr : s_emptyString;
    }

    friend std::ostream& operator<<(std::ostream& os, const ImmutableString& s)
    {
        return os << s.asStdString();
    }
    // ...
};

const std::string ImmutableString::s_emptyString;
```

The internal representation of an `ImmutableString` is an `std::string` object allocated on the heap and accessed via an instantiation of the C++ Standard reference-counted smart pointer, `std::shared_ptr`. ~~The copy and move constructors and assignment operators are defaulted;~~ when an `ImmutableString` is copied or moved, only the smart pointer member is affected. Thus, even large string values can be copied in constant time.

The `insert` member function begins by making a copy of the *internal representation* of the immutable string. The copy is modified and then returned; the representation in the original `ImmutableString` is not modified:

```
void f1()
{
    ImmutableString is("hello world");
    std::cout << is << std::endl;                 // Print "hello world".
    std::cout << is.insert(5, ",") << std::endl;  // Print "hello, world".
    std::cout << is << std::endl;                 // Print "hello world".
}
```

Immutable types are often paired with *builder* classes — mutable types that are used to “build up” a value, which is then “frozen” into an object of the immutable type. Let’s define a `StringBuilder` class ~~with~~ mutating `append` and `erase` member functions that modify its internal state, and a conversion operator that returns an `ImmutableString` containing the built-up value:

```
class StringBuilder
{
    std::string d_string;
```