member function and thus will suppress the generation of other special member functions accordingly:

```
// example.h:

struct S6
{
    S6& operator=(const S6&);  // user-provided copy-assignment operator

    // suppresses the declaration of both move operations
    // implicitly generates the default and copy constructors and the destructor
};

inline S6& S6::operator=(const S6&) = default;
    // Explicitly request the compiler to generate the default implementation
    // for this copy-assignment operator. This request might fail, e.g., if S6
    // were to contain a non-copy-assignable data member.
```

Alternatively, an explicitly defaulted noninline implementation of this copy-assignment operator may appear in a separate (`.cpp`) file; see *Use Cases — Physically decoupling the interface from the implementation* on page 40.

Use Cases

Restoring the generation of a special member function suppressed by another

Incorporating `= default` in the declaration of a special member function instructs the compiler to generate its definition regardless of any other user-provided special member functions. As an example, consider a **value-semantic** `SecureToken` class that wraps a standard string, `std::string`, and an arbitrary-precision-integer, `BigInt`, token code that together satisfy certain invariants:

```
class SecureToken
{
    std::string d_value;  // The default-constructed value is the empty string.
    BigInt      d_code;   // The default-constructed value is the integer zero.

public:
    // All six special member functions are implicitly defaulted.

    void setValue(const char* value);
    const char* value() const;
    BigInt code() const;
};
```

By default, a secure token's `value` will be the empty-string value, and the token's `code` will be the numerical value of zero because those are, respectively, the **default-initialized** values of the two data members, `d_value` and `d_code`: