```
void f()
{
    SecureToken token;                       // default constructed         (1)
    assert(token.value() == std::string());  // default value: empty string (2)
    assert(token.code() == BigInt());        // default value: zero         (3)
}
```

Now suppose that we get a request to add a **value constructor** that creates and initializes a `SecureToken` from a specified token string:

```
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:
    SecureToken(const char* value);  // newly added value constructor

    // suppresses the declaration of just the default constructor --- i.e.,
    // implicitly generates all of the other five special member functions

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

Attempting to compile function `f` would now fail on the first line, where it attempts to default-construct the token. Using the `=default` feature, however, we can reinstate the default constructor to work ~~trivially~~, just as it did before:

```
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:
    SecureToken() = default;         // newly defaulted default constructor
    SecureToken(const char* value);  // newly added value constructor

    // implicitly generates all of the other five special member functions

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