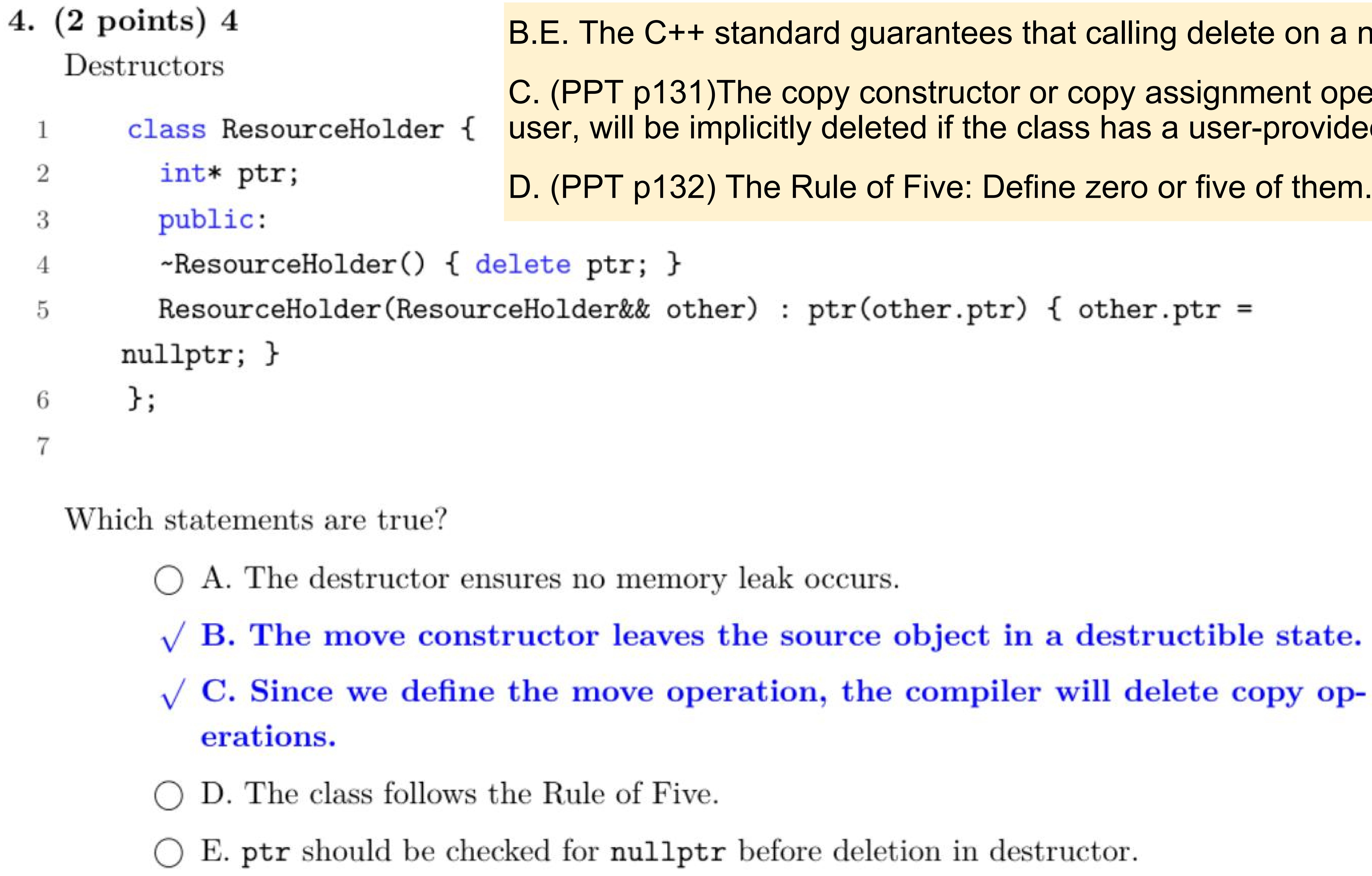
Recitation Class

Week 3

Recitation 3

- hw2 multiple choices
- quiz (six multiple-choice questions, about 15 mins)
- check

Homeworks



4. (2 points) 4

Destructors

```
class ResourceHolder {
  int* ptr;
  public:
  ~ResourceHolder() { delete ptr; }
  ResourceHolder(ResourceHolder&& other) : ptr(other.ptr) { other.ptr =
nullptr; }
};
```

Which statements are true?

- ○ A. The destructor ensures no memory leak occurs.
- ✓ **B. The move constructor leaves the source object in a destructible state.**
- ✓ **C. Since we define the move operation, the compiler will delete copy operations.**
- ○ D. The class follows the Rule of Five.
- ○ E. `ptr` should be checked for `nullptr` before deletion in destructor.

B.E. The C++ standard guarantees that calling delete on a null pointer is safe (do nothing).

C. (PPT p131)The copy constructor or copy assignment operator, if not provided by the user, will be implicitly deleted if the class has a user-provided move operation.

D. (PPT p132) The Rule of Five: Define zero or five of them.

Homeworks

5. Which statements about rvalue references are correct?

✓ **A. `std::move` converts an lvalue to an rvalue reference.**

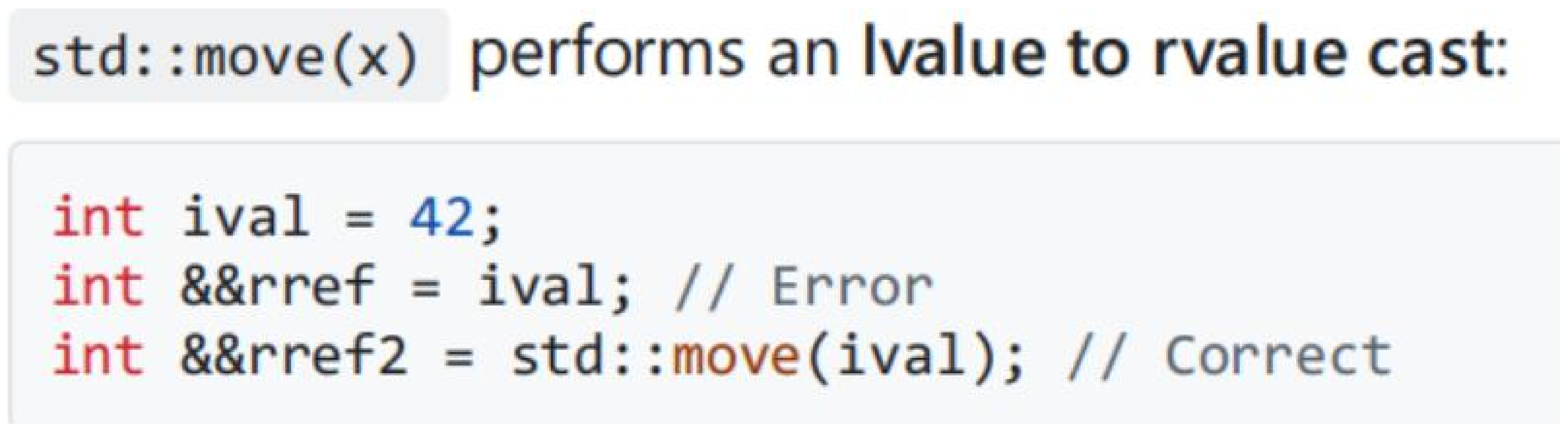
○ B. An rvalue reference parameter can bind to lvalues.

✓ **C. `int&& x = 5;` is valid.**

✓ **D. After moving an object, its state is unspecified.**

○ E. Rvalue references always extend the lifetime of temporary objects.

A.



`std::move(x)` performs an **lvalue to rvalue cast**:

```
int ival = 42;
int &&rref = ival; // Error
int &&rref2 = std::move(ival); // Correct
```

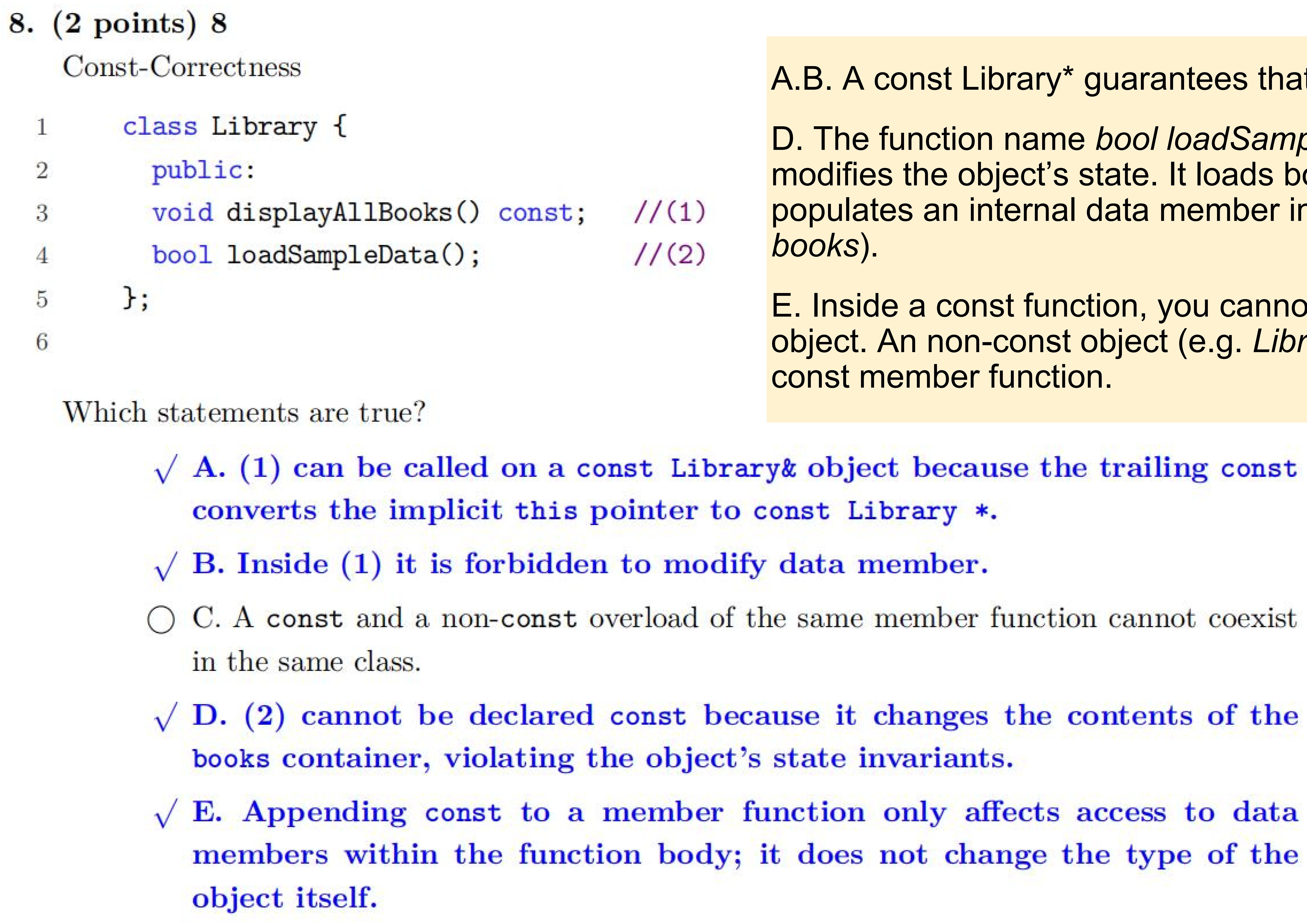
B. (PPT p115) Rvalue reference cannot be bound to lvalue.

C. (PPT p115)

D. (PPT p132) A call to std::move is usually followed by a call to some function that moves the object, after which we **cannot make any assumptions about the value of the moved from object**.

E. “always”

Homeworks



8. (2 points) 8

Const-Correctness

```
class Library {
  public:
    void displayAllBooks() const;    //(1)
    bool loadSampleData();           //(2)
};

```

Which statements are true?

[✓ **A. (1) can be called on a const `Library&` object because the trailing `const` converts the implicit `this` pointer to `const Library *`.**]

[✓ **B. Inside (1) it is forbidden to modify data member.**]

○ C. A `const` and a non-`const` overload of the same member function cannot coexist in the same class.

[✓ **D. (2) cannot be declared `const` because it changes the contents of the `books` container, violating the object's state invariants.**]

[✓ **E. Appending `const` to a member function only affects access to data members within the function body; it does not change the type of the object itself.**]

A.B. A const Library* guarantees that its members won't be modified.

D. The function name *bool loadSampleData()* strongly suggests that it modifies the object's state. It loads book information from a file and populates an internal data member in the class (*std::vector<Book> books*).

E. Inside a const function, you cannot modify any data members of the object. An non-const object (e.g. *Library non_const_lib)* can also call const member function.

Homeworks

9. (2 points) 9

Initializer Lists

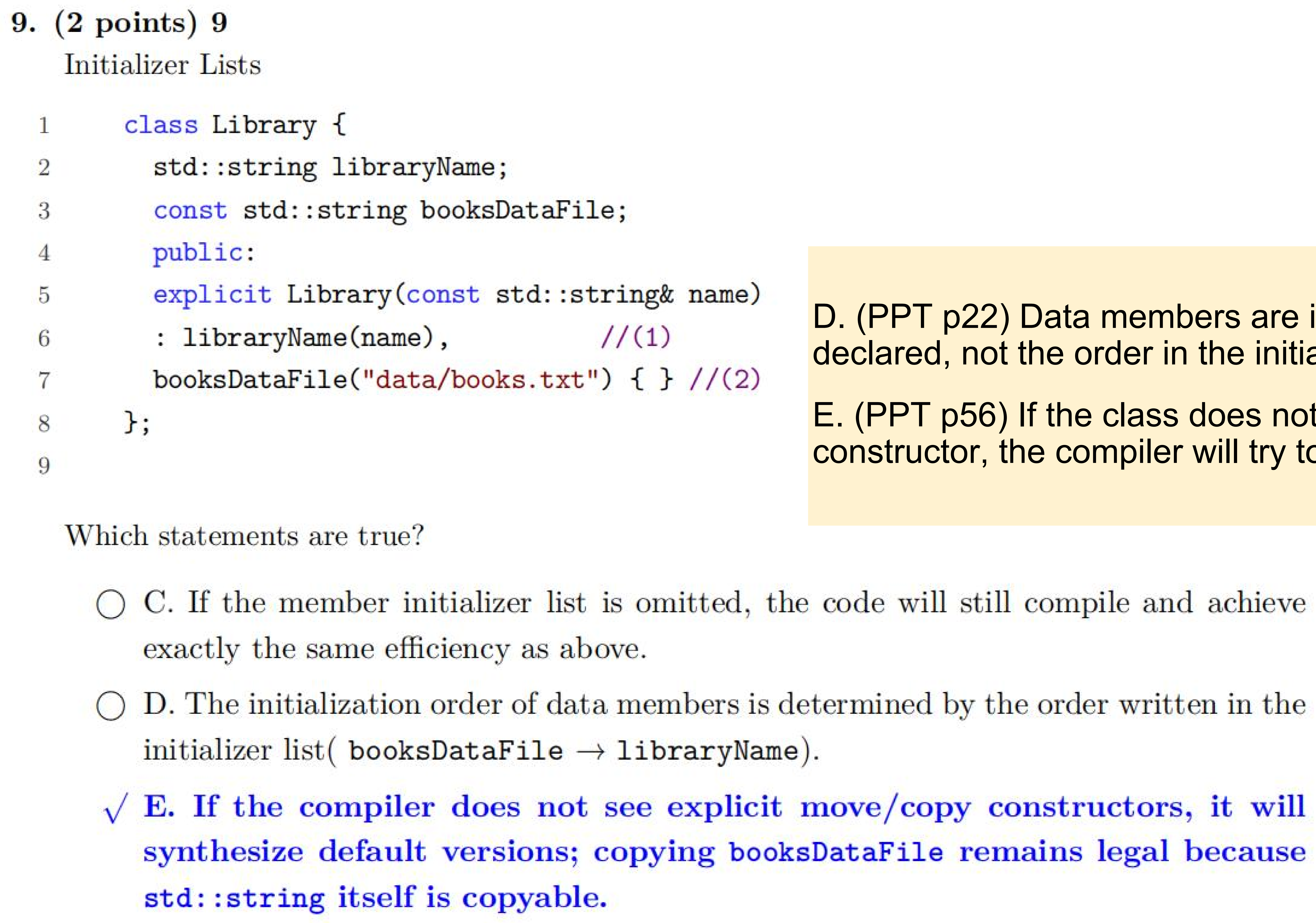
```cpp
class Library {
  std::string libraryName;
  const std::string booksDataFile;
  public:
  explicit Library(const std::string& name)
  : libraryName(name),           //(1)
  booksDataFile("data/books.txt") { } //(2)
};

```

Which statements are true?

- ✓ **A. Lines (2) and (1) are part of the constructor's member initializer list, used to directly construct data members before entering the function body.**
- ✓ **B. `booksDataFile` is `const`, so it can only be assigned in the initializer list and remains read-only thereafter.**

Homeworks


9. (2 points) 9

Initializer Lists

```cpp
class Library {
  std::string libraryName;
  const std::string booksDataFile;
  public:
  explicit Library(const std::string& name)
  : libraryName(name),          //(1)
  booksDataFile("data/books.txt") { } //(2)
};

```

Which statements are true?

○ C. If the member initializer list is omitted, the code will still compile and achieve exactly the same efficiency as above.

○ D. The initialization order of data members is determined by the order written in the initializer list(`booksDataFile` → `libraryName`).

[✓ **E. If the compiler does not see explicit move/copy constructors, it will synthesize default versions; copying `booksDataFile` remains legal because `std::string` itself is copyable.**]

D. (PPT p22) Data members are initialized in order in which they are declared, not the order in the initializer list.

E. (PPT p56) If the class does not have a user-declared copy constructor, the compiler will try to synthesize one:

Homework

10. (2 points) 10

Lifetime

```cpp
Book* Library::findBookByIsbn(const std::string& isbn) {
  auto it = std::find_if(books.begin(), books.end(),
  [&isbn](const Book& b){ return b.getIsbn() == isbn; });
  return (it != books.end()) ? &(*it) : nullptr;  //(1)
}

bool Library::loadBooksFromFile(const std::string& filename){
  ...
  books.push_back(std::move(book));               //(2)
}
```

Which statements are true?

- ~~✓~~ **A. Pointer (1) points to an element inside `std::vector<Book>`; its validity lasts as long as that element remains alive in the container.**
- [✓ **B. Once `books` reallocation occurs (e.g., `push_back` triggers capacity growth), all pointers previously returned by (1) become invalid.**]
- [✓ **C. After (2), the object `book` remains in a valid but unspecified state; only limited operations such as destruction or reassignment are allowed.**]
- ○ D. After moving `book` into `books`, its destructor is called immediately, causing a dangling pointer risk.

A.B. When a vector reallocation occurs, causing all elements to be **moved** to a new memory address, the original pointer became invalid.

C.D. After moving operation, the object book remains unspecified state. Its destructor will be called at the end of its lifetime.