ShanghaiTech University

CS101A Data Structures
Fall 2025

Homework 2

Due date: October 12, 2025, at 23:59

1 Brief Overview

This homework consists of three parts:

- **Multiple Choices(20 pts):** A set of multiple-choice questions covering basic C++ concepts.
- **Codes(29 pts):** Mandatory programming exercises. You must implement and demonstrate the required functions in `Book.cpp`.
- **Check(9 pts):** We will examine the reasons why you designed these functions in Codes and your understanding about Multiple Choices. This part will take place in the recitation class, and you will need to answer the TA's questions.

Your grade for this homework will be computed as

$$\max\big(\text{Multiple Choices} + \text{Codes} + \text{Check},\ 50\text{ pts}\big).$$

Usage of AI. The use of AI tools for searching information or obtaining assistance on this homework is strictly prohibited. All solutions must be derived from your own understanding and effort. Submissions will be reviewed carefully, and any indication of reliance on AI-generated content will result in severe penalties.

Academic Integrity. This course upholds the highest standards of academic integrity. Any form of academic dishonesty, including plagiarism, unauthorized collaboration, or the use of prohibited resources such as AI tools, will be treated as a serious violation. Such actions not only undermine your own learning but also violate university policies.

2 Multiple Choices

1. (2 points) 1

Static Members

```cpp
class Book {
  static int count;
  public:
  static void increment() { ++count; }
  static int get() { return count; }
};

int Book::count = 0;
```

Which statements are true?

- ◯ A. `count` is shared among all instances of `Book`.
- ◯ B. The initialization of `Book::count` must be done outside the class definition.
- ◯ C. `increment()` can access non-static member variables of `Book`.
- ◯ D. `Book::get()` can be called without any `Book` object instance.
- ◯ E. Static member functions have no `this` pointer.

2. (2 points) 2

Move Semantics

```
  class Data {
    int* ptr;
    public:
    Data(Data&& other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }
    Data& operator=(Data&& other) noexcept {
      delete ptr;
      ptr = other.ptr;
      other.ptr = nullptr;
      return *this;
    }
  };

```

Which statements are true?

- ◯ A. The moved-from object will be immediately destructed after calling move assignment/constructor.
- ◯ B. The move assignment operator should check for self-assignment.
- ◯ C. Move operations typically don't throw exceptions and are marked `noexcept`.
- ◯ D. If we define a move constructor, the compiler will automatically generate a copy constructor.
- ◯ E. The expression `other.ptr = nullptr` in move constructor prevents double deletion.

3. (2 points) 3

Operator Overloading

```cpp
class Vector {
  int x, y;
  public:
  Vector operator+(const Vector& rhs) const {
    return Vector(x + rhs.x, y + rhs.y);
  }
  friend std::ostream& operator<<(std::ostream& os, const Vector& v);
};

```

Which statements are true?

- ◯ A. The `+` operator is overloaded as a member function.
- ◯ B. `operator+` can access private members of `rhs`.
- ◯ C. `operator<<` must be declared as a friend to access private members.
- ◯ D. The expression `v1 + v2` is equivalent to `v1.operator+(v2)`.
- ◯ E. `operator+` returns a reference to a temporary object.

4. (2 points) 4

Destructors

```cpp
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.

5. (2 points) 5

Rvalue References

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.

6. (2 points) 6

Friend Functions

```cpp
class Box {
  int width;
  void printWidth(const Box& b) { std::cout << b.width; }
};

```

Which statements are true?

◯ A. `printWidth` can access private members of `Box`.

◯ B. `printWidth` is a member function of `Box`.

◯ C. Friend declarations can appear in any section (public/private) of the class.

◯ D. `printWidth` must be defined inside the class.

◯ E. Friendship is inherited in derived classes.

7. (2 points) 7

Type Aliases

```cpp
class Container {
  public:
  using size_type = std::size_t;
  using value_type = int;
};

```

Which statements are true?

◯ A. `size_type` can be accessed as `Container::size_type`.

◯ B. Type alias members are independent of access specifiers (private, protected, public).

◯ C. `value_type` can be modified by class users.

◯ D. Type aliases can be used within member functions.

◯ E. `using` declarations can replace `typedef` in C++.

8. (2 points) 8

Const-Correctness

```cpp
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.

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.
- ◯ 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` $\rightarrow$ `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.

10. (2 points) 10

Lifetime

```
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.

3 Codes

The objective of this part is to complete the implementation of the `Book` class. The `Book` class is a core component of a library management system, used to represent and manage information for individual books.

You will need to fill in the specific implementations for several key functions within a pre-defined framework in the `Book.cpp` file. The focus of this assignment is to learn **resource management** when a class contains dynamically allocated resources.

Project Directory Structure

```
LibraryManagementSystem/
├── include/ ........................................ Header files
│   ├── Book.h ...................................... Book class
│   └── Library.h ...................... Main library management class
├── src/ ............................................ Source files
│   ├── Book.cpp ............................ Book implementation
│   ├── Library.cpp ...................... Library implementation
│   └── main.cpp .................. Main program with menu system
├── data/ ............................................. Data files
│   └── books.txt ............................... Books database
├── hw2.pdf ............................................ This file
├── Makefile ................................. Build configuration
└── library_system_example.exe ................... Sample program
```

To see the library's functionality in action, run the sample program `library_system_example.exe`. Upon the first launch, the program will automatically create sample data files. You can navigate through the console menu using the number keys.

If you finish all the code, use Makefile to build the project

```
# Build the project
make

# Run the program
make run

# Clean build files
make clean
```

3.1 Resource Management and the "Rule of Five"

Inside the `Book` class, there is a member variable `int* relatedBookId;`. This is a pointer to dynamically allocated memory, used to store a list of related book IDs. When you allocate memory using `new[]`, you must release it at the appropriate time using `delete[]`; otherwise, you will cause a **memory leak**.

```cpp
class Book {
  private:
  // ... other member variables
  std::size_t relatedBookCount; // The number of related books
```

```
    int* relatedBookId;         // A pointer to a dynamic array storing related book IDs
    // ... other member variables
};
```

Relying on compiler-generated default functions is insufficient for correctly handling such dynamic resources. To solve these issues, you need to manually implement the **"Rule of Five"**, which consists of the following five special member functions:

1. **Destructor**
2. **Copy Constructor**
3. **Copy Assignment Operator**
4. **Move Constructor**
5. **Move Assignment Operator**

You need to open the `Book.cpp` file and complete the implementation of the following 8 functions.

3.2 Task 1: "The Rule of Five"

These five functions work together to ensure that the dynamic memory managed by the `relatedBookId` pointer is correctly created, copied, moved, and destroyed.

3.2.1 Copy Constructor `Book::Book(const Book& other)`(4 points)

- To create a new `Book` object from an existing `Book` object (`other`).
- Pay attention to the case that `other.relatedBookCount` is 0. Otherwise allocate a new array for `relatedBookId`.

3.2.2 Copy Assignment Operator `Book& Book::operator=(const Book& other)`(4 points)

- To assign the values of an existing `Book` object (`other`) to another existing `Book` object (`this`).
- Guard against self-assignment.

3.2.3 Move Constructor `Book::Book(Book&& other) noexcept`(4 points)

- To create a new object from a temporary `Book` object (an rvalue, `other`) that is about to be destroyed. This is an optimization that improves efficiency by ”stealing” resources rather than copying them.
- Using std::move for movable members and copying basic types.
- Null out other.relatedBookId and reset other.relatedBookCount to 0.

3.2.4 Move Assignment Operator `Book& Book::operator=(Book&& other) noexcept`(4 points)

- To assign the resources and values of a temporary object (`other`) to an existing object (`this`).
- Check for self-assignment.

3.2.5 Destructor `Book::~Book()`(4 points)

- Release the resources owned by a `Book` object when its lifetime ends.
- Use delete[] on relatedBookId to release the dynamic array when the object is destroyed.

3.3 Task 2: Implement Core Business Logic Functions

Complete the implementation of the following functions according to the comments in the `Book.cpp` file.

3.3.1 `void Book::setRelatedBook(std::size_t count, const int* relatedBooks)`(3 points)

3.3.2 `bool Book::borrowBook(const std::string& memberId, int borrowDays)`(3 points)

3.3.3 `bool Book::operator==(const Book& other) const`(3 points)

4 Submission

Submit `Book.cpp` to OJ.