Project Report: Academic Literature Catalogue

Jakub Stavina
Student ID: ———

Department of Physics and Astronomy
The University of Manchester

November 6, 2024

Abstract

In academia, effective management of scholarly literature plays an important role in facilitating research and academic discourse. This project proposes and develops an academic literature catalogue - a program designed to streamline the organization, retrieval, and manipulation of academic literature and other related entries. At the core of the catalogue lies a structured database, implemented via class `LiteratureDatabase`, accommodating three main types of literature: books, theses, and journals. The database holds an array of entries characterized by essential metadata such as titles, authors, publishers, universities, impact factors, and more. The individual literature types are implemented as classes derived from an abstract `LiteratureEntry` class. The class `LiteratureDatabase` provides the user with options for searching, adding, removing, and editing literature entries. Moreover, the user interface allows for the presentation of data in BibTeX style as well as to save database modifications in a new data file. In this report, we discuss details of the implementation and usage of this literature database management tool.

1 Introduction

The primary objective of this project is to create a system with functionalities to manage a literature catalogue. The program architecture is centered around a hierarchical class design with emphasis on aspects of object-oriented programming such as inheritance, encapsulation, abstraction and polymorphism.

1.1 Data Entries and Database Structure

At the core of the program lies a database holding three types of literature entries: books, theses, and journals. These individual literature entries are implemented as classes derived from an abstract class `LiteratureEntry`. Derived classes - `Book`, `Thesis`, and `Journal` - inherit from this base class data members which hold the titles, years and lists of names. Each class incorporates specific data members to capture additional attributes of its corresponding literature type. For books, additional information includes publisher, subject, and price. Theses also hold a list of supervisors, and the affiliated university. Finally, journals are defined by their year of foundation, impact factor, number of volumes, editor names, and scope.

1.2 Functionality and User Interface

The program's functionality is implemented through the methods of the `LiteratureDatabase` class. This class provides users with options that allow for navigation and interaction with the database. Key functionalities include:

(a) Field-based search, enabling users to search literature entries based on author, title, or type,

(b) Insertion and removal of database entries,

(c) Dynamic editing capabilities, allowing users to modify existing entries.

The user interface is console-based. It is implemented via a collection of functions `user-interface-functions.h` which prompt the user, collect and validate their commands and call the functionality of the underlying `LiteratureDatabase`.

1.3 Additional Features

In addition to core functionalities, the program offers supplementary features aimed which expand its utility. Noteworthy features include the ability to output data in BibTeX style for books and theses. Furthermore, the program enables users to save database modifications in a new data file, preserving integrity of the original database.

2 Code Design and Implementation

2.1 Structure of the Literature Entries

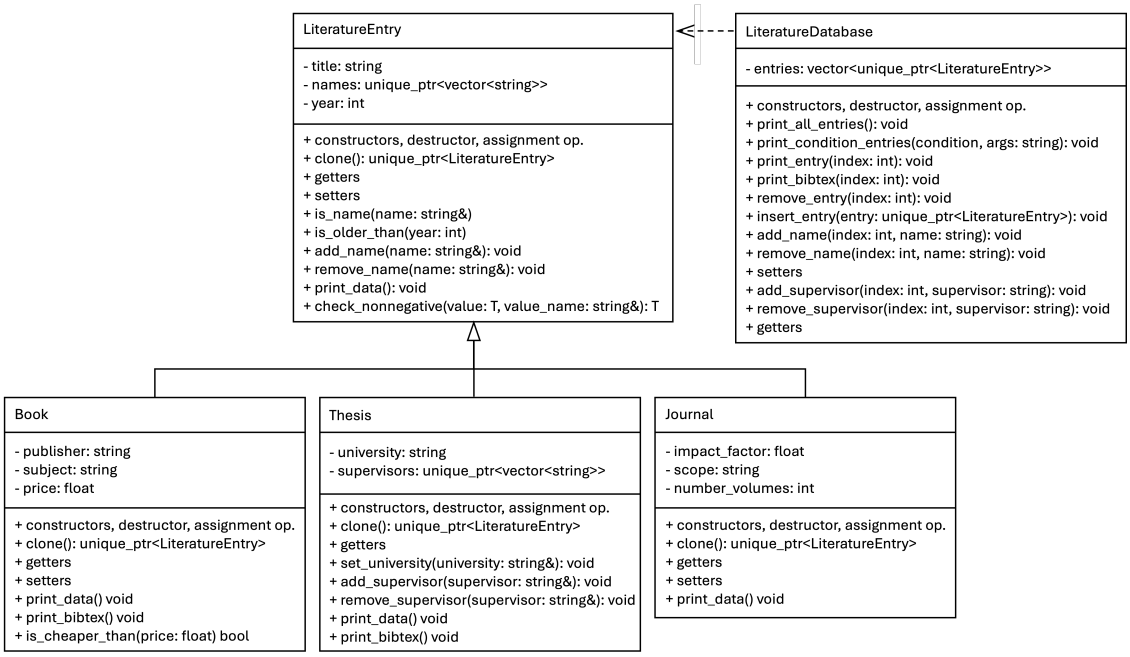

Figure 1: UML class diagram of the system, depicting the class and dependency structure of the various objects used in the project. For brevity, the accessor and mutator functions (also termed setters and getters) were not included among the methods of the classes in full.

2.1.1 Base Class: "LiteratureEntry"

A foundation of the data types stored in the academic literature catalogue program is formed by the `LiteratureEntry` class. Its attributes are

- **title:** A string variable representing the title of the entry.
- **names:** A unique pointer to a vector of strings, storing the names associated with the entry.
- **year:** An integer variable denoting a year associated with the entry.

It contains a methods related to manipulation and retrieval of its data members. The methods include:

- **Parameterised, Copy and Move Constructor:** The parameterised constructor initializes the object with the essential attributes. Input validation for the `year` parameter is integrated into the

constructor. Via deep copy mechanism, the copy constructor creates a new object with an independent copy of the authors' names vector. Using move semantics, the move constructor transfers resources from one object to another, enhancing performance and resource utilization.

- **Copy and Move Assignment Operators:** Both copy and move assignment operators are implemented robustly, ensuring proper handling of self-assignment and correct duplication of resources between objects.
- **is_name():** Using standard library algorithms, the function `is_name()` checks if a given name exists among the authors. It searches for the name using `std::find` [1]. If the name is found, it returns `true`; otherwise, it returns `false`.
- **add_name():** Allowing for addition of names to the literature entry, the `add_name()` function appends the new name to the literature entry's name list.
- **remove_name():** This function deletes name from the list. It employs the `std::remove_if` algorithm [2] to relocate the target name to the vector's end, based on a lambda function that identifies the target name. Subsequently, it uses the `erase` method [3] to eliminate all instances of the name from the vector.

```
void LiteratureEntry::remove_name(const std::string& name) {
  names->erase(std::remove_if(names->begin(),
                              names->end(),
                              [&](const std::string& name_) {return name_ == name;}),
               names->end());
}
```

Listing 1: Implementation of the function `remove_name()`.

- **set_year():** The `set_year()` function modifies the year of the literature entry. Incorporating input validation via the `check_nonnegative()` template function.
- **print_data():** Through the standard library output streams, `print_data()` function, displays details of the literature entry in a human-readable way.
- **check_nonnegative():** This is a template function to ensure that the input value is non-negative. If the value is negative, an `std::invalid_argument` exception is thrown, accompanied by an error message. If the value passes the validation, it is returned unaltered.

The `LiteratureEntry` class is abstract since it contains purely virtual functions:

- **clone():** Creates a deep copy of a `LiteratureEntry` object.
- **get_type():** Retrieves the type of a `LiteratureEntry` object.
- **get_encoding()**: Obtains the encoding of the object in the same style as the database's `.dat` file.

These functions define an interface that derived classes must implement. They allow for polymorphic behavior and enable derived classes to provide custom functionality while adhering to a consistent interface.

2.1.2 Derived Class: "Book"

The `Book` class extends the functionality provided by the base class `LiteratureEntry` to model books within the academic literature catalogue program. as can be seen in the UML diagram in FIG. 1, it introduces additional attributes and book-specific methods:

- **New attributes**:
  - **publisher**: A string variable storing the name of the book's publisher.
  - **subject**: A string variable indicating the subject or topic of the book.
  - **price**: A floating-point variable representing the price of the book.

- **Some of the new methods**:
  - **clone()**: Overrides the base class method to create a deep copy of a `Book` object.

```
std::unique_ptr<LiteratureEntry> clone() const override {
  return std::make_unique<Book>(*this);
}
```

Listing 2: Implementation of the function `clone()`.

  - **New accessors**: Retrieve the type, publisher, subject, and price of the book.
  - **New mutators**: Modify the publisher, subject, and price attributes of the book.
  - **Printing methods**: Print the data of the book, both in a general format and in BibTeX format.
  - **Additional method**: `is_cheaper_than()` compares the price of the book with a given price and returns true if the book's price is less than the given price.

Book Encoding Example: The encoding of a book involves transforming its attributes into a structured format. For example, the encoding of the provided book "A Brief History of Time: From the Big Bang to Black Holes" by Stephen Hawking [4] would look like this:

```
BOOK
A Brief History of Time: From the Big Bang to Black Holes
Hawking, Stephen
1988
Bantam
Cosmology
9.97
```

This format organizes the book's information into distinct lines, with each line representing a different attribute. In order, these are: title, author(s), publication year, publisher, subject, and price.

Example Output of Book Printing: For the previous book example, the output of printing would be:

```
BOOK:
Title         : A Brief History of Time: From the Big Bang to Black Holes
Authors       : Hawking, Stephen
Year          : 1988
Publisher     : Bantam
Subject       : Cosmology
Price         : 9.97
```

Example BibTeX Output of a Book: BibTeX is a tool used for managing references and citations in LaTeX documents. The BibTeX output for the provided book example would look like this:

```
@Book{citekey,
  author    = "Hawking, Stephen",
  title     = "A Brief History of Time: From the Big Bang to Black Holes",
  publisher = "Bantam",
  year      = "1988"
}
```

This format adheres to the BibTeX syntax, with each attribute enclosed in curly braces and preceded by its corresponding field name. The "citekey" serves as a unique identifier for referencing the book within a LaTeX document.

2.1.3 Derived Class: "Journal"

The `Journal` class inherits from the base class `LiteratureEntry` and extends its functionality to represent academic journals. As can be seen in the UML diagram in FIG. 1, this class introduces new attributes and methods unique to journal publications.

- **New attributes:**
  - **impact_factor:** A positive float representing the impact factor of the journal.

- **scope:** A string indicating the field of study covered by the journal.
- **number_volumes:** A positive integer denoting the total number of volumes published by the journal.

- **Some of the new methods:**
  - **Clone Function:** A function to create a deep copy of a `Journal` object.
  - **Accessors:** Retrieve the type, impact factor, scope and the number of volumes.
  - **Mutators:** Modify the impact factor, scope or number of volumes. Where appropriate, input checking is done as discussed in the section on `LiteratureEntry`.
  - **Printing:** `print_data()` function displays the details of the journal entry in a human-readable way.

Journal Encoding Example: The encoding of a journal "Physical Review Letters" would look like this:

```
JOURNAL
Physical Review Letters
Chate, Hugues;Garisto, Robert;Mitra, Samindranath
1958
8.6
Physics
131
```

The attributes in order are: title, editor(s), foundation year, impact factor, subject, and number of volumes. Different editors are separated by a semicolon.

2.1.4 Derived Class: "Thesis"

Expanding upon `LiteratureEntry`, the `Thesis` class encapsulates attributes and functionalities specific to academic theses. It inherits the core attributes and methods from its base class as can be also seen in the UML diagram in FIG. 1. The new attributes are:

- **university:** The name of the university where the thesis was completed.
- **supervisors:** A unique pointer to a vector of strings, this attribute stores of the names of thesis supervisors. It allows for the association of multiple supervisors with a single thesis.

The `Thesis` class also implements specialized methods. Among these are the usual accessors and mutators as well as:

- **Parametrized Constructor:** In addition to initializing the attributes inherited from `LiteratureEntry`, this constructor initializes the `university` and `supervisors` attributes specific to the `Thesis` class.
- **Copy Constructor:** Ensuring deep copy semantics, this constructor creates an independent copy of the `Thesis` object, including its `supervisors` vector.
- **Move Constructor:** Leveraging move semantics, the move constructor efficiently transfers resources from one `Thesis` object to another.
- **Assignment Operators:** Both copy and move assignment operators are implemented to ensure proper handling of resource transfer and duplication between `Thesis` objects.
- **add_supervisor():** Adds supervisor names to the thesis entry.
- **remove_supervisor():** Provides functionality for removing a specific supervisor's name in a way similar to the `remove_name` function discussed above.
- **print_data():** Overrides the base class method to print essential details of the thesis entry in a human-readable format. It also utilizes the base class printing to avoid code duplication.
- **print_bibtex():** Printing functionality to generate a BibTeX entry representing the thesis.

Thesis Encoding Example: The encoding of the thesis "Gravity actions from matter actions" by Christof Witte [5] would be:

```
THESIS
Gravity actions from matter actions
Witte, Christof
2014
Humboldt University of Berlin
Frederic P. Schuller
```

The attributes in order are: title, author(s), year of publication, university, and supervisor(s).

Example BibTeX Output of a Thesis: BibTeX entry for this thesis would look like:

```
@phdthesis{citekey,
  author    = "Witte, Christof",
  title     = "Gravity actions from matter actions",
  school    = "Humboldt University of Berlin",
  year      = "2014"
}
```

2.2 The "LiteratureDatabase" class

The `LiteratureDatabase` class depends on the class `LiteratureEntry` (see FIG. 1). It provides a framework for managing a collection of literature entries, with functionality for storage, retrieval, modification, and presentation of entries. It holds a vector of unique `LiteratureEntry` base class pointers, allowing for dynamic memory management. The class involves the standard "rule of five" constructors and assignment operators.

2.2.1 Parameterised constructor

This constructor initializes a `LiteratureDatabase` object by reading data from a file specified by the input `filename`. It attempts to open the file specified by `filename` using an input file stream `std::ifstream`. If the file fails to open, it throws a `std::runtime_error` with a descriptive message. It then reads the file entries using `std::getline`. For each entry from the file, it identifies its type (Book, Journal, or Thesis). It reads the data for each entry type from the following lines in the file. Based on the type, it constructs the corresponding literature object using the extracted data. It stores the created entry objects in the `entries` vector using `std::make_unique` to create unique pointers and `push_back` to add them to the vector. If the entry type is unrecognized, it outputs a warning message to `std::cerr`. Finally, it closes the file stream after reading all the data. This constructor is a crucial part of the `LiteratureDatabase` class as it allows for the initialization of the database from an external data source.

2.2.2 Other Member functions

The class provides a variety of member functions for interacting with the database. These include printing all entries, printing all entries satisfying a specific condition, printing individual entries in the BibTeX format, removing entries, and inserting new entries. There are mutator functions to modify individual entries. The accessor and mutator functions often need to retrieve or change aspects of `LiteratureEntry`-derived objects which are not accessible through base class pointer. This is dealt with through the use of dynamic casting. For example:

```
// Setter for scope at specified index; attempt at a dynamic cast
void LiteratureDatabase::set_scope(int index, std::string scope) {
  if (entries[index]->get_type() == "Journal") {
    dynamic_cast<Journal*>(entries[index].get())->set_scope(scope);
  }
}
```

Listing 3: Implementation of the function `set_scope()`.

Dynamic casting is also employed in the `print_bibtex` function, where the `try-catch` semantics is used to handle cases of bad casting or lack of data-associated `print_bibtex` method. Advanced functions are also available to retrieve information from the database, such as the average book price and the total number of entries.

Finally, there is a function to obtain the encoded data of the entire catalogue in the style in accordance to the provided `.dat` data file.

2.3 User Interface Functions

The `user-interface-functions.h` is a collection of functions that serves as a central component for facilitating user interaction with the literature database. It implements functions for collecting user input, validating input, executing user commands, and interacting with the underlying database. It begins by defining **standard library sets of strings** of valid commands, fields and types (`valid_commands`, `valid_fields`, `valid_types`). Dictionaries are declared to map strings to the corresponding enumerators for commands, fields, and types. Functions for printing information (`print_info()`), validating commands (`validate_command()`), and collecting and validating user input for commands, types, and fields (`collect_and_validate_command()`, etc.) are defined. Functions for validating integers and floats, and prompting the user for yes/no responses are included. Wrapper functions are defined for printing entries of a specific type (`type_print()`) and printing entries based on a specified field (`general_print()`). Most of these functions interact with an instance of the `LiteratureDatabase`. Functions for adding new entries to the database are provided for each supported type of literature (`add_book()`, etc.). These functions prompt the user for information and return a pointer to the newly created object. Additional functions are included for saving a string to a file (`save_string_to_file()`), editing existing literature entries (`edit_literature_entry()`, `edit_book()`, etc.), and executing user-specified commands (`execute_command()`). The `switch` semantics is used heavily whenever case handling related to commands, fields or types is needed.

3 Results

Upon compiling and running the main program "acad-lit-catalogue.cpp", the user is prompted to enter the name of the data file to be read. The user is presented with an informative message detailing the available commands, valid fields, and types.

3.1 Available Commands

Users can perform various actions by typing these commands into the terminal:

1. "INFO": Prints information about the available commands.
2. "PRINT": Prints an entry from the database.
3. "QUIT": Exits the program.
4. "DELETE": Removes an entry from the database.
5. "EDIT": Allows users to edit an existing entry in the database.
6. "ADD": Adds a new entry to the database.
7. "BIBTEX": Prints BibTeX information.
8. "SAVE": Saves the data.
9. "PRICE": Calculates and prints the average price of Book entries in the database.

3.2 Valid Fields

The user can search for entries based on the following fields:

1. "ALL": Prints all entries in the database.
2. "NAME": Searches for entries based on the name of the author or contributor.
3. "TITLE": Searches for entries based on the title of the literature.
4. "TYPE": Searches for entries based on the type of literature (e.g., Book, Journal, Thesis).

3.3 Usage

The user can navigate through the available commands to perform tasks by typing the relevant commands into terminal. They will be prompted to specify the field and type to narrow down their search criteria when printing entries. A user may use the "PRINT" command to print all entries based of the desired field (e.g., "NAME", "TITLE") or type (e.g., "Book", "Journal"). The "ADD" command allows users to add new entries to the database. The user will be prompted to specify the type of literature and to provide relevant information. Users can also save their database contents after performing any changes using "SAVE" or type in "PRICE" to calculate the average price of entries.

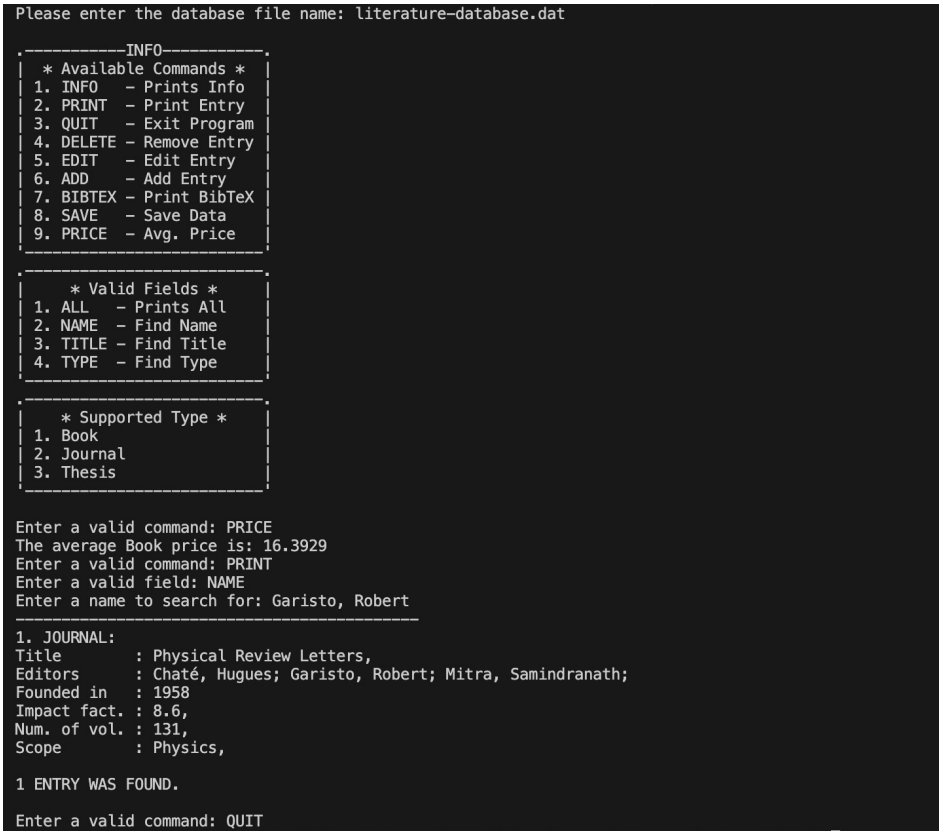

Figure 2: Example of reading from the file "literature-database.dat", calculating the average book price and searching for a name. The information about available commands was also displayed.

4 Discussion and Conclusions

4.1 Discussion

While the proposed project covers essential functionalities, there are several ways it could be improved and extended:

1. **Enhanced Search Options:** Supporting new aspects based on which the search could be performed, such as publication year, subject area, or keyword search.
2. **Improved Interface for Deleting Entries:** Enhancing the deleting functionality, by incorporating a multi-select option, would streamline the process and make it more user-friendly. Providing confirmation prompts before deletion to prevent accidental entry removal.
3. **Export and Import Functionality:** Adding further functionality to export and import the database in different existing formats (e.g., CSV, JSON).

4. **User Authentication and Access Control:** Implementing user authentication and access control mechanisms would allow for secure access to the program.

4.2 Conclusions

In summary, this academic literature catalogue represents a solution tailored to meet the demands of managing and accessing scholarly resources. Through database structuring, intuitive user interface design, and a variety of supplementary features, the program presents a toolkit well-suited to streamline management of a small academic literature database which contains limited variety of entry types.

References

[1] cplusplus.com. `https://cplusplus.com/reference/algorithm/find/`. [Accessed 06-05-2024].

[2] cplusplus.com. `https://cplusplus.com/reference/algorithm/remove_if/`. [Accessed 06-05-2024].

[3] cplusplus.com. `https://cplusplus.com/reference/vector/vector/erase/`. [Accessed 06-05-2024].

[4] Stephen Hawking. *A Brief History of Time: From the Big Bang to Black Holes*. Bantam, 1988.

[5] Christof Witte. *Gravity actions from matter actions*. PhD thesis, Humboldt University of Berlin, 2014.