Implementing a Star Catalogue in C++

Alexander Stansfield
10626406
School of Physics and Astronomy
University of Manchester
(Dated: May 15, 2023)

This project aims to develop a star catalogue that allows users to perform various actions, such as reading files into the catalogue and interacting with objects. It implements classes and smart pointers to help organise the code, promote code reusability, and ensure proper memory management. The implemented user interface (UI) allows users to interact with the star catalogue effectively. The user can navigate through sections, view objects, and save the catalogue into a .csv file in a comma-delimited format. The UI provides a convenient way for users to manage and explore the data in the catalogue, enhancing their experience and facilitating efficient navigation and interaction within the system.

I. INTRODUCTION

The rate at which astronomical objects are being observed and documented is ever-increasing, especially with the rise of new telescopes such as the James Webb Telescope. Currently, an estimated 200 sextillion stars exist in the universe and many other types of objects. As such, it is necessary to have a catalogue enabling a user to store, manipulate and save the data of every newly observed object. However, another critical aspect is a user-friendly interface which allows the user to easily navigate through the catalogue and display information about an interested Galaxy, Star, etc.

This element of a good catalogue lends itself well to an object-oriented approach since each catalogue section can be separated using different classes. Moreover, each object type can be restricted to a predefined class, enabling easy separation of object features and behaviour. However, features are also shared between all objects, such as names. As such, using an abstract class from which all the other astronomical classes are derived is appropriate.

This project aimed to incorporate the essential components of a star catalogue by storing, manipulating and saving data while maintaining a simple interface to view all the different objects and their information. To achieve this functionality, a hierarchy of objects was created, as shown in Figure 1, and each level in the hierarchy was given its own "section" class. The user can then input in single/multiple .csv files or add an object manually. Once all filenames are given, the files are read into the database, and every object is initialised depending on its type and then added to the correct section. The program finds objects with the same name and prompts the user to decide which to keep in the catalogue and which to delete. Since each object type is added to different sections, the UI can print all the names of one type depending on what section the user wants to view. This allows easy navigation of the catalogue.

II. CODE DESIGN & IMPLEMENTATION

The code is split into four main components: the UI handling, the star catalogue class, the sections class and the astronomical objects class. Each component is separated into a header (.h file) file, where the functions are declared, and a source (.cpp) file, where the functions are defined. Doing this for the class files is helpful mainly for readability; however, this approach is vital for the UI functions, which are accessed using a namespace. Since the UI header file is used in other source files, but itself includes other header files, the declarations of the functions need to be separate from their definitions to not define a function multiple times in compilation or have circular inclusion, which leads to a linking error.

A. Class Hierarchy

The AstroObject class is the base class for all astronomical objects, as is the Section class for all section types in the catalogue. These provide common functionality and define pure virtual functions that the subclasses must implement.

The AstroObject class has protected variables (name, ascension, declination etc.) shared amongst all objects. As well as this, it has functions already defined (such as get_astro_name), which all derived objects can use. The classes for objects of type Galaxy, Stellar Nebula, Solar System, Star and Planet are derived from the AstroObject class, but each has its variables separate from the base class. For example, Galaxy has a variable which determines its Hubble type. This specific attribute was solved using an std::set since there are a countable number of options for the Hubble type. When the user makes a Galaxy object manually, the user's input for the Hubble type is checked against a predetermined set using the find() function. If the iterator that find() returns points to the end of the set, then the user's input does not match any strings in the set and so can be asked for another input.

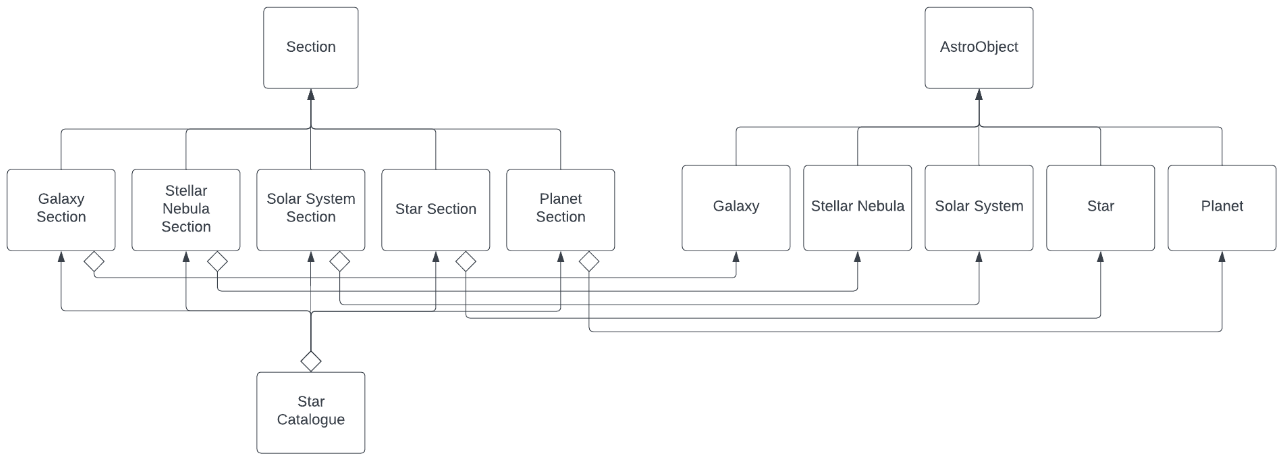
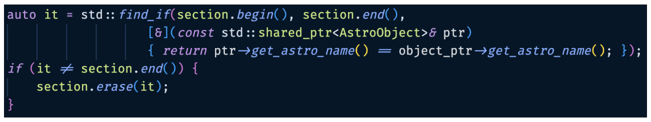


FIG. 1. Figure showing the structure of the classes used in this project. The flow diagram is made using the UML design.

The Selection class is less similar since the derived classes are not strictly necessary. Instead of having five derived classes for the type of astronomical object, five instances of Selection could have been made (if it were not an abstract class since abstract classes cannot be instantiated). However, the design choice to have five derived section types from an abstract class allows for easier tracking of what section holds what type and easier debugging. This is because each section contains a vector of shared pointers that can point to AstroObject instances rather than just a derived class instance.

The StarCatalogue class acts as the central controller for the star catalogue. It manages the sections and provides functionality for reading object data from a file, setting object relationships, and retrieving objects based on their names. Setting up parent-child relationships was essential since Stellar Nebulae are contained in Galaxies and so on. However, having a linear relationship up and down the class hierarchy was not enough since, for example, a star can be part of a galaxy but not within a stellar nebula. To add this functionality, each object entry has the name of the parent object if it has a parent. All objects are instantiated to avoid referencing an object that does not yet exist, and then the parent-child relationship is set at the end of the catalogue creation. Each object has two additional attributes, m_parent (a pointer) and m_children (a vector of pointers). For each object (child object), the catalogue grabs the parent pointer from the correct section and copies this into the m_parent attribute of the child. The pointer to the child object is then copied to m_children in the parent object. To remove an object from the catalogue, e.g. when a duplicate is found, std::find_if was used to iterate over all elements inside the section's vector. A lambda function was used as the search criterion, returning true once the object's name was found, as shown in Figure 2. Once this happens, the iterator of the object is returned and can be easily used with the function erase() to remove the object from the section. However, it is also critical to remove the object from the parent's children list and set the pointer for all its children to a null pointer.



```
auto it = std::find_if(section.begin(), section.end(),
                       [&](const std::shared_ptr<AstroObject>& ptr)
                       { return ptr->get_astro_name() == object_ptr->get_astro_name(); });
if (it != section.end()) {
    section.erase(it);
}
```

FIG. 2. This code block uses a lambda function which returns true if the object_ptr passed through as a parameter has the same name as the current pointer in the iteration.

B. Object Management & Memory

The project uses smart pointers to manage object ownership, ensuring proper memory management and avoiding memory leaks. The code uses container classes like std::vector and std::map to store and organise astronomical objects.

All AstroObject instances are constructed using ‘std::shared_ptr‘. This type of smart pointer provides shared ownership of dynamically allocated objects. It allows multiple ‘std::shared_ptr‘ instances to point to the same object, keeping track of the number of references. This is important as the Section vector contains a pointer to an AstroObject, but the parent of the AstroObject will also need to hold the same pointer. The object remains alive as long as at least one ‘std::shared_ptr‘ points to it, helping reduce the likelihood of a memory leak. This shared ownership mechanism is useful when multiple parts of the code need access to the same object without worrying about explicit memory deallocation.

For the pointer to the parent object, a’std::weak_ptr‘ was used. It is also a smart pointer but provides non-owning, weak references to an object managed by 'std::shared_ptr'. Weak pointers have many uses but are particularly useful in scenarios where cyclic references between objects exist, as 'std::weak_ptr' breaks the cycle and avoids memory leaks. In this case, there could be a cyclic reference between the parent and child object. However, the use of a weak pointer ensures that memory is safe if such an event were to occur. By checking if the object is still valid using the 'std::weak_ptr''s 'expired()' function, one can safely access the object or convert it back to a 'std::shared_ptr' using the 'lock()' function.

Since only the StarCatalogue instance should be able to access each section, i.e. other classes such as AstroObject should never be able to access the section, the StarCatalogue uses 'std::unique_ptr'to reference each section instance. A unique pointer is a smart pointer that provides exclusive ownership of a dynamically allocated object. Unlike 'std::shared_ptr', it cannot be copied, only moved, which enforces a clear ownership model. It is the preferred smart pointer when exclusive ownership is required, as it guarantees that there is only one owner of the object. This prevents accidental sharing, which is essential as the sections should always be stored in the main StarCatalogue instance.

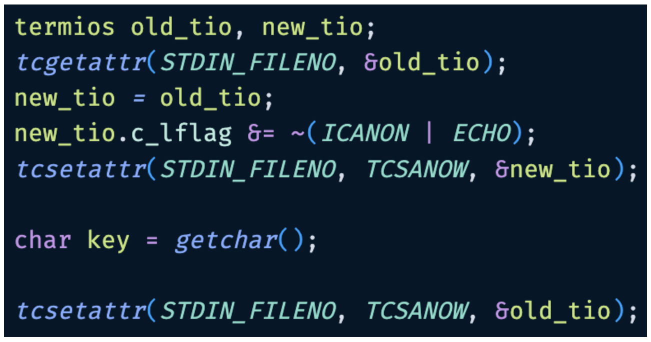
To easily access the correct section inside the star catalogue, a 'std::map' was used to store the unique pointers. Each pointer has a key associated with it. For example, the Galaxy section has an std::string 'Galaxy'. This allows the code to access the correct section with a logical keyword.

C. User Interface

The UI allows users to interact with the star catalogue by performing various actions such as inputting filenames, reading files, navigating through the catalogue, and executing menu options.

The UI consists of several functions that handle different aspects of the user interface. The 'clear_terminal()' function clears the terminal screen. The 'input_filenames()' function prompts the user to enter filenames to be read into the catalogue. The 'read_files()' function reads the specified files into the catalogue. The 'move_up()' and 'move_down()' functions move the catalogue up and down one section, respectively.

The menu functions, main_menu(), make_usr_object_menu() and print_catalogue(), display the appropriate actions and an arrow to indicate which action the user will execute using a nested for loop inside a while loop. At each iteration of the while loop, the menu is reprinted onto the terminal along with the current arrow position. At the end of the iteration, a 'handle keypress' function is run to take in the user input. To allow the user to input a command without having to press the enter button each time, which is necessary if one is to use std::cin, the termios and unistd libraries are used.



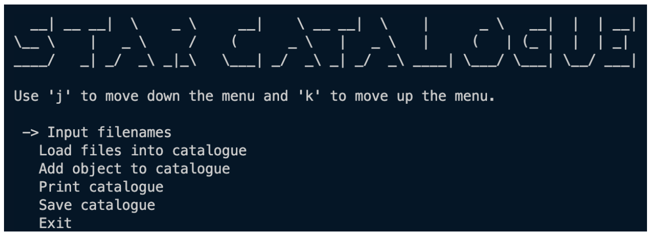
```
termios old_tio, new_tio;
tcgetattr(STDIN_FILENO, &old_tio);
new_tio = old_tio;
new_tio.c_lflag &= ~(ICANON | ECHO);
tcsetattr(STDIN_FILENO, TCSANOW, &new_tio);

char key = getchar();

tcsetattr(STDIN_FILENO, TCSANOW, &old_tio);
```

FIG. 3. This code snippet prepares the terminal to read raw input without echoing characters, reads a single character, and then restores the original terminal settings.

The code snippet in Figure 3 configures the terminal to read a user input without echoing them back to the screen. It temporarily modifies the terminal settings, reads a single character from the user, and then restores the original settings to maintain the expected behaviour of the terminal. This lets the user press a key to execute the correct action automatically. This functionality allows for swift navigation, improving the user experience.

III. RESULTS

Once the program is executed, the user is met with the main menu screen, showing the user's options, as shown in Figure 4. The user can navigate these options using 'j' and 'k' on the keyboard to move the arrow down and up the menu. Once the user presses enter, the function associated with the option at which the arrow is placed is executed.

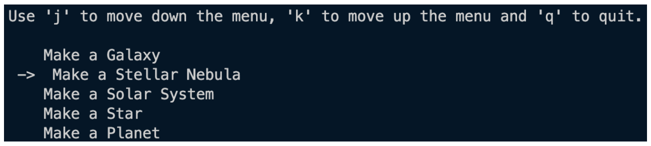


```
  __| __ __|   \     _ \      __|     \ __ __|   \    |       _ \   __|  |  | __|
\__ \    |    _ \      /     (       _ \   |    _ \   |      (   | (_ |  |  | _|
____/   _|  _/  _\ _|_\    \___| _/  _\ _| _/  _\ ____| \___/ \___| \__/ ___|

Use 'j' to move down the menu and 'k' to move up the menu.

 -> Input filenames
   Load files into catalogue
   Add object to catalogue
   Print catalogue
   Save catalogue
   Exit
```

FIG. 4. Main menu immediately after the program is executed. Note that the position of the arrow changes when the user presses the keys' j' or 'k'.

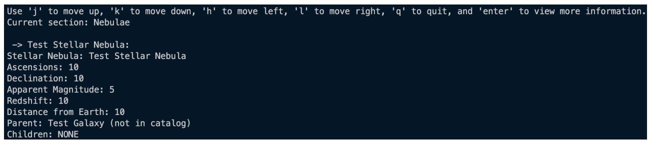
The "Input filenames" option requests the names of .csv files which contain entries for the astronomical objects and their attributes. Once these filenames are added, the option "Load files into catalogue" needs to be executed for the objects to be created, duplicates shown to the user and correct one deleted, relationships between objects to be set and each object added to the correct section in the catalogue.

"Add object to catalogue" will create a new menu to allow the user to manually create an object, as shown in Figure 5. The correct prompts for the object's information are given depending on the object type the user decides to add. The relationship of the user object to other objects in the catalogue is then also set once a valid object is inputted. As seen in Figure 6, the object is added to the correct section in the catalogue.



```
Use 'j' to move down the menu, 'k' to move up the menu and 'q' to quit.

    Make a Galaxy
 ->  Make a Stellar Nebula
    Make a Solar System
    Make a Star
    Make a Planet
```

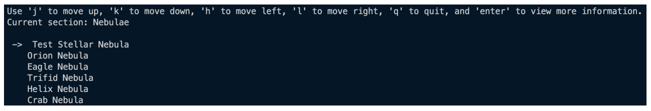
FIG. 5. Object menu for the user to manually create an object. The correct prompts are determined by which option the user presses.



```
Use 'j' to move up, 'k' to move down, 'h' to move left, 'l' to move right, 'q' to quit, and 'enter' to view more information.
Current section: Nebulae

 -> Test Stellar Nebula:
Stellar Nebula: Test Stellar Nebula
Ascensions: 10
Declination: 10
Apparent Magnitude: 5
Redshift: 10
Distance from Earth: 10
Parent: Test Galaxy (not in catalog)
Children: NONE
```

FIG. 6. Test object of Stellar Nebula type, which was made through the Object menu, correctly added and printed in the catalogue.

"Print catalogue" prints a new menu, displaying the names of every object in the Galaxy section. The section can then be changed using the 'l', shown in Figure 7 and 'h' keys to move down and up the class hierarchy. The user can then view the complete information about the object by moving the arrow to the desired object and pressing enter. The enter button needs to be pressed again to return to the catalogue.



```
Use 'j' to move up, 'k' to move down, 'h' to move left, 'l' to move right, 'q' to quit, and 'enter' to view more information.
Current section: Nebulae

 ->  Test Stellar Nebula
    Orion Nebula
    Eagle Nebula
    Trifid Nebula
    Helix Nebula
    Crab Nebula
```

FIG. 7. Catalogue menu where the section being printed was changed to the Stellar Nebula section using the 'l' key. Note that objects from a .csv file have been inputted into the catalogue.

"Save catalogue" will save the catalogue to a new .csv file called "saved_catalogue.csv". This will put all objects across all files and those inputted manually into one datasheet, where all duplicates have been deleted according to the user's preference.

IV. DISCUSSION & CONCLUSION

The code design follows good object-oriented principles by encapsulating functionality within classes, promoting code reusability, and managing object relationships effectively. Using smart pointers helps ensure proper memory management and avoids memory leaks. With this design, the star catalogue application provides a structured and modular approach for managing and accessing astronomical objects. This design allows for flexibility and extensibility, making it easy to add new types of astronomical objects in the future.

The UI provides a user-friendly way to interact with a star catalogue, allowing users to manage files, navigate through the catalogue, view object details, and perform various operations on the catalogue data.

In the future, classes derived from AstroObject can also have their own functions, for example, "explode" for stars into stellar nebula or black holes. Currently, there is no check whether the inputted .csv file is in the correct format, which could lead to segmentation faults if they vastly differ from the expected input. Moreover, a sort and filter algorithm could be implemented as additional functionality to the catalogue.

Word Count: 2344