Object-Oriented Programming in C++ Project Report

Daniel Thacker

September 19, 2022

Abstract

A programme for a catalogue of astronomical objects that can store both single variable member data and vectors of experimental data is presented. Data for types of stars, galaxies and planets may be inserted via keyboard or file and presented through the console or file. Objects may be selected and sorted by a variety of parameters and searched for by name or type by the use of a class template function and a database of shared pointers. A telescope offers a dynamic grid-based representation of the night sky from data held and can be moved with user input.

1 Introduction

The field of astronomy relies heavily on data storage and requires careful classification of astronomical objects, necessitating a logical database with which to store this information. The programme presented in this report aims to categorise and present a large amount of astronomical data for observed objects, such as magnitude, mass and distance, in an easy to use and clear way.

The core intention of the design was to be as useful for an astronomer as possible, allowing input from any number of files or the keyboard, storing both object data, such as galaxies, stars and planets, but also experiment data corresponding to each object. For instance, all galaxies have additional containers storing distance, velocity and velocity error for orbiting objects, data often used in dark matter experiments. These data are easily accessed and added to from the main menu by file or keyboard and may be presented on the screen or a file.

An abstract base class 'observed_object' begins the classification tree, followed by 'branch one', including galaxies, planets and stars. Branch two contains further denominations of these. For instance, the planet is a parent to 'gas giant' and stars contain 'cepheids'. In each branch an additional attribute is included.

The main functionality of the code derives from the 'menu' template class, accepting all of the observed object children, providing a platform for the user to add an object, read all data corresponding to one object type (e.g. all galaxies) and add data to a specific object (e.g. add velocity data to a specific galaxy).

In a similar manner, the 'database' class stores shared pointers to all the objects stored, allowing the user to search for a specific object name or type and output all

corresponding information for that object. An entire list of all stored objects, sorted in alphabetical or ascending order for any object attribute, is also presented by the database class.

As an additional feature, a 'telescope' class was included. This creates a dynamic grid with the x and y scale corresponding to bins of ascension and declination. The number of bins and the range of ascension and declination values are fully customisable by the user. The grid presents the positions of all currently stored objects in the sky using their assigned ascension and declination values and printing a marker, such as a 'G' for galaxy. The telescope field of view may be moved, allowing the user to explore the entire sky.

2 Code design and implementation

The class hierarchy begins with the abstract base class 'observed_object'. This class not only provides a template for all proceeding children with the use of pure virtual functions, but contains functions common to all observed objects, meaning functions do not have to be repeated in each child and are stored in a logical format. Proceeding children, or 'branch one' classes, are planets, galaxies and stars, each of which are treated as objects in a similar manner to their children. The branch one children serve as more generic observed objects, for instance, they are used if a star is discovered and it is unclear what further classification is required.

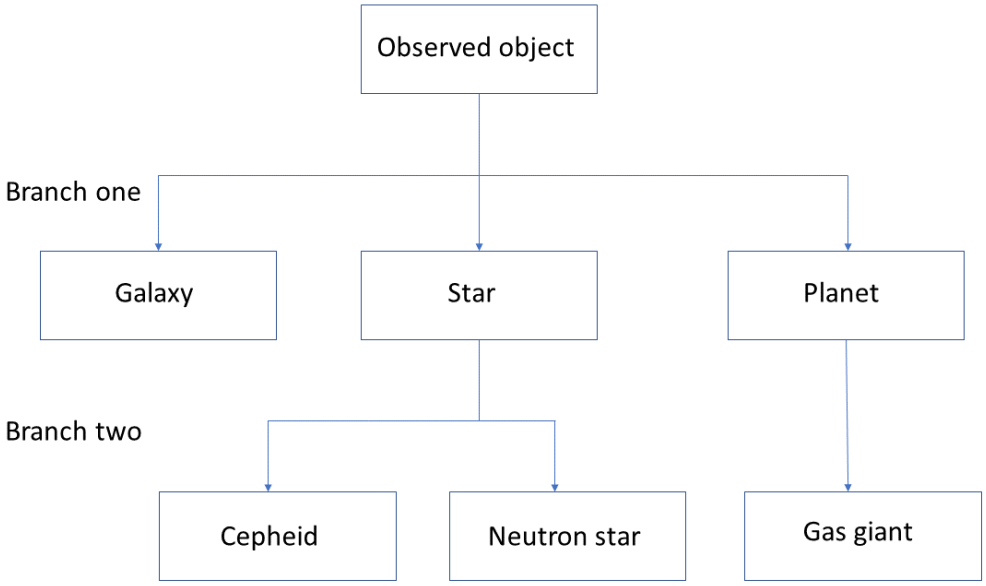

Figure 1: The class hierarchy followed by the star catalogue. The specialisation rule 'is a' is used. A neutron star is a star which is an observed object.

Branch two classes include gas giants, cepheids and neutron stars, following the rule of specialisation of 'is a'. A diagram of the class hierarchy is seen in Figure 1. Three children in each branch were considered sufficient to demonstrate the use of multiple inheritance and polymorphism.

As an example, the branch two classes 'neutron star' and 'cepheid' inherit functions from 'star' such as 'file_output' and 'print_first_vector', as well as vectors of flux and wavelength. These functions begin as pure virtual functions in 'observed object' and are overridden specifically for storing 'star' data. In this way, 'star' and its children are different from other branch one objects and utilise inheritance effectively. Both 'neutron star' and 'cepheid' override functions from star, such as 'get_type_name', and have functions that make them unique, such as 'set_period' for neutron star, containing a different attribute than 'cepheid', which has 'set_luminosity_period' instead.

Functions from 'observed_object' are also employed in branch one to make the code more readable and efficient. Functions such as 'add_experimental_data' and 'validate_file_input' mean large blocks of code do not have to be written repeatedly for each branch one child, allowing the branch one member functions to code specifically to their own needs and nothing else.

Each object is stored in a specific vector which is initialised at the start of the catalogue, along with the corresponding 'menu' object. Each of the 'cataloguing' classes, i.e. the menu, database and telescope, parse the vectors by reference to each other in a recurring switch case until the user decides to shut the application down. Referencing was used so multiple copies of each vector are not made and memory usage is efficient. Vectors were chosen as the primary container choice for a number of reasons. Vectors may be of varied length with a non index-based structure. This is ideal since it is unknown how many objects the user wishes to create at run-time and the order in which the observed objects are added is not important. The higher memory requirement for vectors compared to arrays was considered a worthwhile trade-off.

The 'menu' template class accepts all class types, allowing for code that is modular and efficient, exemplifying the practicalities of polymorphism. For instance, Figure 2 presents the 'add_object' function in the menu, called when the user decides to create an object by the keyboard, a new template object is initialised and an overloaded extraction operator is called. Each of the input classes has a different overloaded extraction operator that enables different values to be created by the user, with different prompts depending on which object the user has chosen to create. In a similar way, the member functions 'get_type_name' and 'get_name' have different operations depending on which class is inserted into the template, allowing for code that is interchangeable between each class.

```
//Add a new object to the vector
template<class stellar_object>
void menu<stellar_object>::add_object()
{
    //Call overloaded extraction operator for each input class
    stellar_object user_created_object;
    std::cin >> user_created_object;
    std::cout << "Created a " << user_created_object.get_type_name() << " called " << user_created_object.get_name() << std::endl;
```

Figure 2: The add_object function in the class template 'menu', demonstrating the effectiveness of polymorphism with an overloaded extraction operator.

A drawback of using a template class to dynamically create new objects is that each function used in the class must be shared by all input classes. This means the 'observed_object' base class must contain functions that do nothing and are overridden by a specific class, for instance 'set_period' for 'neutron star'. Additionally, branch one and two classes have different numbers of stored properties with different data types

depending on which class is being used. This means functions like 'many_object_input' must make great use of 'if' statements and the class comparison function 'std::is_same' to distinguish between how each class accepts data. This is computationally demanding and could be confusing.

The 'database' class is the primary method by which data is searched for and presented in a useful manner in the code. A vector of shared pointers to all of the object vectors is created when the database is initialised. Shared pointers were chosen to prevent memory leakage as they are automatically deleted when out of scope, as well as allowing for shared ownership semantics. Individual pointers, therefore, may be parsed to functions such as 'sort_database' any number of times, allowing multiple pointers to reference a single object.

```
//clear previous shared pointers to re-add the vectors to shared pointer vector
all_data.clear();
try {
    //Assign each vector a shared pointer to add to the entire data set for memory efficiency
    for (size_t k = 0; k < star_vector.size(); k++) {
        //allocate unique pointer to observed object
        std::shared_ptr<observed_object> pointer(new star(star_vector[k]));
        //pointer must be moved to insert into vector
        all_data.push_back(std::move(pointer));
    }
```

Figure 3: Code demonstrating how each vector is added to in the database. Repeated for each class.

The code in Figure 3 demonstrates how the database implements memory allocation. Initially, the vector of shared pointers is cleared, such that more data may be added after initially opening the database, without adding multiple copies of the same object. The shared pointer is moved so it can be added to the vector. The movement operator was used rather than copying because the shared pointer reference count is atomic [1]. Copying would have been many times slower since it would result in an increase in reference count, which is much more computationally expensive than simply taking a pointer and transferring ownership. A try/ except block is also used to handle failed memory allocation.

```
//Sort data alphabetically by using lambda function that compares 2 shared pointers from the all_data vector
if (user_input == 1) {
    std::sort(all_data.begin(), all_data.end(), [&](std::shared_ptr<observed_object> object1,
        std::shared_ptr<observed_object> object2) -> bool
        {return object1->get_name() < object2->get_name(); });
}
```

Figure 4: Code demonstrating the use of 'std::sort' and lambda functions to efficiently sort through the object data. This is repeated for many of the observed object attributes.

Having a large vector of shared pointers to all the currently held data is advantageous for a number of reasons. Demonstrated well in the 'sort_database' function, the entire database may be iterated over and sorted through without requiring computationally heavy 'for' loops. Big 'O' notation is a scale that describes computational demand when the argument tends towards a large value [2]. In the STL, the 'algorithm' header

has a number of efficient methods for handling large amounts of data, one of which is the 'std::sort' function, which runs as $O(N\log_2 N)$, much faster than a nested 'for' loop, which run as $O(N^2)$. The code in Figure 4 presents the use of the 'sort' algorithm used in conjunction with a lambda function that accepts two shared pointers to compare their names. The shared pointers are parsed by reference, meaning multiple unnecessary copies of each object are not made. The resulting output is a list with each object sorted alphabetically by name.

3 Results

There are many methods of data input and output in the presented catalogue. From the 'menu' class, many objects of a specific class can be created at once via a 'csv' file of eight columns for branch one classes and nine columns for branch two classes. One object at a time may be added by the keyboard, involving prompts from an overloaded extraction operator. The user can also add vector data (i.e. flux and wavelength data for stars) by the keyboard or a file of two columns of 'csv' data (three for galaxies because they contain three vectors). Opportunities for errors, such as empty cells in the input data file, are handled by the file input functions.

A list of all objects of a specific class along with their attributes is presented by the 'menu' class. Specific objects can be selected and have their vector data printed to screen or text file. Galaxies may have stars or planets added to them from the main menu, by the use of a contained vector of shared pointers to the 'observed object' class, member names of which are displayed when the galaxy data is printed.

The database class outputs data in a similar way, except all object data is presented and may be sorted by any attribute. A search algorithm can be used to select any object of a specific name by user input and outputs all contained data for it. Galaxies may have stars or planets added to them from the main menu, member names of which are displayed when galaxy data is printed.

Finally, a 'telescope' class presents the user with a visual representation of the night sky from the ascension and declination of the celestial objects they have inserted into the database. The telescope 'field of view' can be moved left, right, up or down by user input, incrementally changing the x and y axis values. An input via the keyboard allows the user to set the initial bounds, bin spacing and number of columns and rows by the keyboard. This was necessary to account for variations in user screen resolution and size.

4 Conclusion

A catalogue for the storage of data for a variety of observed objects is presented. Design features include input from keyboard and file and output to screen and text files in a number of ways. Main features include a template class and class of shared pointers for object storage.

The design could be readily expanded upon and improved. For instance, more object data could be stored to produce a more complete astronomical picture. Both more branch one and two classes could be added, along with more member data attributes.

Restrictions were placed on the versatility of the code by the use of the 'menu' template class. All objects were required to have the same parameter list as 'observed object', with branch two data having to be added by the use of 'if' statements and setter functions. This somewhat limited the range and variability of the data that could be contained.

A map of multiple keys, each corresponding to an astronomical property, could have been used to store the data instead. Maps offer simple template arguments to compare keys which would have provided a faster search and comparison speed than a vector of shared pointers. When inserting into a vector, the vector must be resized, necessitating a copy of the entire array to new memory. A map does not require this and has a quicker insertion time that scales as $O(\log_2 N)$. This was not implemented due to the complexity of creating a dynamically sized map from user input.

In addition, it is quite laborious for a user to go through each object and input experimental vector data by file for each object. A method to accept one file and supply all objects with experimental data would decrease the work required from the user. Similarly, a more aesthetically pleasing interface with clickable option buttons would improve the user experience.

References

[1] *Encyclopedia of Database Systems*. Springer New York, New York, NY, second edition. edition, 2018.

[2] Sammie Bae. Big-o notation. In *JavaScript Data Structures and Algorithms*, pages 1–11. Apress, Berkeley, CA, 2019.