An astronomical catalogue in C++

Timothé Rhein

10139740

Department of Physics and Astronomy

The University of Manchester

Project Report

Object-Oriented Programming in C++

May 2022

Abstract

An astronomical catalogue was written using object-oriented programming in C++. The program stores data from astronomical observations and allows the user to search using different criteria. This report details the advanced coding features such as the use of smart pointers, run-time polymorphism, inheritance and template functions used in the code.

1. Introduction

Humans have been studying and recording stars for thousands of years. The first star catalogue dates from approximately 1155 BC where ancient Babylonians recorded the positions of 36 stars onto clay tablets [1]. Historical recordings of celestial objects and astronomical phenomena have allowed scientists to study the changes in Earth's rotation throughout recorded history and provide information on the forces governing those fluctuations [2]. Some star catalogues, such as the Bonner Durchmusterung from 1903 which recorded over 320 000 stars [3], contained immense amount of data before the advent of computerisation. The digitalization of astronomical catalogues makes processing data from the incredible number of observations manageable and accessible worldwide. The current largest catalogue, collected by the Gaia satellite, contains over 1.8 billion objects [4], which demonstrates the need for efficient software to store and process this data.

This report details the code of a star catalogue program written in C++ using object-oriented programming. The main aim of the program is to store and search data taken from observations of celestial objects. A class hierarchy of celestial objects was designed to use run-time polymorphism to simplify input and output of data. Run-time polymorphism in this program makes use of smart pointers rather than raw pointers in order to avoid memory leaks caused by the omission of a `delete` statement, meaning the stored memory will be inaccessible and useless for the duration of the program. Classes containing dates and angles are used to manipulate different formats, and a `DataContainer` class was designed to store and search the data. Finally, an `Interface` class controls the command-line interface with the user.

2. Code design and implementation

The code comprises of 4 main class types. Firstly, the `CelestialObject` abstract base class and its derived classes store data about individual observations. The class is designed to easily input and output data to and from a stream to improve code clarity. In support of this the `Date` and `Angle` classes were created to manipulate these quantities since they are crucial to astronomical observations. The observations are stored in the `DataContainer` class which controls the manipulation of data. The interaction between the user and the program is through the console is controlled by the `Interface` class. This section will focus on the advanced coding features used to implement this program.

2.1. `CelestialObject` class hierarchy

The `CelestialObject` abstract base class contains variables for the basic observational data that could be taken for any celestial object: its name; the experiment it was measured in; the apparent magnitude, mass and distance of the object; the date of observation (`observational_epoch`); and the position in terms of right ascension and declination. The classes `IntrasolarObject` and `ExtrasolarObject` are abstract base classes derived from it, and the classes representing physical celestial objects are derived from those, as shown in Figure 1.

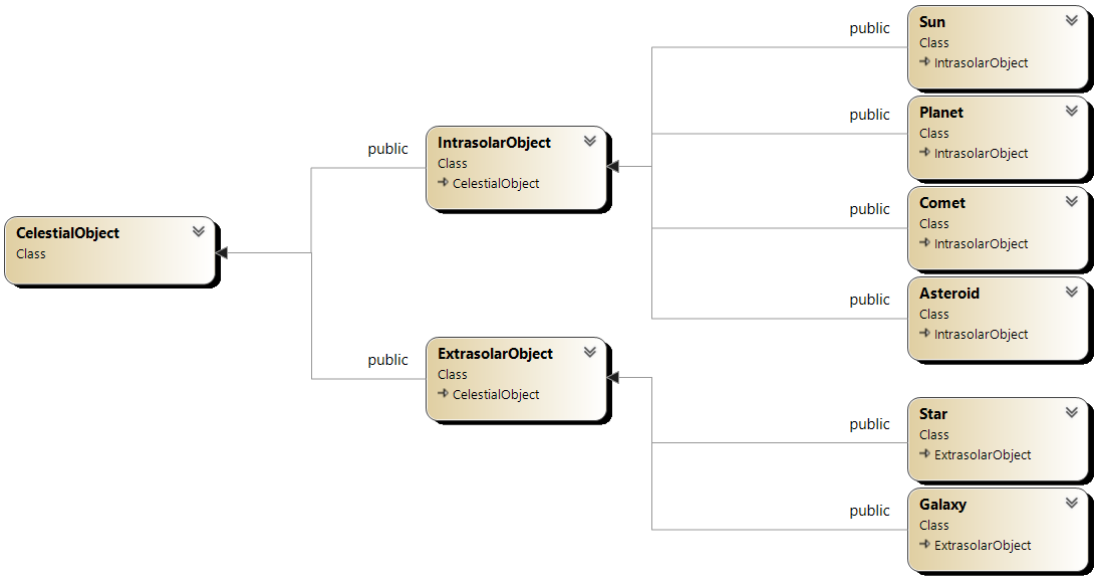

Figure 1. Class diagram for `CelestialObject` class hierarchy.

Since the observational data defining an object would be read from either a file or a terminal, it was logical to override the stream insertion (<<) and extraction (>>) operators. However, since these are friend functions they cannot be overridden in derived classes like other member functions. In order to make use of run-time polymorphism and not need to implement these operators in each derived class the operators were only defined in CelestialObject, while two virtual functions `iostream_input_body(std::istream& is)` and `iostream_output_body(std::ostream& os)` were created to handle the behaviour of the operators for the different objects. This implementation (shown in Figure 2) effectively simulates virtual friend functions. The `inline` keyword requests the compiler to expand the function in the line it is called in to increase efficiency when calling short functions [5].

```
// Virtual friend functions implementation.
inline std::ostream& operator<<(std::ostream& os, const CelestialObject& object)
{
	object.iostream_output_body(os);
	return os;
}

inline std::istream& operator>>(std::istream& is, CelestialObject& object)
{
	object.iostream_input_body(is);
	return is;
}
```

Figure 2. Implementation of virtual friend functions. `iostream_input_body(std::istream& is)` and `iostream_output_body(std::ostream& os)` are virtual functions to be overridden in the derived classes.

The overridden virtual friend functions output data in the same format it is read in to ensure that data saved in a file can be read again. The format for insertion into an `ExtrasolarObject` derived class is 12 comma-separated fields, while for an `IntrasolarObject` derived class it is 8 comma-separated fields, excluding the object

type specifier that the user needs to input first. Formatting details can be found in the About section of the program.

The `CelestialObject` class hierarchy also makes use of static member variables in order to count the number of instances that are present at any one time in the program. The static variable `use_count` is implemented in each class of the hierarchy so that the number of derived-objects can be counted. The count is accessed using the static member function `get_use_count()`.

Error handling in the `CelestialObject` derived insertion operators is done using `try` and `catch` blocks. Errors such as the user entering an incorrect number of fields or a string in a double field will throw exceptions. When these exceptions occur, since the object is already created but will hold incorrect or incomplete data, the variable `CelestialObject::incorrect_read` will be set to true, allowing the `DataContainer` class to later use this as a condition to remove these defective objects.

2.2. `Date` and `Angle` classes

The `Date` and `Angle` classes were created for easy insertion or extraction using different formats, as well as to prepare for future functionality of this astronomical catalogue.

The `Date` class stores the date and time as a Julian date, often used in astronomy. Conversions to and from calendar dates are implemented to facilitate date entry for the user. Several functions such as `get_obliquity_ecliptic()` which are unused in the program are left in to show the potential for this class to be used in calculating positions of stars at different dates and their azimuth and elevation when viewing from a specified position on Earth.

The `Angle` class is also used to facilitate angle entry for the user. Its derived classes `RightAscension` and `Declination` are used to specify the positions of stars within the `CelestialObject` hierarchy. Unlike the `Angle` class, they check their angles are within acceptable ranges : between -90º and 90º for declination and between 0º and 360º for right ascension. Within the overridden `iostream_input_body(std::istream& is)` function in both `ExtrasolarObject` and `IntrasolarObject` classes run-time polymorphism is used to create the `RightAscension`, `Declination` and `Angle` variables to use as parameters to the `set_data(…)` function, as shown in Figure 3. The smart pointer `unique_ptr` is used here to prevent memory leaks which may be caused by the omission of `delete` when using raw pointers. While iterating over the vector of pointers to the base class `Angle`, the string stream `iss` containing the entry is inserted to the `Angle` derived class to create the appropriate object.

```
// entries[7] holds the observational_epoch field.
std::istringstream iss{entries[7]};
Date obs_epoch;
iss >> obs_epoch;
iss.str("");
iss.clear();

std::vector<std::unique_ptr<Angle>> angles;
angles.reserve(4);
angles.push_back(std::make_unique<RightAscension>());
angles.push_back(std::make_unique<Angle>());
angles.push_back(std::make_unique<Declination>());
angles.push_back(std::make_unique<Angle>());

// entries[8] through entries[11] hold the angle fields
// (ra, ra yearly change, dec, dec yearly change)
for (size_t i{0}; i < 4; i++) {
      iss.str(entries[i + 8]);
      iss >> *angles[i];
      iss.str("");
      iss.clear();
}

set_data(entries[0], entries[1], entries[2], entries[3], std::stod(entries[4]),
      std::stod(entries[5]), std::stod(entries[6]), obs_epoch, (*angles[0]),
      (*angles[1]), (*angles[2]), (*angles[3]));
```

Figure 3. A code snippet showing run-time polymorphism from within the ExtrasolarObject::iostream_input_body(std::istream& is) function.

2.3. `DataContainer` class

`CelestialObject` derived objects are stored in the `DataContainer` class. `DataContainer` uses the variable `data` of type `std::set<std::unique_ptr<CelestialObject>>` to store the data. Since this is a `set` of base class pointers, all objects contained within it are accessed using run-time polymorphism. A `set` is a container from the Standard Template Library. The container `set` was chosen because the time complexity of operations such as `std::set::insert` or `std::set::find` are proportional to the logarithm of its size. This behaviour is advantageous in an astronomical catalogue since they may contain millions of objects.

Another potential advantage of using a set is the possibility of ordering elements based on a custom comparator function. This would allow data to be sorted based of, for example, an object's name, experiment or the date of observation. This was attempted by overloading the operator `<` for the `CelestialObject` class and writing a lambda function to use as the comparator. However, due to time constraints and the difficulty in passing a `unique_ptr` to a lambda expression this functionality was abandoned.

In order to insert an object into `DataContainer` the insertion operator was once again overloaded, as shown in Figure 4. The first entry defining the object type (whether it is a star, planet, etc.) is first extracted from the stream and then compared to the expected string for each available type using the `DataContainer::create_object(…)` template function (see Figure 5). The expected string to create a particular object type is found by calling the static member function `type::get_object_type()` which will return the expected string without needing to

first create the object. If the condition is fulfilled, a `unique_ptr` to that type is created, the rest of the stream is inserted into the object and the `unique_ptr` is then "moved" in order to be inserted into the `data` variable.

```
std::istream& operator>>(std::istream& is, DataContainer& data_cont)
	// Inserts an object into the data container.
{
	std::string object_type;
	bool data_read{false};

	is >> std::ws;
	std::getline(is, object_type, ',');

	// Condition to not run this code if a new-line character is encountered.
	if (object_type.size() > 1) {
		// Will create an object only if the object_type matches with the
		// class.
		data_cont.create_object<Star>(is, object_type, data_read);
		data_cont.create_object<Galaxy>(is, object_type, data_read);
		data_cont.create_object<Sun>(is, object_type, data_read);
		data_cont.create_object<Planet>(is, object_type, data_read);
		data_cont.create_object<Comet>(is, object_type, data_read);
		data_cont.create_object<Asteroid>(is, object_type, data_read);

		if (data_read == false) {
			std::cout << "Error: data not read properly. Please ensure
				the object type specifier is correct.\n";
		}
	}
	return is;
}
```

Figure 4. Overloading of `operator>>` for the `DataContainer` class. Found in DataContainer.cpp.

```
template<class type>
inline void DataContainer::create_object(std::istream& is, const std::string
object_type, bool& data_read)
	// Creates an object if object_type is equivalent to
	// type::get_object_type(). type must be derived from CelestialObject.
{
	if (object_type == type::get_object_type()) {
		auto object = std::make_unique<type>();
		is >> *object;
		data.insert(std::move(object));
		data_read = true;
	}
}
```

Figure 5. Template function `DataContainer::create_object(…)`. Found in DataContainer.h.

2.4. `Interface` class

The behaviour of the user interface for this program is controlled by the `Interface` class, and more specifically by the `menu()` function and the `input_commands` variable. The `menu()` function consists of several nested switch statements that define the user interface's menu and is repeatedly called until the `exit` variable is true, which

only occurs when the user selects exit. The `input_commands` vector of integers controls the state of the menu and what it displays: only the deepest level of the code within the switch statement will be run. The `menu()` function is rerun every time the user reaches the end of an option chain by the `while` loop in `Interface`'s constructor.

```
Interface::Interface()
{
	read_data_from_file("exit_save.txt");
	welcome_text();
	while (!exit) {
		data_container.check_incorrect_read();
		menu();
	}
	save_data_to_file("exit_save.txt");
}
```

Figure 6. `Interface` constructor.

The `Interface` constructor is shown in Figure 6. The program's behaviour is completely contained within this constructor. An exit save feature is implemented so that all data present in the program on exit is saved and will be directly loaded into the program on the next start. Within the `while` loop the `data_container.check_incorrect_read()` function removes all objects that have had an error occur while being created, and so ensures no such object is present when the user is interacting with the program.

3. Results

The main menu is shown in Figure 7 after having loaded data from the file "nav_star_eq.txt" (data adapted from [6]), which contains names, constellations, positions and yearly changes in positions from all stars used for celestial navigation and the file "solar_system_objects.txt" which contains data for a few solar system objects. Choosing option 5 from the main menu shows you output shown on the right of Figure 7, which uses the static `use_count` variable for each class to display the number of objects present of each type. Figure 8 shows the program after searching this data for the name "Halley" and choosing to output the data to the terminal.

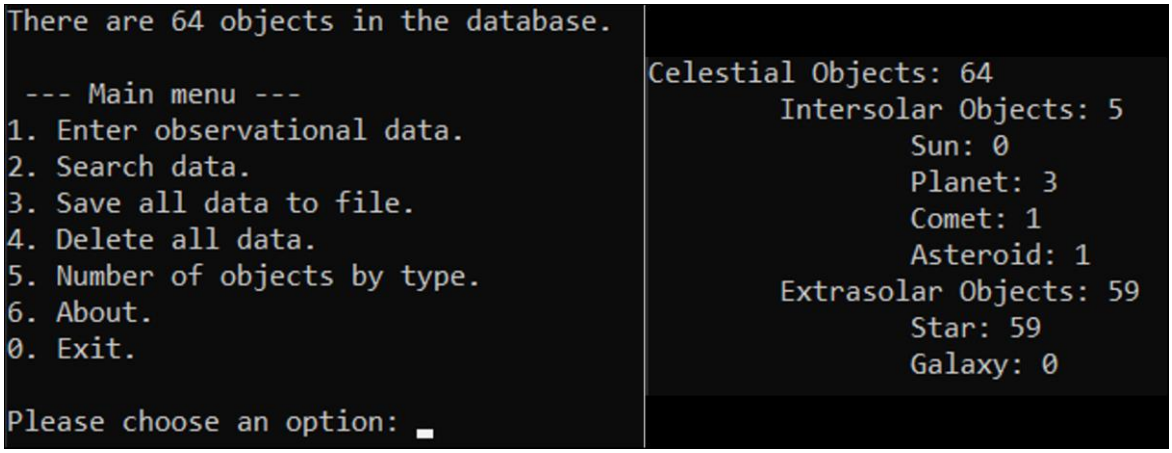

Figure 7. The image on the left shows the main menu after loading data from the files “nav_star_eq.txt” and “solar_system_objects.txt”. The image on the right shows the output after choosing option 5 from the main menu.

```
Choose a field to search by:
1. Name.
2. Experiment.
3. Object type.
0. Back.

Please choose an option: 1
Please enter the name of the object you want to find: Halley


Number of results: 1

1. Print details in terminal.
2. Output results to file.
0. Back.
Please choose an option: 1


comet, Halley, test_experiment, 30.97, 1.1e-16, 0.000172, 2022-5-21 15:40:22, hms 8 14 58, dms 3 3 15
```

Figure 8. Output of the program when searching for the comet named “Halley”.

4. Conclusion

This program makes use of object-oriented programming in the C++ language to handle data from astronomical observations. It uses advanced features such as smart pointers and template functions to avoid problems such as memory leaks and to improve the clarity and brevity of the written code.

The next improvements to this star catalogue would come in the form of using the data contained within `ExtrasolarObject` derived classes to predict the positions of stars at different instances. The classes `Date` and `Angle` and their derivatives already have the core functionality required for this next step.

Improvements to the code could include the addition of a comparator function in the `DataContainer::data` variable using a lambda closure that can access information from the objects pointed to by the `unique_ptr`’s. This would sort the data within the

set improving the readability of the program's output and ensure that duplicate data is not held within the `set`. Addressing this problem would also mean a lambda closure could be used as a conditional function in the `std::find_if` algorithm. This conditional function would make it much easier to implement different searches, such as finding objects within a range of apparent magnitudes.

References

[1] M. A. D. M. Azhar, "The historical development of star catalogues," in *The 4th East Asia & Southeast Asia Conference on Philosophy of Science 2014*, 2014.

[2] F. R. Stephenson, L. V. Morrison and C. Y. Hohenkerk, "Measurement of the Earth's rotation: 720 BC to AD 2015," *Proceedings of the Royal Society A,* vol. 472, no. 2196, 2016.

[3] F. Argelander et al., "Bonner Durchmusterung : I/122," Strasbourg astronomical Data Center, 24 September 2012. [Online]. Available: https://cdsarc.u-strasbg.fr/cgi-bin/Cat?I/122#/seealso. [Accessed 20 May 2022].

[4] European Space Agency and Gaia Data Processing and Analysis Consortium, "Gaia Early Data Release 3," ESA, 2021.

[5] "`inline` specifier," cppreference.com, 27 April 2022. [Online]. Available: https://en.cppreference.com/w/cpp/language/inline. [Accessed 20 May 2022].

[6] B. D. Yallop and C. Y. Hohenkerk, "Astronomical algorithms for use with micro-computers," UKHO HM Nautical Almanac Office, 1989.

Word count: 2166.