# Creating A Simple Manager for Astronomical Data

A program has been created to manage basic data from telescope experiments. This has been achieved by the use of an abstract class representing observable objects with derived specialisations for specific types of objects. Container classes have also been defined to handle storage of these objects, to allow for ownership of objects in orbits around others and to allow for dynamic creation and manipulation of data and objects at runtime. A basic command-line-style interface has been provided to allow dynamic use of the program by the user at runtime and can handle several defined instructions such as selection, data reports and the exporting/importing of data to/from file. Future revisions of the program could account for more edge case uses and handle more robust and detailed data.

## Introduction

The aim of this project is to create a simple celestial object manager to store and operate on basic data from several experiments. It should be simple for a user to operate at run time and allow for all of the data that a user could need to enter to be handled as dynamically created objects. Data entered by the user should be relatively easy to access within the program whilst avoiding confusion between objects due to similar properties and multiple separate sets of data should be able to be handled at runtime as needed. In addition, the ability to write data to files which can be read from by later sessions of the program to restore data objects would make the program more convenient and usable in the long term than manually entering data in each program session. Overall, this would produce a convenient, simple and somewhat user-friendly interface for keeping track of basic data from astronomical experiments over a long time span.

To implement methods that have allowed the specification to be met, the following headers from the C++ Standard Library have been used:

- <vector> - for storing objects within the program.
- <string> - for use of C++-style strings.
- <iostream> - for the use of cin, cout and getline().
- <fstream> and <filesystem> - to import files to and export files from the program whilst checking existence within a specified location.
- <ctime> - for timestamping file names when exporting data in the case of a file with the same name already existing.
- <sstream> - for the use of stringstreams whilst parsing files.
- <algorithm> - for the use of sorting and finding algorithms over vectors.
- <memory> - for the use of smart pointers instead of raw pointers.

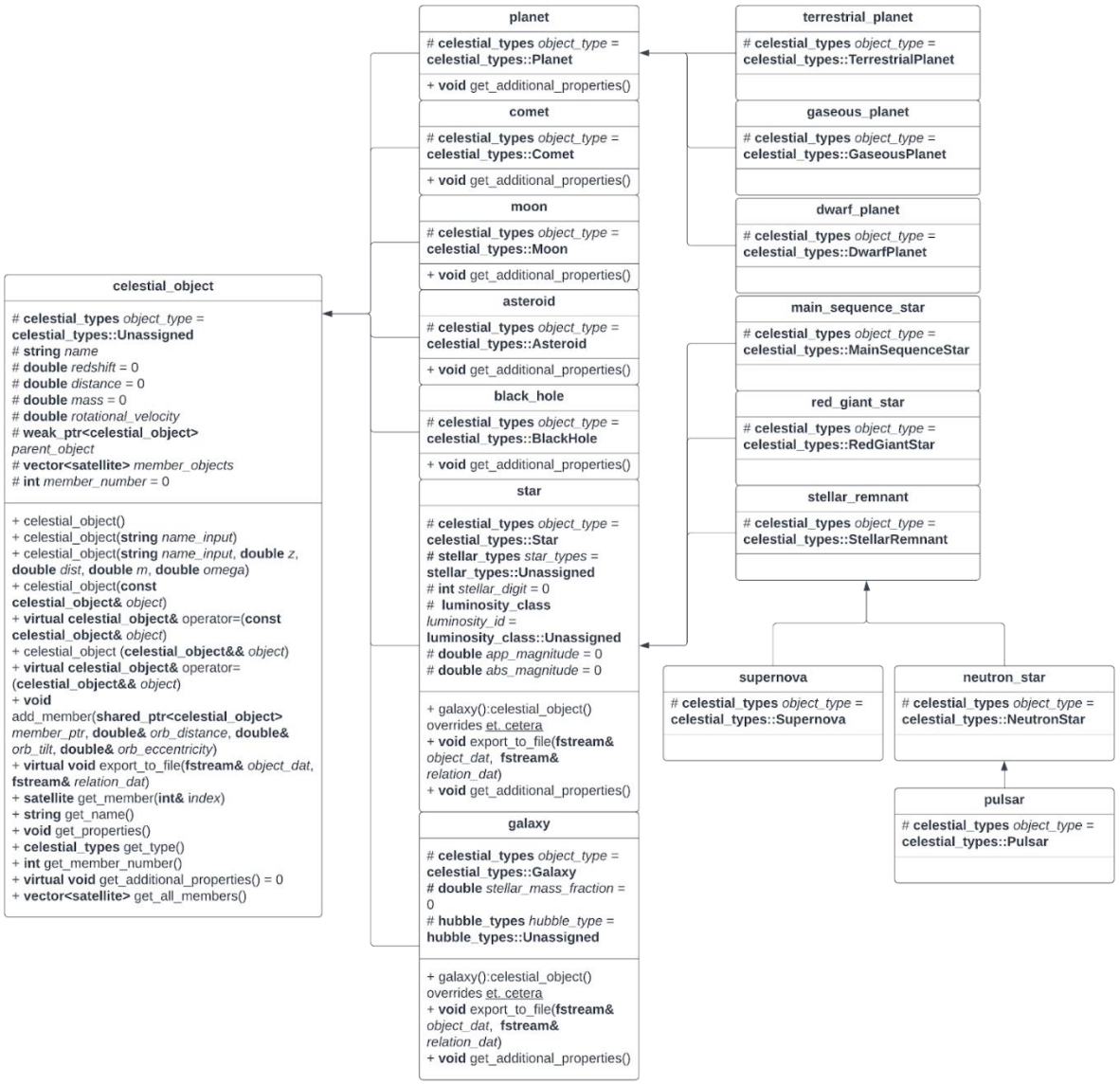
## Class & Function Design

All classes designed for this project have been defined in a header called "celestial_objects.h" within a namespace called *celestial_objects* with multiple-line non-constructor and methods defined in the source file "celestial_objects.cpp". The behaviour of the program at runtime is scripted in the file "catalogue_project_main.cpp", which contains the **int** *main()* function.

Enum classes are used to create a limited range for key variables used in the program - Hubble types for galaxies, stellar types and luminosity numbers for stellar objects, object types and parameters (for representation of attributes when taking user input). Each enum class has a vector of strings to represent each value in plain text during output and to compare user input to.

The classes used within this program are organised into an inheritance tree of classes representing celestial objects, a satellite container class representing an object orbiting

about another object and a catalogue container class, used to contain celestial objects observed in an experiment.
.




*Inheritance tree for celestial_object classes*

An abstract base class **celestial_object** is used as the base class for all objects and hence contains all attributes and methods common to all of the objects. All attributes are protected so that they can be inherited by only accessed by public setter and getter methods and friend class methods, hence all derivative classes must use public inheritance. The attributes of this base class are:

- **std::string** *name* - the name of the object, used as a unique identifier within the program
- **celestial_objects::celestial_types** *object_type* - defines the type of object that is being referred to
- **double** *mass* - the mass of the object, in $M_{Sun}$
- **double** *redshift* - the redshift, z of the object
- **double** *distance* - the distance to the object in parsec (this is not always perfectly correlated with the redshift due to gravitational effects)
- **double** *rotational_velocity* - the rotational velocity of the object

- **std::weak_ptr<celestial_object>** *parent_object* - a weak pointer to the parent object of this object (nullptr if it has no parent). Weak pointers have been used to prevent cyclical references (and hence memory leakage) from shared pointers.
- **std::vector<celestial_objects::satellite>** *member_objects* - a vector containing all children of this object as stored in satellite objects.
- **int** *member_number* - keeps track of the number of objects in the *member_objects* vector

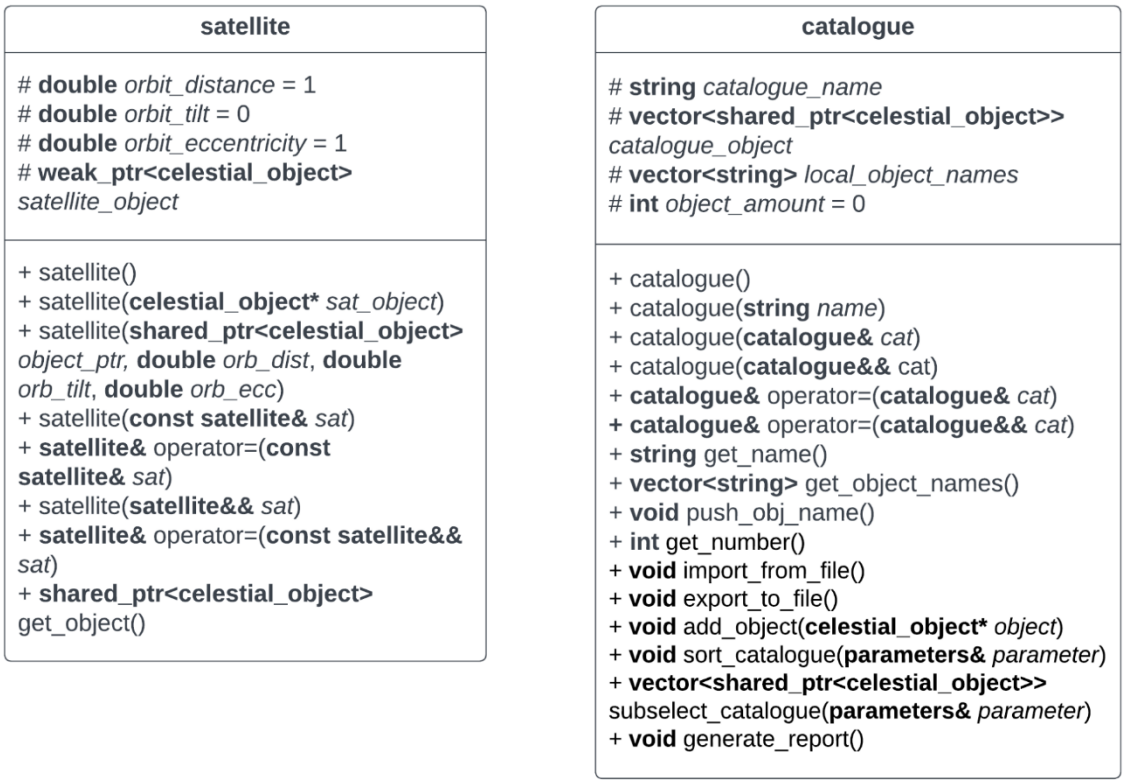
With regards to parent hierarchies, objects can only be parented to another if: they do not have a parent, the parent is not itself and parenting the object does not create a closed loop (and hence lead to smart pointer memory leakage, though weak pointers are deliberately used to avoid this). Using smart pointers to a base class raw pointer makes use of polymorphism to allow all of the objects created to be stored alongside each other in single vectors, hence greatly simplifying many aspects of the program.

There are also several shared public methods:
- Constructors - used to construct the object in question, with a parameterised constructor for a given name only (which is the only one called by a user), a fully parameterised constructor (used during import) and a move constructor and a copy constructor for efficient use of memory when constructing from lvalue and rvalue objects.
- Destructor - this is the default destructor as memory management is handled via the use of smart pointers.
- Copy and move assignment operators - these are used to handle assignment from other objects in an efficient way and are both virtual to allow for overriding in derived classes.
- **void** *add_member*(**std::shared_ptr** *member_ptr*, **double&** *orb_distance*, **double&** *orb_tilt*, **double&** *orb_eccentricity*) - adds a satellite containing *member_ptr* with the orbit parameters (distance (in pc), tilt and eccentricity) member to the *member_obejcts* array.
- **void** *remove_member*() / *remove_member*(**int** *index)* - removes an object from *member_objects* at the end/given index respectively.
- **celestial_objects::satellite** *get_member*(**int** *index*) - returns an object from *member_objects* at the given index
- **std::vector<celestial_objects::satellite>** *get_all_members*() - returns the *member_objects* vector.
- **std::string** *get_name*() - returns *name*
- **void** *get_properties*() - prints the attributes of the object
- **virtual void** *get_additional_properties*() - prints derived-class specific properties, in the base class this is a pure virtual function hence making it abstract.
- **virtual void** *export_to_file*(**std::fstream&** *object_dat*, **std::fstream&** *relationship_dat*) - writes the object data and the parenting relationships with this object to separate files, called from a **celestial_objects::catalogue** object containing a **celestial_object**.

When constructors and destructors are used in derived classes, they are explicitly passed to their base class constructors to avoid having to write redundant lines for each of the five constructors in every class and to increase program efficiency.
There is one main derived class branch from the base class: the **star** object branch which defines any non-galactic luminous object, adding the following protected attributes:
- **celestial_objects::stellar_types** *star_type* - defines the stellar type of the **star** object as according to the Hertzsprung-Russel diagram, O - K
- **int** *stellar_digit* - defines the digit that follows the stellar type (e.g G7)

- **celestial_objects::luminosity_class** *luminosity_id* - gives the luminosity class of the stellar object, 0, Ia-Ib, II - VII
- **double** *abs_magnitude* - represents the absolute magnitude of the object
- **double** *app_magnitude* - represents the apparent magnitude of the object

The other notable derived class is the **galaxy** class, which adds the two magnitude doubles as in the star branch alongside the attributes **celestial_objects::hubble_types** *hubble_types*, which is the Hubble type of the galaxy and **double** *stellar_mass_fraction*, which gives the proportion of the galaxy which is luminous.




*Class definitions for the containers*.

In addition, there are two container classes: **satellite**, which contains a weak pointer to a **celestial_object** (with a corresponding access function) and three **double** attributes representing the orbit distance, tilt and eccentricity and the **catalogue** class, which represents an experiment. **Catalogue** contains a vector of shared pointers to **celestial_object** instances *catalogue_objects*, a **std::string** *catalogue_name* and an **int** *object_amount* tracking the number of objects in *catalogue_objects*. The names are also stored in a **std::vector<std::string>** to ensure that names are not used twice in the same catalogue. In addition, the **catalogue** class has the following unique methods:

- **void** *export_to_file*() - creates/opens two data files and calls the corresponding *export_to_file*() method in each contained **celestial_object**
- **void** *import_from_file*() - imports two given .dat files (object and relationship data), parses the data given and creates and parents objects based on the data.
- **void** *sort_catalogue*(**parameters&** parameter) - argsorts *catalogue_objects* based on the given parameter, in ascending order. Lambda expressions are used alongside defined functors to achieve this.
- **std::vector<std::shared_ptr<celestial_objects::celestial_object>>** *subselect_catalogue*(**celestial_objects::celestial_types&** *type*) - returns a selection with the given object type within the catalogue

- **void** *generate_report*() - generates a report on the distribution of objects in the catalogue

Within each of the classes, exceptions that could occur at runtime due to input are handled by try/catch blocks, notably **bad_alloc** errors given the extensive usage of dynamic memory, standard bad assignments due to incorrect/invalid data or other states of variables that should not be allowed by the program but may have not been directly accounted for elsewhere.

**Runtime Behaviours**

The runtime behaviour of the program is defined within "catalogue_project_main.cpp". Within this source file, several global variables are defined:

- **command** and **contexts** enum classes, with corresponding string vectors for input/output.
- A string vector *current_object_names* keeping track of all the names currently in use
- A **celestial_objects::catalogue** vector *catalogues* keeping track of all catalogues loaded in the program (and thus allowing for multiple to be loaded at once)
- A weak pointer to a **celestial_objects::catalogue** object and a weak pointer to a **celestial_objects::celestial_object** object within the catalogue, both used for runtime selection.
- A **bool** *quit*, used to track whether the program should still be running. The default value is **false**, corresponding to the program running.

A function, **void** *user_interface*() is also defined to handle user input and use the input to operate on objects of classes within **celestial_objects**. This user interface is not graphical, as that is too far out of scope for this project and presents compatibility issues dependent on the operating system, so a simplified command-line style interface is used instead, using command words and context words to complete operations. User input is error checked so that a valid command is entered and then handled by a switch statement over the **command** enum class. This interface acts as a frontend for calling methods contained within the objects defined in the "celestial_objects.h" header, as allowing objects to operate on and communicate to each other is a simpler way of handling processes and uses the paradigms of object-oriented programming.

The given commands are:

- *Select <catalogue/object/selection>* - selects a catalogue of a specified name/an object of a given name within a selected catalogue/a selection of objects within the catalogue of a given type.
- *Create <type>* - creates an object of a given type, first asking for an unused name and then passing to the relevant name constructor of the type of object specified
- *Parent <name>* - parents the currently selected object within a catalogue to the one with the given name, if it exists.
- *Import* - constructs an empty catalogue, calls the *import_from_file*() of that catalogue and adds the catalogue to the *catalogues* vector.
- *Export* - calls on the *export_to_file*() method in the selected catalogue
- *Report* - generates a report on the selected catalogue by calling its *generate_report*() method
- *List <catalogues/objects>* - lists all of the catalogues currently loaded in the program/all of the objects in the selected catalogue
- *Sort <parameter>* - calls the *sort_catalogue*() method in the selected catalogue based on a given parameter
- *Help* - Lists commands and gives general information about the program
- *Quit* - sets *quit* to **true**, hence leading to the program terminating

When the program runs, a catalogue is created for an included test file to demonstrate the program's capabilities and then *user_interface*() is called while *quit* is **false**. The program will not terminate until the user uses the *Quit* command or manually terminates it.

## Conclusion of Project Capabilities and Possible Improvements

Overall, this project has met the aspects of the minimum class design specification: an abstract base class is used for observable object, a suitable hierarchy of classes has been defined to make use of inheritance, container classes are used to handle operations and storage and virtual functions are used for common functions with specific behaviours in derived classes. The minimum functional behaviour has also been met: data files can be read in to create objects alongside user input, containers are used throughout the program to handle multiple experiments and allow for selection by object type, data can be extracted to console and the program can generate reports.

In addition to this, the program can also export data to files which can then be read in by the program, parent objects to each other, handle basic orbital information using the parent hierarchy and take user input to be interpreted as commands. The use of dynamic memory allocation via smart pointers also allows for extensive use of the program at runtime by allowing the storage of selected objects and modular functions stored as methods within objects. The use of headers and multiple source files alongside a custom namespace improves the readability of the source code and reduces redundant code.

However, there is one major issue at run-time: *get_additional_properties*() attempts to access illegal memory for objects created within *user_interface*() despite the use of copy and move semantics. The current workaround for this is to export a catalogue with user-defined objects, import it in another session and then generate a report from that catalogue as a prenament fix has been increasingly hard to find.

Possible future additions and improvements I could make to this project include:

- A class to handle celestial distances, given that all distances are currently handled in parsecs only, which becomes inconvenient for objects orbiting stars or planets.
- A class to handle celestial coordinates, so that they can be specified for all objects and hence extend the functionality of the program.
- Functions to handle several basic astronomical calculations, such as orbital positions at given times, finding recessional velocities from distances and redshifts and to infer properties such as radius from others such as mass.
- A basic graphical interface using an open-source and well-supported GUI toolkit such as Qt to further increase ease of use by users.
- If the above features are added, it could be possible to create an interactive and accurate star map by interpreting the data to parse object colour, shape, apparent size and brightness or to create a Hertzsprung-Russel diagram of all of the **star** objects inside a given **catalogue** object.
- Rewriting all error checks so that they use try/catch blocks to make the program far more consistent and robust against exceptions.