**Assignment 2**

**Introduction**

For the purpose of this assignment, we decided to implement a generic pattern for a behavioural design pattern namely the Observer pattern. This design pattern is used to create a one-to-many dependency between objects, so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically, which is useful in scenarios where an object should broadcast changes to other objects without knowing who those objects are. The observer pattern usually has the following use cases:

* Event handling systems: GUI frameworks, where user interactions including clicks and key presses need to be handled by multiple components.
* Model-View-Controller (MVC) architecture: Ensures that views update automatically when the data model changes.
* Real-time data streaming: Applications including stock tickers where multiple displays need to be updated in real time.
* Notification systems: Email, messaging or logging systems where various parts of the system need to be notified of certain events.

The observer pattern offers flexibility to manage dependencies and ensures consistency for applications where state changes may need to be propagated to multiple dependent objects. We made use of 2 scenarios in this assignment for testing purposes to show that the implementation works with various scenarios.

**Design Choice**

The Observer design pattern makes use of 2 class templates namely, observer and subject, which enables us to implement the Observer pattern in a type-independent manner. The observer class template defines an interface where concrete observers must implement, it uses a template parameter ‘T’ to represent the type of data that the observers will receive. The Subject class template manages the list of observers and provides methods to either add, remove and/or notify them. To use the observer and subject class templates a concrete implementation of both classes for a specific type of data will need to be created.

* Observer Template: Defines a generic interface with an update method that concrete observers must implement.
* Subject Template: Manages a list of observers and provides methods to add, remove, and notify them.
* Concrete Implementations: Specific types like int are used to create concrete observers and subjects, demonstrating how the pattern can be applied to real-world scenarios.
* Main Function: Illustrates adding, notifying, and removing observers, showcasing the dynamic nature and flexibility of the Observer pattern.

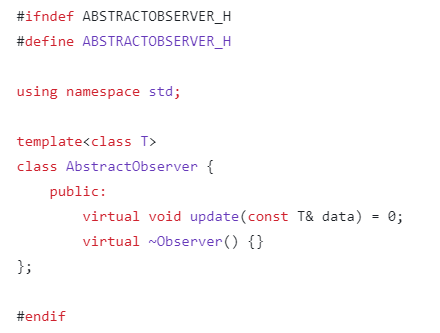
Some of the benefits of using templates to create a generic implementation of the Observer pattern are as follows:

* Independent types – through the use of templates the observer and subject classes can handle any data type which allows for reuse of the code without any duplications or rewrites
* Reuse of code – the same classes can be used in various parts of an application without redundancy
* Maintainability – any and all changes or bug fixes within the template are automatically applied to all instances of the template
* Compile-Time type checking – the use of templates ensures that type mismatches are caught at compile time instead of at runtime which reduces the chances of errors
* Performance – as templates are resolved at compile time there are no runtime overheads associated with dynamic type checking or type casting
* Content interface – there’s a uniformed interface regardless of the data type used
* Simplified code – the use of templates decreases the need for repetitive boilerplate code which results in comprehensive implementations

**Implementation**

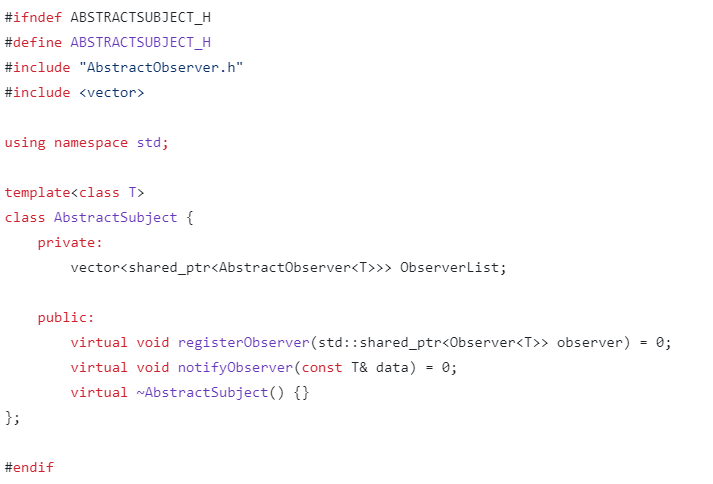
* Provide code for observer, subject and concrete implementations
* Describe the functionality of each component

Abstract Observer defines the interface or abstract base class that a concrete observer class should implement. This class declares the methods that will be called by the subject to notify the observer of any changes



* Template parameter – T represents the type of data that will be passed to the observer once the update has been completed
* Method – virtual update method is implemented by concrete observer classes as it defines how the observer reacts to updates
* Destructor – a virtual destructor makes sure that derived classes are updated accordingly once an observer object is deleted or removed

Abstract Subject is the base class that defines a unified interface and behaviour for concrete subjects, this helps various types of subjects to inherit from the abstract subject which helps with consistency and reusability.



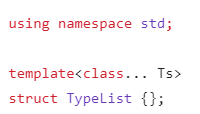
Template parameter – ‘T’ represents the type of data that will be passed to the observers during notification

Smart pointer – std::shared\_ptr is used to retain shared ownership of an object through a pointer as it manages observers

Method – notifyObserver method calls all registered observers and passes them the notification, while registerObserver method registers all

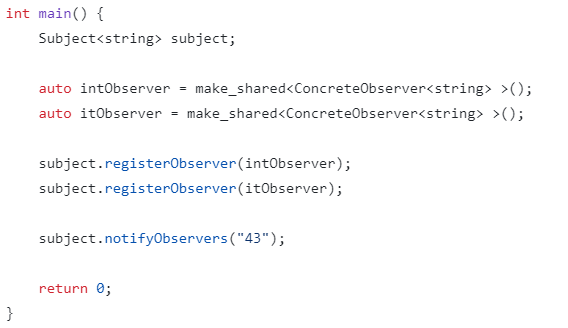
Destructor – virtual destructor AbstractSubject is used to ensure that the destuctor of the derived class is called when an object is deleted through a pointer to the base class

Typelist represents a list of types at compile time and is usually used to create flexible and reusable code that can operate different types



The typelist structure is defined, the parameter ‘Ts’ is used to represent the typelist for the template

Main function is used to refer to the point of entry for the program. This method is used to execute the program.



**Scenario 1: Stock Market Notification**

* Example code showing how to use the library
* Explain how the subject interacts with the observer

**Scenario 2: Retail Stock Notification**

* Example code showing how to use the library
* Explain how the subject interacts with the observer

**Conclusion**