**ЗМІСТ**

[**1.** **ВСТУП** 2](#_Toc515908594)

[**2. КОРОТКІ ВІДОМОСТІ ПРО С++** 2](#_Toc515908595)

[**2.1 ВІДМІННОСТІ СІ ТА С++** 4](#_Toc515908596)

[**3 КОНЦЕПЦІЯ ООП** 6](#_Toc515908597)

[**3.1 ІНКАПСУЛЯЦІЯ** 8](#_Toc515908598)

[**3.2 ПОЛІМОРФІЗМ** 9](#_Toc515908599)

[**3.3 НАСЛІДУВАННЯ** 9](#_Toc515908600)

[**4. ПОНЯТТЯ БАЗИ ДАНИХ** 10](#_Toc515908601)

[**4.1 ПОНЯТТЯ СУБД** 11](#_Toc515908602)

[**4.2 ОСНОВНІ ФУНКЦІЇ СУБД** 16](#_Toc515908603)

[**5. РОБОЧЕ СЕРЕДОВИЩЕ** 18](#_Toc515908604)

[**6. ПРАКТИЧНА ЧАСТИНА** 19](#_Toc515908605)

[**6.1 КОМАНДНИЙ ІНТЕРФЕЙС** 19](#_Toc515908606)

[**6.2. СТРУКТУРА ОРГАНІЗАЦІЇ ДАНИХ У КЛАСАХ** 20](#_Toc515908607)

[**6.3. РЕАЛІЗАЦІЯ TABLE ТА ПОВ’ЯЗАНИХ КЛАСІВ** 23](#_Toc515908608)

[**6.4. РЕАЛІЗАЦІЯ КЛАСУ VECTOR НА ЗАМІНУ STD::VECTOR** 25](#_Toc515908609)

[**6.5. EXPRESSION BUILDER** 25](#_Toc515908610)

[**6.6. ПРИКЛАД РОБОЧОГО ПРОЦЕСУ** 28](#_Toc515908611)

[**7. ВИСНОВОК** 31](#_Toc515908612)

[**ДОДАТКИ** 32](#_Toc515908613)

[**ДОДАТОК А: ЛІСТИНГ ПРОГРАМИ** 32](#_Toc515908614)

1. **ВСТУП**

Об'єктно-орієнтоване програмування (ООП) - парадигма програмування, в якій основними концепціями є поняття об'єктів і класів (в менш відомому варіанті - прототипів).

Клас - це тип, що описує як влаштовані об'єкти. Поняття «клас» має на увазі деяку поведінку і спосіб подання. Поняття «об'єкт» має на увазі щось, що володіє певною поведінкою і способом подання. Кажуть, що об'єкт - це екземпляр класу. Клас можна порівняти з кресленням, згідно з яким створюються об'єкти. Зазвичай класи розробляють таким чином, щоб їх об'єкти відповідали об'єктам предметної області.

Клас є описуваною мовою термінології вихідного коду моделлю, ще не існуючої сутності, так званого об'єкту.

Об'єкт - сутність в адресному просторі обчислювальної системи, що з'являється при створенні екземпляра класу (наприклад, після запуску результатів компіляції вихідного коду на виконання).

Прототип - це об'єкт-зразок, за образом якого створюються інші об'єкти.

З точки зору мови програмування клас об'єктів можна розглядати як тип даних, а окремий об'єкт - як дані цього типу.

Визначення програмістом власних класів об'єктів для конкретного набору задач повинно дозволити описувати окремі завдання в термінах самого класу завдань (при відповідному виборі імен типів та імен об'єктів, їх параметрів і виконуваних дій).

    Таким чином, об'єктно-орієнтований підхід передбачає, що при розробці програми повинні бути визначені класи використованих у програмі об'єктів і побудовано їх описи, а потім створені екземпляри необхідних об'єктів і визначено взаємодію між ними.

**2. КОРОТКІ ВІДОМОСТІ ПРО С++**

C++ (Сі-плюс-плюс) — мова програмування високого рівня з підтримкою декількох парадигм програмування: об'єктно-орієнтованої, узагальненої та процедурної. Розроблена Б'ярном Страуструпом (англ. Bjarne Stroustrup) в AT&T Bell Laboratories (Мюррей-Хілл, Нью-Джерсі) у 1979. Базується на мові С.

У 1990-х роках С++ стала однією з найуживаніших мов програмування загального призначення. Мову використовують для системного програмування, розробки програмного забезпечення, написання драйверів, потужних серверних та клієнтських програм, а також для розробки розважальних програм таких як відеоігри. С++ суттєво вплинула на інші, популярні сьогодні, мови програмування: С# та Java.

При створенні С++ прагнули зберегти сумісність з мовою С. Більшість програм на С справно працюватимуть і з компілятором С++.

С++ має наступні нововведення порівняно з С:

-підтримка об'єктно-орієнтованого програмування через класи;

-підтримка узагальненого програмування через шаблони;  
-доповнення до стандартної бібліотеки;  
-додаткові типи даних;  
-обробка винятків;  
-простори імен;  
-вбудовані функції;  
-перевантаження операторів;  
-перевантаження імен функцій;  
- посилання і оператори управління вільно розподіленою пам'яттю.

 Назва «С++» була вигадана Ріком Масситті (Rick Mascitti) і вперше було використана в грудні 1983 року. Раніше, на етапі розробки, нова мова називалася «Сі з класами». Ім'я, що вийшло у результаті, походить від оператора Сі «++» (збільшення значення змінної на одиницю) і поширеному способу присвоєння нових імен комп'ютерним програмам, що полягає в додаванні до імені символу «+» для позначення поліпшень. Згідно зі Страуструпом, «ця назва вказує на еволюційну природу змін Ci».

 C++ працює під керуванням багатьох операційних систем. До них входять: Mac, Linux (включаючи дистрибутив Ubuntu усіх версій), Windows.

Особливості роботи у Linux. Щоб писати програми у Linux на НЕ потрібне жодне додаткове ПЗ, достатньо блокнота (але обов’язково вказувати формат документа .cpp). наступник кроком, щоб зкомпілювати програму і запустити нам буде потрібний тільки термінал (програма запуститься у терміналі також).

Додам ще тільки, що я переважно буду працювати під OC Windows 10 Pro. і використовувати Dev C++.

При розробці мови С ++ одним з найважливіших критеріїв вибору була простота. Коли виникало питання, що спростити: посібник з мови та іншу документацію або транслятор, - то вибір робили на користь першого. Величезне значення надавалося сумісності з мовою С, що завадило видалити його синтаксис.

**2.1 ВІДМІННОСТІ СІ ТА С++**

    В С ++ немає типів даних і елементарних операцій високого рівня. Наприклад, не існує типу матриця з операцією звернення або типу рядок з операцією конкатенації. Якщо користувачеві знадобляться подібні типи, він може визначити їх в самій мові. Програмування на С ++ по суті зводиться до визначення універсальних або залежних від галузі застосування типів. Добре продуманий користувальницький тип відрізняється від вбудованого типу лише способом визначення, але не способом застосування.

    З мови виключалися можливості, які можуть привести до накладних витрат пам'яті або часу виконання, навіть якщо вони безпосередньо не використовуються в програмі. Наприклад, було відкинуто пропозицію зберігати в кожному об'єкті деяку службову інформацію. Якщо користувач описав структуру, яка містить дві величини, що займають по 16 розрядів, то гарантується, що вона поміститься в 32-х розрядний регістр.

    Мова С ++ проектувалася для використання в досить традиційному середовищі, а саме: в системі програмування С операційної системи UNIX. Але є цілком обґрунтовані аргументи на користь використання С++ програмному середовищі. Такі можливості, як динамічне завантаження, розвинені системи трансляції та бази даних для зберігання визначень типів, можна успішно використовувати без шкоди для мови.

    Типи С ++ розраховані на певний синтаксичний аналіз, проведений транслятором для виявлення випадкового псування даних. Вони не забезпечують секретності даних і захисту від навмисного порушення правил доступу до них. Однак, ці кошти можна вільно використовувати, не хвилюючись витрат пам'яті і часу виконання програми. Враховано, що конструкція мови активно використовується тоді, коли вона не тільки витончено записується на ньому, а й цілком доступна звичайним програм.

Мова С ++ проектувалася як "кращий С", що підтримує абстракцію даних і об'єктно-орієнтоване програмування. При цьому вона повинна бути придатною для більшості основних завдань системного програмування.

    Основні труднощі для мови, які створювалися в розрахунку на методи приховування даних, абстракції даних і об'єктно-орієнтованого програмування, в тому, що для того, щоб бути мовою загального призначення, вона повинна:

    - працювати на “традиційних” машинах;

    - співіснувати з “традиційними” операційними системами і мовами;

    - змагатися з “традиційними” мовами програмування в ефективності

виконання програми;

    - бути придатним у всіх основних областях використання.

    Це означає, що повинні бути можливості для ефективних числових операцій (арифметика з плаваючою точкою без особливих накладних витрат, інакше користувач вважатиме за краще Фортран) і засоби такого доступу до пам'яті, який дозволить писати цією мовою драйвери пристроїв. Крім того, треба вміти писати виклики функцій в досить незвичні записи, прийняті для звернень в традиційних операційних системах. Нарешті, повинна бути можливість з мови, що підтримує об'єктно-орієнтоване програмування, викликати функції, написані на інших мовах, а з інших мов викликати функцію на цій мові, що підтримує об'єктно-орієнтоване програмування.

    Далі, не можна розраховувати на широке використання шуканого мови програмування як мови загального призначення, якщо реалізація його цілком покладається на можливості, які відсутні в машинах з традиційною архітектурою.

    Якщо не вводити в мову можливості низького рівня, то доведеться для основних завдань більшості областей додатки використовувати деякі мови низького рівня, наприклад С або асемблер. Але С ++ проектувався з розрахунком, що в ньому можна зробити все, що допустимо на С, причому без збільшення часу виконання. Взагалі, С ++ проектувався, виходячи з принципу, що не повинно виникати ніяких додаткових витрат часу і пам'яті, якщо тільки цього явно не побажає сам програміст.

    Мова проектувалася в розрахунку на сучасні методи трансляції, які забезпечують перевірку узгодженості програми, її ефективність і компактність подання. Основним засобом боротьби зі складністю програм бачиться, перш за все, строгий контроль типів і інкапсуляція. Особливо це стосується великих програм, що створюються багатьма людьми. Користувач може не бути одним із творців таких програм, і може взагалі не бути програмістом. Оскільки ніяку справжню програму можна написати без підтримки бібліотек, що створюються іншими програмістами, останнє зауваження можна віднести практично до всіх програм.

    С ++ проектувався для підтримки принципу, що будь-яка програма є модель деяких існуючих в реальності понять, а клас є конкретним поданням поняття, взятого з області. Тому класи пронизують всю програму на С ++, і накладаються жорсткі вимоги на гнучкість поняття класу, компактність об'єктів класу і ефективність їх використання. Якщо працювати з класами буде незручно або занадто накладно, то вони просто не будуть використовуватися, і програми виродяться в програми на "кращому С". Значить користувач не зможе насолодитися тими можливостями, заради яких, власне, і створювався мову.

**3 КОНЦЕПЦІЯ ООП**

    Класи об'єктів часто зручно будувати так, щоб вони утворювали ієрархічну структуру. Наприклад, клас «Студент», що описує абстрактного студента, може служити основою для побудови класів «Студент 1 курсу», «Студент 2 курсу» і т.д., які мають всі властивості студента в загальному, а також і деякими додатковими властивостями, що характеризують студента конкретного курсу. При розробці інтерфейсу з користувачем програми можуть використовувати об'єкти загального класу «Вікно» і об'єкти класів спеціальних вікон, наприклад, вікон інформаційних повідомлень, вікон введення даних, тощо. У таких ієрархічних структурах один клас може розглядатися як базовий для інших, похідних від нього класів. Об'єкт похідного класу має всі властивості базового класу і деякі власні властивості, він може реагувати на ті ж типи повідомлень від інших об'єктів, що і об'єкт базового класу, а також і на повідомлення, що мають сенс тільки для похідного класу. Зазвичай кажуть, що об'єкт похідного класу успадковує всі властивості свого базового класу.

    Деякі параметри об'єкта можуть бути локалізовані всередині об'єкта і недоступні для прямого впливу ззовні об'єкту. Наприклад, під час руху об'єкта-автомобіля об'єкт-водій може впливати тільки на обмежений набір органів управління (рульове колесо, педалі газу, зчеплення і гальма, важіль перемикання передач) і йому недоступний цілий ряд параметрів, що характеризують стан двигуна і автомобіля в цілому.

Концепція об’єктно-орієнтованого програмування має на увазі, що основою управління процесом реалізації програми є передача повідомлень об'єктам. Тому об'єкти повинні визначатися спільно з повідомленнями, на які вони повинні реагувати при виконанні програми. У цьому полягає головна відмінність ООП від процедурного програмування, де певні структури даних окремо передаються в процедури (функції) в якості параметрів. Таким чином, об'єктно-орієнтована програма складається з об'єктів - окремих фрагментів коду, що обробляє дані, які взаємодіють один з одним через певні інтерфейси.

Об'єктно-орієнтована мова програмування повинна мати наступні властивості:

* абстракції - формальне про якості або властивості предмета шляхом уявного видалення деяких деталей або матеріальних об'єктів;
* інкапсуляції - механізму, що зв'язує разом код і дані, якими він маніпулює, і захищає їх від зовнішніх перешкод і некоректного використання;
* успадкування процес, за допомогою якого один об'єкт набуває властивостей іншого, тобто підтримується ієрархічної класифікації;
* поліморфізму - властивості, що дозволяє використовувати один і той же інтерфейс для загального класу дій.

Розробка об'єктно-орієнтовані програми складаються з таких послідовних робіт:

- визначення основних об'єктів, необхідних для вирішення завдання;

- визначення закритих даних (даних стану) для вибраних об'єктів;

- визначення другорядних об'єктів і їх закритих даних;

- визначення ієрархічної системи класів, що представляють вибрані об'єкти;

- визначення ключових повідомлень, які повинні обробляти об'єкти кожного класу;

- розробка послідовності виразів, які дозволяють вирішити поставлену задачу;

- розробка методів, що обробляють кожне повідомлення;

- очищення проекту, тобто усунення всіх допоміжних проміжних матеріалів, що використовувалися при проектуванні;

- кодування, налагодження, компонування і тестування.

Об'єктно-орієнтоване програмування дозволяє програмісту моделювати об'єкти певній предметної області шляхом програмування їх змісту та поведінки в межах класу. Конструкція «клас» забезпечує механізм інкапсуляції для реалізації абстрактних типів даних. Інкапсуляція як би приховує і подробиці внутрішньої реалізації типів, і зовнішні операції і функції, допустимі для виконання над об'єктами цього типу. Ми вже звикли використовувати в своїх програмах процедури і функції для програмування складних дій з обробки даних, які доводиться виконувати багато разів. Використання підпрограм свого часу було важливим кроком на шляху до підвищення ефективності програмування. Підпрограма може мати формальні предмети, які при зверненні до неї замінюються фактичними предметами. В цьому випадку є небезпека виклику підпрограми з неправильними даними, що може привести до збою програми і її аварійного завершення при виконанні. Тому природне узагальнення традиційного підходу до програмування є об'єднанням даних і підпрограм (процедури і функції), призначена для їх обробки.

**3.1 ІНКАПСУЛЯЦІЯ**

Інкапсуляція є найважливішою властивістю об'єктів, на якому будується об'єктно-орієнтоване програмування. Інкапсуляція полягає в тому, що об'єкт приховує в собі деталі, які несуттєві для використання об'єкта. У традиційному підході до програмування з використанням глобальних змінних програміст не був застрахований від помилок, пов'язаних з використанням процедур, які не призначені для обробки даних, пов'язаних з цими змінними. Припустимо, наприклад, що є "не-ООП» програма, призначена для нарахування заробітної плати співробітникам якоїсь організації, а в програмі є два масиви. Один масив зберігає величину заробітної плати, а інший - телефонні номери співробітників (для складання звіту для податкової інспекції). Що станеться, якщо програміст випадково переплутає ці масиви? Очевидно, у відділі бухгалтерії виникне деяке непорозуміння. Жорстке пов’язування даних та процедур їх обробки в одному об'єкті дозволить уникнути неприємностей такого роду. Інкапсуляція і є засобом організації доступу до даних тільки через відповідні методи.

Якщо в програмі є опис декількох змінних зазначеного типу, то для кожної змінної резервується своя власна область пам'яті для зберігання даних, а покажчики на точки входу в процедуру і функції - загальні. Виклик кожен метод можливо тільки за допомогою складного імені, що зрозуміло показує, для обробки яких даних призначений даний метод.

**3.2 ПОЛІМОРФІЗМ**

Поліморфізм - це властивість, яка дозволяє одне і те ж ім'я використовувати для вирішення двох або більше схожих, але технічно різних завдань. Метою поліморфізму, стосовно об'єктно-орієнтованого програмування, є використання одного імені для задання загальних для класу дій. Виконання кожної конкретної дії буде визначатися типом даних. Наприклад для мови Сі, в якій поліморфізм підтримується недостатньо, знаходження абсолютної величини числа вимагає трьох різних функцій: abs (), labs () і fabs (). Ці функції підраховують і повертають абсолютну величину цілих, довгих цілих і чисел з плаваючою точкою відповідно. В С++ кожна з цих функцій може бути названа abs (). Тип даних, який використовується при виконанні функції, визначає, яка конкретна версія функції дійсно виконується. В С ++ можна використовувати одне ім'я функції для безлічі різних дій. Це називається перевантаженням функцій (function overloading).

У більш загальному сенсі, концепцією поліморфізму є ідея "один інтерфейс, безліч методів". Це означає, що можна створити загальний інтерфейс для групи близьких за змістом дій. Перевагою поліморфізму є те, що він дозволяє використання того ж інтерфейсу для задання єдиного класу дій. Вибір же конкретного дії, в залежності від ситуації, покладається на компілятор. Вам, як програмісту, не потрібно робити цей вибір самому. Потрібно тільки пам'ятати і використовувати загальний інтерфейс. Приклад з попереднього абзацу показує, як, маючи три імені для функції визначення абсолютної величини числа замість одного, звичайне завдання стає більш складною, ніж це дійсно необхідно.

**3.3 НАСЛІДУВАННЯ**

Наслідування класів - дуже потужна можливість в об'єктно орієнтованому програмуванні. Воно дозволяє створювати похідні класи (класи спадкоємці), взявши за основу все методи і елементи базового класу (класу батька). Таким чином економиться маса часу на написання і налагодження коду нової програми. Об'єкти похідного класу вільно можуть використовувати все, що створено і налагоджено в базовому класі. При цьому, ми можемо в похідний клас, дописати необхідний код для удосконалення програми: додати нові елементи, методи і т.д .. Базовий клас залишиться недоторканим.

Наслідування дозволяє уникнути дублювання зайвого коду при написанні класів. Нехай в базі даних ВНЗ повинна зберігатися інформація про всіх студентів та викладачів. Представляти всі дані в одному класі не вийде, оскільки для викладачів нам знадобиться зберігати дані, які для студента не застосовні, і навпаки.

# **4. ПОНЯТТЯ БАЗИ ДАНИХ**

База даних (скорочено — БД) — впорядкований набір логічно взаємопов'язаних даних, що використовуються спільно та призначені для задоволення інформаційних потреб користувачів. У технічному розумінні включно й система керування БД.

Головне завдання БД — гарантоване збереження значних обсягів інформації (так звані записи даних) та надання доступу до неї користувачеві або ж прикладній програмі. Таким чином, БД складається з двох частин: збереженої інформації та системи керування нею.

В БД повинні зберігатися дані, логічно пов'язані між собою. Для того щоб дані можна було пов'язати між собою, і зв'язати так, щоб ці зв'язки відповідали реально існуючим в даній предметній області, останню піддають детальному аналізу, виділяючи сутності чи об'єкти.

Сутність або об'єкт - це те, про що необхідно зберігати інформацію. Сутності мають деякі характеристики, звані атрибутами, які теж необхідно зберігати в БД. Визначивши сутності і їхні атрибути, необхідно перейти до виявлення зв'язків, які можуть існувати між деякими сутностями. Зв'язок - це те, що об'єднує дві або більше сутностей.

Зв'язки між сутностями також є частиною даних, і вони також повинні зберігатися в базі даних. Проектована база даних повинна володіти певними властивостями:

- Цілісність бази даних досягається внаслідок введення обмежень, пов'язані з нормалізацією БД. У кожен момент часу існування БД відомості, що містяться в ній, повинні бути несуперечливі.

- Відновлювання - можливість відновлення БД після збою системи або окремих видів поломок системи.

- Безпека - припускає захист даних від навмисного і ненавмисного доступу, модифікації або руйнування.

- Ефективність. Властивість ефективності зазвичай оцінюється двома параметрами: мінімальний час реакції на запит користувача і мінімальні потреби в пам'яті, а так же поєднанням цих параметрів. У свою чергу, система управління базами даних, за допомогою якої користувачі можуть визначати, створювати і підтримувати базу даних, а також здійснювати до неї контрольований доступ, ділиться на два класи - система загального призначення та спеціалізована система.

## **4.1 ПОНЯТТЯ СУБД**

СУБД загального призначення володіє засобами налаштування на роботу з конкретною БД в умовах конкретного застосування. У деяких ситуаціях СУБД загального призначення не дозволяють добитися необхідних проектних та експлуатаційних характеристик (продуктивність, зайнятий обсяг пам'яті та інше), але створення спеціалізованих СУБД досить трудомісткий процес і для того, щоб його реалізувати, потрібні дуже вагомі підстави.

У процесі реалізації своїх функцій СУБД постійно взаємодіє з базою даних і з іншими прикладними програмними продуктами користувача, призначеними для роботи з даною БД і званими додатками. Для того щоб СУБД успішно справлялася зі своїми завданнями, вона повинна володіти певними можливостями:

1. СУБД включає в себе мову визначення даних, за допомогою якого можна визначити базу даних, її структуру, типи даних, а також засоби завдання обмежень для збереженої інформації.

2. СУБД дозволяє вставляти, видаляти, оновлювати і витягувати інформацію з бази даних за допомогою мови управління даними.

3. Більшість СУБД можуть працювати на комп'ютерах з різною архітектурою і під різними операційними системами, причому на роботу користувача тип платформи впливу не робить.

4. Стратегії СУБД мають достатньо розвинені засоби адміністрування БД.

5. СУБД надає контрольований доступ до бази даних за допомогою: системи забезпечення безпеки; системи підтримки цілісності бази даних; системи управління паралельною роботою додатків, контролюючої процеси їх спільного доступу до бази даних; системи відновлення, що дозволяє відновити базу даних до попереднього несуперечливого стану.

До головних функцій системи управління базами даних (СКБД) відносять такі:

Управління даними в зовнішній пам'яті - надання користувачам можливості виконання самих основних операцій, які здійснюються з даними - це збереження, вилучення та оновлення інформації, а так само забезпечення необхідних структур зовнішньої пам'яті як для зберігання даних, так і для прискорення доступу до них.

Управління транзакціями - управління послідовністю операцій над БД, виконуваних з метою доступу або зміни вмісту бази даних.

Відновлення бази даних - підтримку надійності зберігання даних в БД для можливості відновлення останнього узгодженого стану БД після апаратного або програмного збою.

Підтримка мов бази даних - використання спеціальних мов, що містять всі необхідні засоби для роботи з БД.

Управління паралельним доступом - управління одночасним доступом користувачів до оброблюваних даних. Коли два або більше користувачів одночасно отримують доступ до бази даних, конфлікт з небажаними наслідками легко може виникнути, наприклад, якщо хоча б одні з них спробує оновити дані. СУБД повинна гарантувати, що при одночасному доступі до бази даних багатьох користувачів подібних конфліктів не відбудеться.

Управління буферами оперативної пам'яті - підтримка власного набору буферів оперативної пам'яті з дисципліною заміни буферів для збільшення швидкості обігу до елементу БД.

Контроль доступу до даних - механізм, що гарантує можливість доступу до бази даних тільки санкціонованих користувачів і захищає її про будь-якого несанкціонованого доступу. У сучасних СУБД підтримується одні з двох широко поширених підходів до питання забезпечення безпеки даних: вибірковий підхід або обов'язковий підхід.

У більшості сучасних систем передбачається вибірковий підхід, при якому якийсь користувач володіє різними правами при роботі з різними об'єктами. Значно рідше застосовується обов'язковий, де кожному об'єкту даних присвоюється певний класифікаційний рівень, а кожен користувач має деяким рівнем допуску.

Підтримка цілісності даних - відображає коректність і несуперечність збережених у БД даних. СУБД повинна містити відомості по тих правилах, які не можна порушувати при роботі з даними, і володіти інструментами контролю за тим, щоб дані та їх зміни відповідали заданим правилам.

Види:

* оперативна БД
* ретроспективна БД

З метою забезпечення ефективності доступу записи даних організовують як множину фактів (елемент даних).

Часто зустрічається характеристика БД на основі певних параметрів або необхідних вимог, наприклад:

* значна кількість даних;
* незалежність даних;
* відкритий доступ до даних;
* підтримка транзакцій з гарантією відповідних властивостей;
* гарантована відсутність збоїв;
* одночасна робота з багатьма користувачами.

З подальшим розвитком БД змінюються й ці вимоги та додаються нові, тому одностайності щодо повноти цієї характеристики немає.

Структуровані БД використовують структури даних, тобто структурований опис типу фактів за допомогою схеми даних, більш відомої як модель даних. Модель даних описує об'єкти та взаємовідношення між ними. Існує декілька моделей (чи типів) баз даних, основні: плоска, ієрархічна, мережна та реляційна. Приблизно з 2000 року більше половини БД використовують реляційну модель.

До неструктурованих БД належать повнотекстові бази даних, які містять неструктуровані тексти статей чи книг у формі, що дозволяє здійснювати швидкий пошук.

Створення баз даних є досить поширеним і в той же час важливим завданням.

Для користувача ті об’єкти, якими він оперує під час роботи з системою, і є даними, але зі сторони програмування зрозуміло, що всі дані зберігаються в файлах.

Файли – це ті структури зберігання даних, якими маніпулює операційна система. Але файли – це всього лише названа послідовність байтів, що зберігається в пам’яті. При створенні інформаційної системи не дуже зручно оперувати послідовностями байтів. Безпосередньо з файлами працюють системні програмісти, а прикладні програмісти працюють з іншою абстракцією. Ця абстракція знаходиться на середньому рівні між рівнем файлової системи (фізичний рівень) та рівнем користувача (зовнішній рівень). Цей проміжний рівень можна назвати логічним або прикладним. Це можна побачити на рисунку нижче:

|  |
| --- |
| Зовнішній рівень |
| Логічний рівень |
| Внутрішній рівень |

Рівень користувача

Рівень прикладних програмістів та адміністраторів

Рівень файлової системи та системних програмістів

СУБД – це система управління базами даних. Для того щоб зрозуміти що таке СУБД потрібно звернутися до малюнку вище. Для того, щоб прикладні програмісти сприймали базу даних не у вигляді набору файлів, а як деяку структуру, яка описує предметну область, між ними і файловою системою повинен бути деякий прошарок. Саме він і являє собою набір процедур (програмний інтерфейс), за допомогою якого прикладний програміст може керувати базою даних. Цю функцію і виконує СУБД. Іншими словами, СУБД відділяє прикладного програміста від фізичної структури бази даних. У найпростішому випадку СУБД складається тільки з програмного інтерфейсу. Так відбувається в деяких системах програмування. В більш складних системах, ми маємо також інтегроване середовище, що позволяє в інтерактивному режимі керувати базами даних. Крім цього, СУБД також надає розробникам засоби безпеки, такі як: утиліти для резервного копіювання, підтримки паралельної роботи додатків, систем підтримки цілісності баз даних. Більшість сучасних інформаційних систем будуються саме на основі СУБД.

На малюнку нижче показана схема взаємодії СУБД та прикладного програмного забезпечення в схемі побудови інформаційної системи. Тут важливо зрозуміти, що база даних в певній моделі існує для прикладного програмного забезпечення, тоді як програмне забезпечення СУБД взаємодіє з даними на рівні файлової системи та системних викликів операційної системи.

Прикладне програмне забезпечення

СУБД

Дані

Використання СУБД при побудові інформаційних систем повинно реалізувати фізичну і логічну незалежність прикладного програмування від даних. Фізична незалежність від даних полягає в тому, що робота програмного забезпечення інформаційної системи не буде залежати від змін, що стаються на внутрішньому, фізичному рівні. Ці зміни можуть полягати, наприклад, в тому, що буде змінена файлова система, або ж в тому, що зміниться структура тих файлів, що складають базу даних.

Логічна незалежність прикладного програмування від даних при використанні СУБД в трирівневій структурі доступу до даних полягає, насамперед в тому, що додавання нових елементів(наприклад додавання нового рядка або стовпця в таблицю) в структуру даних ніяк не впливає на функціонування програмного забезпечення. Добитися логічної незалежності від операцій видалення елементів із бази даних загалом нереально, оскільки на об’єкти, що видаляють, як правило, існують посилання в програмному забезпеченні.

В якості основних етапів узагальненої технології роботи з БДможна виділити наступні:

•    створення структури бази даних;

•    введення і редагування даних в таблицях;

•    обробка даних, що містяться в таблицях;

•    виведення інформації з БД.

Практично всі використовувані БД зберігають дані наступних типів: текстовий (символьний), числовий, календарний, логічний. Якщо БД, яка обробляється включає декілька взаємозв'язаних таблиць, то необхідне визначення ключового поля в кожній таблиці, а також полів, за допомогою яких буде організований зв'язок між таблицями.

Створення структури таблиці не пов'язане із заповненням таблиць даними, тому ці дві операції можна рознести в часі. Заповнення таблиць даними можливо як безпосередньо введенням даних, так і у результаті виконання програм і запитів. Практично всі СУБД дозволяють вводити і коректувати дані в таблицях двома способами:

•    за допомогою стандартної форми у вигляді таблиці;

•    за допомогою екранних форм, що спеціально створюються користувачем.

При роботі з будь-якою базою даних розрізняють проектувальний і експлуатаційний режими. При проектуванні бази даних основним етапом є вибір виду представлення даних в базі даних (вибір моделі представлення бази даних) і введення даних в базу даних.

У експлуатаційному режимі користувач за допомогою спеціальних програм дістає можливість обробки даних, що містяться в базі.

## **4.2 ОСНОВНІ ФУНКЦІЇ СУБД**

Основні функції СУБД можна проілюструвати наступним чином:

Основні функції СУБД

Визначення структури даних

Визначення схеми бази даних

Визначення обмежень цілісності

Маніпулювання даними

Додавання даних

Оновлення даних

Пошук і вибирання даних

Видалення даних

Збереження даних

Захист даних

Забезпечення дотримання обмежень цілісності

* Визначення даних та відображення їх певним, формальним способом. Реалізація цих визначень у вигляді відповідних об'єктів.
* Маніпулювання даних (СУБД сприймає, інтерпретує і обробляє запити користувачів на вибирання, оновлення, і видалення неявних данних, або на додавання

Існують такі види баз даних:

* Ієрархічні
* Реляційні
* Мережеві
* Об’єктно-орієнтовані
* Об’єктно-реляційні

За ступенем розподіленості вони поділяються на:

* Локальні СУБД (коли всі частини СУБД розміщуються на одному комп’ютері)
* Розподілені СУБД (частини СУБД можуть розміщатися на двох і більше комп’ютерах)

Вбудована СУБД – СУБД, яка може поставлятися як складова частина деякого програмного продукту, не вимагаючи процедури самостійної установки. Вбудована СУБД призначена для локального зберігання даних свого додатку і не розрахована на колективне використання в мережі. Фізично вбудована СУБД найчастіше реалізована у вигляді бібліотеки, що підключається. Доступ до даних з боку додатка може відбуватися через спеціальні програмні інтерфейси, тощо.

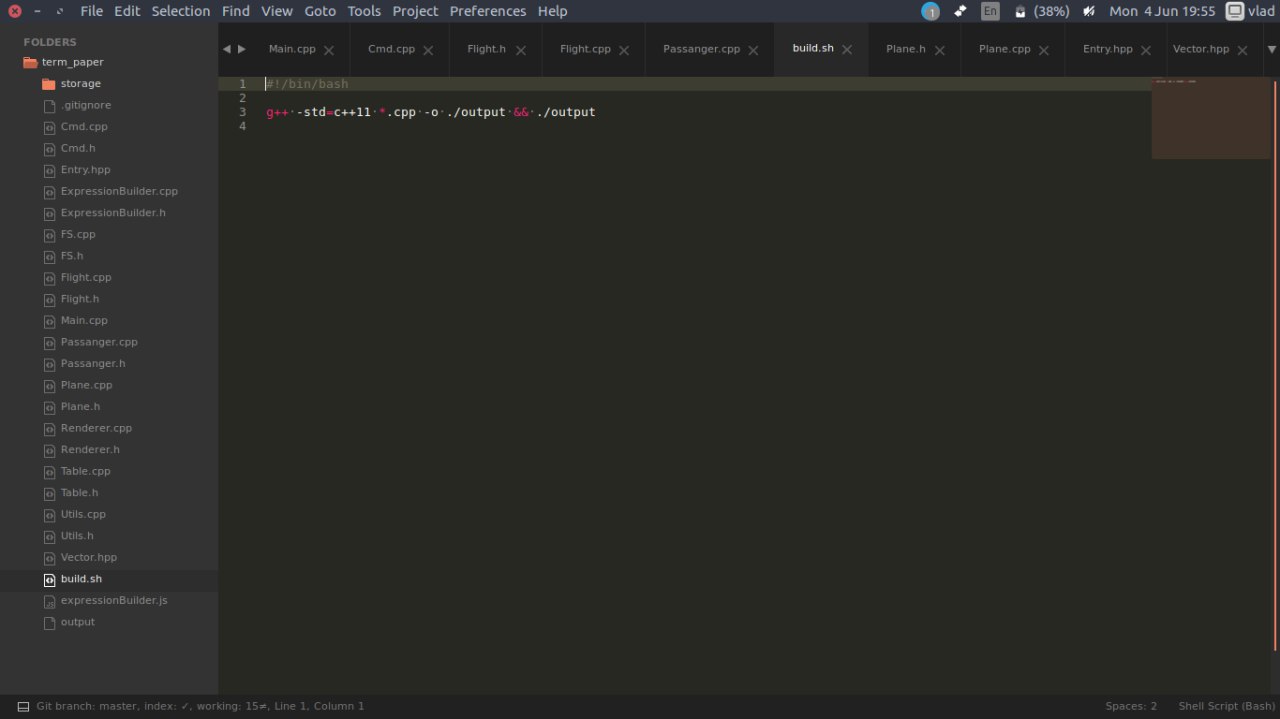
Основною перевагою використання СУБД є те, що з нею взаємодіють не лише люди, а й інші прикладні програми. Вона встановлює єдині правила як для людей, так і для програм, щоб створити своєрідну оболонку навколо бази даних. Такий підхід означає, що дані в БД більш захищені, ніж у документах інших типів, до яких прикладні програми можуть звертатися безпосередньо. Будь-які введені дані, що не відповідають схемі бази або деяким іншим умовам, будуть заблоковані СУБД. Це називають цілісністю даних.

Під цим поняттям розуміють несуперечність даних, що зберігаються в базі. Для уникнення суперечливих ситуацій при модифікації баз даних, співвідношення між даними контролюються спеціальними засобами цілісності БД. Умови, що накладаються на дані та яких дотримуються за будь-яких оновлень, розробляються спеціальною службою-адміністраторами баз даних, а СУБД надають необхідні інструментальні заходи.

Адміністратор бази даних (АБД) – особа, що відповідає за вироблення вимог до бази даних, її проектування, реалізацію, ефективне використання та супровід, включаючи управління обліковими записами користувачів БД і захист від несанкціонованого доступу. Не менш важливою функцією адміністратора БД є підтримка цілісності бази даних.

# **5. РОБОЧЕ СЕРЕДОВИЩЕ**

Для створення даного програмного забезпечення я обрав текстовий редактор Sublime Text 3 та компілятор g++. Вибір компілятора обумовлений тим, що g++ є стандартним компілятором C++ для linux-систем, і включає в себе велику кількість корисних флагів компіляції для відлогодження програм. Текстовий редактор Sublime Text 3 пропонує гнучкість в налаштуванні та зручний інтерфейс, тому я обрав саме його. Одним з недоліків даної конфігурації є те, що сукупність редактора тексту і компілятора, що працюють окремо не становить собою повноцінну IDE, тому примітивні синтаксичні та лексичні помилки (відсутність ‘;’, неправильна типізація) неможливо було виправити до початку компіляції, яка могла займати деякий час (зазвичай до 30с). Запуск компілятора відбувався з bash-скрипта у директорії проекту, що робить процес компіляції та запуску програми значно зручнішим.



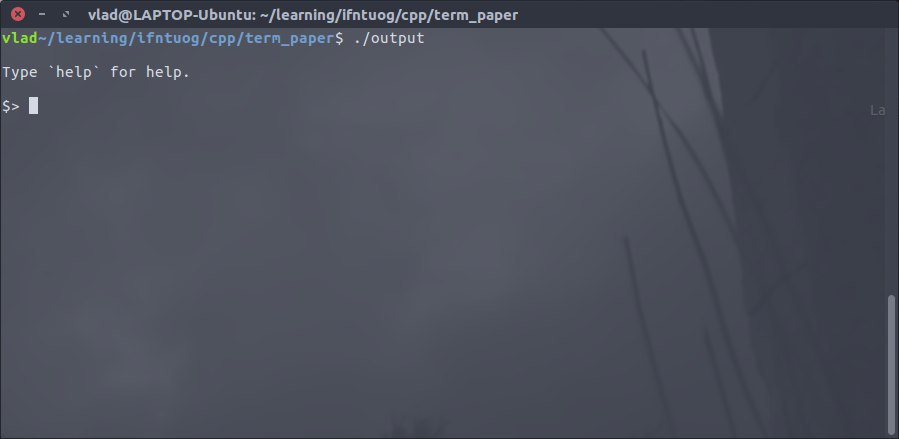
*Рисунок 5.1. Робоче середовище (на прикладі файлу build.sh)*

# **6. ПРАКТИЧНА ЧАСТИНА**

Завданням даної курсової роботи є розробка консольної системи управління базою даних аеропорту. Управління БД здійснюється шляхом введення семантичних команд (‘add’, ‘update’, ‘remove’…) та їх параметрів. Фізичне представлення БД на комп’ютері користувача – список \*.txt файлів, кожен з яких відповідає відповідній таблиці БД. Першим рядком кожного файла є список заголовків таблиці, відділених табуляцією. Кожен наступний рядок до кінця файлу – дані, збережені у відповідній таблиці. Записи у межах 1 кортежу також відділені табуляцією. Вмістом будь-якого поля можуть бути будь-які символи таблиці ASCII, крім керуючих кодів та символу табуляції. При введенні забороненого символу, цей символ замінюється пробілом. Також при записі відбувається додаткова нормалізація користувацьких даних, а саме процедура видалення зайвих пробілів (усіх з початку та кінця запису та видалення усіх зайвих, якщо їх є більше ніж 1 поспіль).

## **6.1 КОМАНДНИЙ ІНТЕРФЕЙС**

Для введення даних користувачем передбачено спеціальний командний інтерфейс (CLI), який за принципом роботи є схожим на термінал linux-систем.



*Рисунок 6.1.1 – Запит на введення команди.*

Даний інтерфейс передбачає декілька класів команд:

1. CRUD-команди: `find’, ‘add’, ‘update’, ‘remove’;
2. Діагностичні команди: ‘tables’, ‘interpolations’, ‘show’;
3. Довідкові команди: ‘help’.

У програмі присутня довідка, яка коротко описує синтаксис та принцип роботи усіх доступних команд. На спробу виконання невалідної команди, програма відповідає “Invalid query.” Та продовжує виконання, очікуючи наступну команду. Парсер команд працює лінійно, керуючись першою командою. Наприклад, після ключового слова find повинна слідувати назва таблиці, за якою слідує ключове слово where та умова. Приклад валідного запиту виглядає наступним чином: find flights where $0 == NP2003 || ($1 > 100 && $1 < 200). Якщо першою командою є find, а наступна не є валідною назвою таблиці – запит вважається помилковим. Якщо вказана таблиця існує, але за її назвою не слідує “where” запит також відхиляється. У випадку синтаксичних помилок в умові пошуку, користувач бачить повідомлення про помилку з посиланням на невалідний оператор/операнд, а умова вважається такою, що завжди є неправдивою (false), тож їй не відповідатиме жодне поле вказаної таблиці. Парсинг інших команд відбувається аналогічним чином.

## **6.2. СТРУКТУРА ОРГАНІЗАЦІЇ ДАНИХ У КЛАСАХ**

За збереження користувацьких даних відповідають класи, декларація яких міститься у файлах: ‘Flight.h’, ‘Plane.h’, ‘Passanger.h’. У файлі ‘Entry.hpp’ міститься суперклас, який є базовим класом для усіх щойно перелічених класів. Саме він інкапсулює в собі логіку роботи з БД на найнижчому рівні абстракції (пошук по записах, заміна записів, синхронізація поточного стану БД з її представленням у файловій системі тощо).

Розглянемо принцип роботи класу Flight. Одним з основних членів даного класу є статичний вектор flights, який зберігає у собі об’єкти цього ж класу. При запуску програми цей вектор синхронізується з файловою системою (кожен запис у файлі – 1 об’єкт у векторі), та синхронізується з оновленнями даних від користувача (видалення записів, додавання, їх зміна). Цей вектор є дуже зручним для використання коли необхідно оновити усі записи у файлі або вивести всю таблицю на екран. У такому разі запускається метод серіалізації усіх записів Flight::serialize, який повертає загальну структуру даної таблиці (vector<vector<string >>), яку уже можна використовувати у методах FS::writeFile, Table::render. Цей метод є обгорткою над методом Entry::serializeEntry, який в свою чергу циклічно викликає Flight::serializeLn, який повертає структуру типу vector<string> що містить в собі 1 запис для даного об’єкту, з яким було викликано Flight::serializeLn. Приватні текстові поля класу Flight – це власне дані, які зберігаються у кожному об’єкті, якими і оперує метод Flight::serializeLn.У класі присутні поліморфні конструктори, але не всі вони застосовуються для створення корисних об’єктів відповідних записів. Основний конструктор приймає структуру запису (vector<string>) та викликає метод Flight::setFields, який в свою чергу і займається записом отриманої структури у відповідні поля об’єкта. Причиною винесення логіки запису даних у поля об’єкта є метод Flight::update, який також вимагає реалізацію інтерфейсу запису даних у поля об’єкта. Для того щоб не дублювати дану реалізацію мною і було прийнято рішення виділити її у окремий метод.

Другий конструктор Flight::Flight(), що не приймає параметрів не використовується для створення записів у БД, він ‘не знає’ про існування файлової системи чи Flight::flights, натомість його задача полягає у створенні об’єкту-заглушки типу Flight. Цей об’єкт необхідний для vector::push\_back, оскільки даний метод повинен володіти інформацією про фізичний розмір об’єктів класів для динамічного виділення відповідного об’єму пам’яті для них. Детальніше даний механізм описано у розділі «Реалізація класу vector на заміну std::vector» цієї курсової роботи.

Статичний метод Flight::updateFS викликає серіалізатор даного класу, який збирає у єдиний вектор дані з усіх членів вектора Flight::flights, і передає їх у Table::TFlights.writeFile. Останній метод реалізує інтерфейс роботи з файловою системою, тож клас Flight є абстрагованим від неї. Flight::updateFS викликається кожного разу коли є необхідність зафіксувати внесені користувачем зміни. Наприклад, коли він видаляє, змінює, видаляє записи. Натомість, коли відбувається початкова ініціалізація об’єктів даного класу на основі даних, отриманих з файлів, синхронізація з ФС непотрібна.

Метод Flight::factory є практичною реалізацією паттерну програмування Factory. Він уніфікує процес створення екземплярів даного класу (на базі даних, отриманих у вигляді vector<string>), та автоматично зберігає ці екземпляри у Flight::flights. Даний «завод» використовується для динамічного створення об’єктів на базі даних, прочитаних із відповідного файлу (див. Table::parse).

Усі інші методи (‘find’, ’update’, ‘remove’) є реалізацією функціоналу CRUD (create, read, update, delete). Дані методи є статичними і оперують над вектором Flight::flight, оскільки кожен із них повинен мати доступ не до властивостей 1 конкретного об’єкта, а до усіх властивостей усіх об’єктів класу для пошуку по них об’єктів, що відповідають умові пошуку.

З усіх методів класу Flight лише Flight::setFields та Flight::serializeLn є власними унікальними методами даного класу. Це пов’язано з тим, що архітектура кожного класу даних (таблиці) БД є різною, тож різні класи реалізують різні поля та повинні мати різну логіку для внутрішньої роботи з цими даними. Згадані методи за способом перетворення даних є протилежними: setFields трансформує вектор у властивості об’єктів, а serializeLn навпаки – об’єднує властивості поточного об’єкту у вектор. Усі інші загальнодоступні (public) методи класу Flight є обгортками над відповідними методами базового класу Entry, яким додатково передається масив Flight::flights, для роботи над ним. Необхідність безпосереднього передавання даного вектора обумовлена тим, що клас Entry реалізує виключно template-методи, за рахунок чого є абслютно поліморфним і абстрагованим від класів Flight, Plane, Passanger, кожен з яких має однакові вимоги до інтерфейсу роботи з БД. Дублювати однаковий код роботи методів `find`, `update` між цими класами було б недоцільно, тому і було створено клас Entry, який збирає усю вищезгадану логіку в 1 місці.

Методи Entry::findEntries та Entry::updateEntries використовують однаковий алгоритм пошуку записів у БД, що відповідають заданим критеріям пошуку. Дані методи у якості результату повертають вектор записів, що відповідають заданим критеріям. Пошук і відбір записів є обов’язковим, оскільки зазвичай при роботі з БД користувач повинен мати можливість фільтрувати записи, виводячи лише ті, що його цікавлять і, аналогічно, редагувати не усі записи у базі, а лише ті, що його цікавлять. Дана логіка була інкапсульована у приватному методі getMatchingEntries, який повертає вектор об’єктів, що відповідають запиту. Перевірка на відповідність запиту здійснюється класом ExpressionBuilder, його детальніше описано у главі Expression Builder даної курсової роботи.

Варто зазначити, що реалізація та принцип роботи класів Plane та Passanger повністю відповідає реалізації класу Flight за очевидним винятком внутрішньої архітектури приватних властивостей (репрезентацій полів таблиці) і, відповідно, методів serializeLn та setFields.

## **6.3. РЕАЛІЗАЦІЯ TABLE ТА ПОВ’ЯЗАНИХ КЛАСІВ**

Клас Table містить загальнодоступні статичні поля цього ж типу Table, кожне з яких містить об’єкт даного класу, що відповідає відповідній таблиці бази даних (TFlights, TPlanes, TPassangers). Ці поля є основним зовнішнім інтерфейсом даного класу, шляхом виклику методів над ними здійснюється робота окремо з кожною таблицею (читання з файла, парсинг даних, запис у файл). Приватними є властивості Table::headers та Table::serializer. Перше поле – це вектор заголовків даної таблиці, який використовується для запису у файл і додається перед виведенням результатів виконання будь-якого CRUD-запиту користувача. Друге поле є більш цікавим. Воно становить собою примітивну реалізацію концепції callback-функцій засобами C++ і містить вказівник на функцію serialize відповідного класу (Flight, Passanger тощо). При ініціалізації кожної таблиці (див. Main.cpp) конструктору Table::Table окрім назви файла, що відповідає даній таблиці також передається вказівник на serializer та factory відповідного класу. Серіалізатор викликається кожного разу коли необхідно вивести усю табличку цілком, таким чином гарантується актуальність даних, що виводяться у будь-який момент часу (див. Table::render). Метод Table::parse очікує «сирі» дані з щойно прочитаного файлу. Його задача полягає у парсингу цих даних і запису їх у поля відповідних об’єктів класів Flight, та інших. Однак, прямого доступу до цих даних метод Table::parse не має, оскільки вони є приватними і не повинні мати можливість їх зміни ззовні. Саме тому даний метод також отримує вказівник на метод factory відповідного класу, якому він і передає розпізнану строку, що відповідає запису у файлі. Таким чином реалізовано механізм функцій зворотнього виклику у даному класі. Варто зазначити, що метод Table::parse циклічно проходить по усіх рядках із файлу, але оскільки по внутрішній конвенції першим рядком будь-якого файлу є список заголовків таблиці, перша ітерація цього циклу не передає розпізнані дані у метод factory, як цей механізм описано вище. Натомість, ці дані поміщаються у приватний вектор Table::headers. Цей вектор є приватним, оскільки після первинної ініціалізації не повинен мати можливість бути перевизначеним ззовні, але деякі класи повинні отримати доступ до списку заголовків будь-якої з таблиць (мова йде про клас Cmd та ін.), тому реалізовано загальнодоступний (public) метод getHeaders, який просто повертає список заголовків таблиці, відносно якої його було викликано.

Клас Table наслідується від класу FS, щоб бути абстрагованим від усіх нюансів роботи з файловою системою на найнижчому рівні абстракції. Клас FS у дескрипторі protected зберігає назву таблиці (яка за сумісництвом є назвою відповідного файла). Це поле не є приватним, оскільки його задає і використовує дочірній клас Table. Також клас FS реалізує методи readFile та writeFile, які звертаються до відповідного файлу (FS::tableName). здійснюють читання/запис даних. FS::readFile не приймає параметрів і повертає вектор лінійок, зчитаних з файла (цей вектор використовується методом Table::parse, описаним вище), а FS::writeFile нічого не повертає, але очікує отримати параметром вектор усіх записів (дані, серіалізовані методом serialize).

Коцептуально пов’язаним з класом Table є клас Renderer. Однак, вони не пов'язані наслідуванням. Даний клас реалізує логіку виведення таблиці в консоль. Для цього в private секції зберігаються внутрішні дані про графічну репрезентацію таблички (ширина таблиці, ширина найширшого стовпця (для вирівнювання усіх стовпців по ширині), кількість стовпців тощо).

Метод padCenter – це внутрішня утиліта, що приймає число – довжину результуючої строки і оригінальну строку. Його задача полягає в тому, щоб симетрично додати пробіли зліва та справа оригінальної строки так, щоб вона виявилась «відцентрованою» по заданій ширині.

Метод analyse призначений для первинного аналізу заданої таблиці з метою визначення ширини найширшого стовпця, загальної ширини таблиці тощо. Пізніше дані, отримані в результаті цього статичного аналізу будуть використані методом Renderer::render, який власне і виводить графічне представлення таблиці у консоль. Для забезпечення естетичного вигляду виведеної таблички зовнішній кут з кожного боку замінюється на символ «+», а стовпці відділені між собою символом «|», для того щоб визначити, коли який символ необхідно вивести, існує метод getRowSeparator, який повертає 1 з цих 2 символів по заданій умові.

Конструктор даного класу є перевантаженим (поліморфним). Представлено 2 його версії: з 2 параметрами та з 1. Якщо конструктор викликається з 1 – цей параметр вважається суцільною табличкою і виводиться на екран. Якщо з 2 – перший з них вважається списком заголовків, а другий – списком даних, які містить таблиця. Реалізація кожного з них передбачає виклик методу Renderer::analyse для статичного аналізу вхідних даних.

## **6.4. РЕАЛІЗАЦІЯ КЛАСУ VECTOR НА ЗАМІНУ STD::VECTOR**

Для повноцінної реалізації завдання даної курсової роботи було необхідно раціонально використовувати ресурси, надані комп’ютером. Оперувати масивами з попередньо визначеним розміром було б не надто ефективно з точки зору зайнятої оперативної пам’яті, тому від цього способу представлення даних у додатку довелось відмовитись. Натомість я реалізував динамічний масив (вектор), який має можливість підлаштовувати зайнятий простір пам’яті під реальні потреби користувача, тому ресурси не було змарновано. Внутрішнім представленням вектора є звичайний масив статичного розміру. При додаванні нового елемента в кінцець вектора (vector::push\_back) цей масив копіюється у нове місце у пам'яті, за яким слідує елемент, який користувач бажає додати. Пам’ять, зайнята старою копією масиву при цьому звільняється. Даний підхід не є цілком оптимальним, оскільки фактично було обміняно ресурс пам’яті на ресурс процесора, оскільки операція копіювання займає деякий процесорний час, однак я не вважаю фактор швидкодії критичним для консольного додатку, тому і вирішив реалізувати саме такий підхід. Також серед приватних полів класу vector знаходиться поле curr\_size, яке просто є лічильником зайнятої довжини масива, і використовується при запиті роміру вектора користувачем (vector:size) та при перевірці на доступність елемента з даним індексом у векторі (vector::operator[]) (захист від помилок сегментації пам’яті). Також реалізовані допоміжні методи: vector::erase – копіювання усього масиву у нову область пам’яті за винятком 1 елемента, vector::back – повернення останнього елемента масиву, та інші.

## **6.5. EXPRESSION BUILDER**

Конструктор запитів використовується для фільтрування полів таблиці при їх пошуку, зміні та видаленні, застосовується у Entry::getMatchingFields. На вхід конструктор даного класу очікує отримати вектор кортежу таблиці для інтерполяції заголовків. Цей функціонал реалізовано з метою позбавлення користувача необхідності кожного разу зазначати складні та довгі назви полів при запитах (Н-д: замість запиту … where Passangers Count < 200 користувач повинен ввести запит … where $1 < 200, за умови, що 1 – це порядковий номер стовпця у таблиці). Цей вектор пізніше використає метод interpolationResolver для підстановки відповідних значень у результуючий запит. Метод ExpressionBuilder::parse приймає строку, яка є строкою запиту. З неї видаляються усі пробіли, після чого вона потрапляє на 1 стадію обчислень.

Оскільки кожен математичний оператор має свій пріоритет виконання (кон’юнкція передує диз’юнкції, дужки мають найвищий пріоритет тощо), у даного класу є метод, який визначає пріоритет даного оператора і повертає його у числовому виді, де 0 – найвищий, ∞ - найнижчий. Називається цей метод getOperatorPriority. Обробка кожного запиту поділена на кілька стадій згідно з пріоритетом виконання (кожна стадія реалізована в межах 1 методу).

1 стадія – parenthesisResolver – виконує оператори «зсередини назовні», на кожній стадії позбавляючись однієї пари дужок;

2 стадія – priorityResolver – виконує логічні та алгебраїчні оператори у порядку їх пріоритетів, приймає на вхід вираз без дужок (усі дужки було усунено на стадії 1);

3 стадія – interpolationResolver – виконує підстановку реальних даних з масиву substitutions у результуючий вираз для подальших обчислень;

4 стадія – typeResolver – визначає типи операндів кожного окремого оператора і визначає метод для їх обробки: 1 із 2 перевантажних методів executor (для строкових та числових типів відповідно).

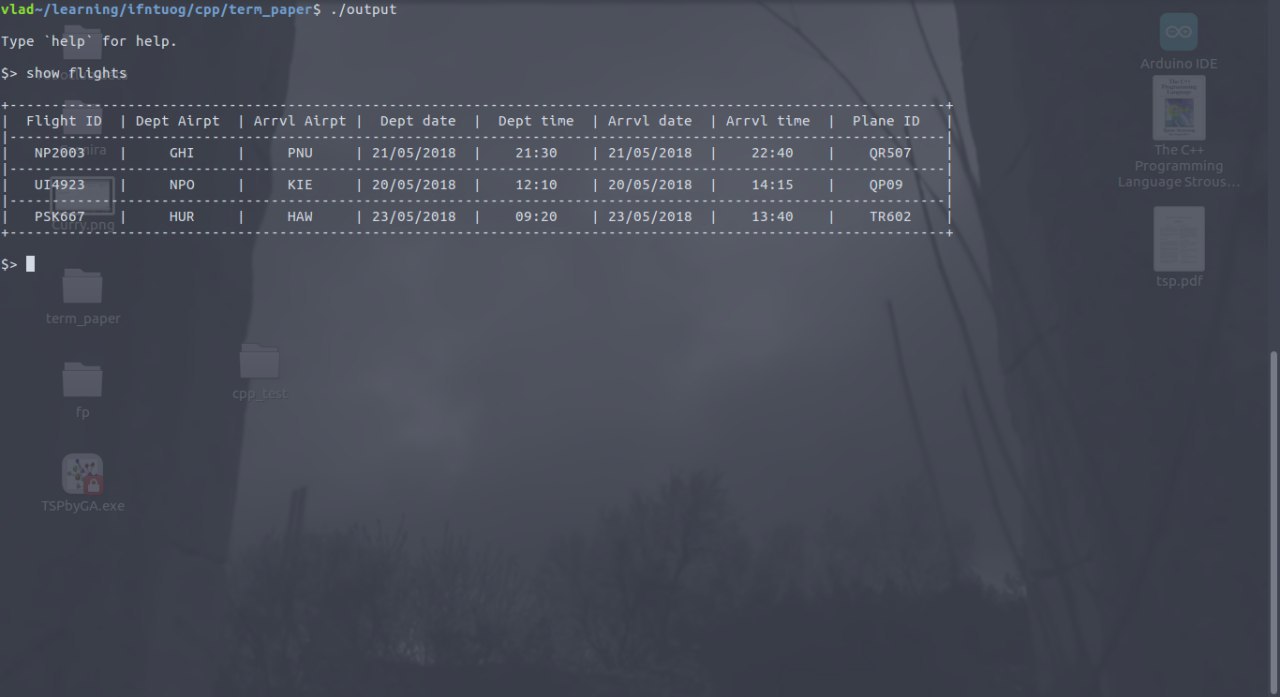
5 стадія – executor – виконання елементарних логічних/алгебраїчних операцій, повернення результуючих літералів на попередню стадію обробки.

Варто зазначити, що кожна з цих стадій є рекурсивною і виконується доти, доки у запиті присутні дані, якими може оперувати метод даної стадії, якщо таких немає, управління передається наступній стадії і так далі.

Розглянемо метод parenthesisResolver. Він статично аналізує строку, шукаючи найглибший рівень вкладеності дужок. Позиція кожної знайденої відкриваючої дужки (символу «(») записується у вектор openParenthesis. Алгоритм визначення найглибшої пари дужок є схожим на алгоритм пошуку найбільшого елемента масиву. Перед аналізом строки поточний рівень вкладеності вважається рівним -1. З кожною знайденою дужкою «(», значення лічильника інкрементується, а позиція даної дужки записується у вектор openParenthesis. При виявленні символу «)», у пару deepestCmdPos записуємо координату початку команди, що знаходиться між даною дужкою «)» і відповідною їй «(», а така відповідна дужка гарантовано є останньою у векторі openParenthesis, координати якої уже відомі. Другим значенням пари deepestCmdPos є координата кінця вкладеної команди, яка також є координатою знайденого символу «)». На даному етапі не можна вважати значення deepestCmdPos координатами найглибшої команди, оскільки дані дужки, які щойно були проаналізовані можуть всередині містити інші дужки. Тому процес продовжується циклічно до кінця строки, щоб гарантовано мати координати найглибшої команди, яка не містить дужок (Н-д ‘0 && 2', '0 || '1 && 2 != 5‘ тощо). Отримавши координати даної команди, ми можемо замість неї і відповідних дужок, що її огортають підставити результат її виконання і рекурсивно продовжити пошук найглибших дужок, доки їх у вихідній строці запиту не залишиться. Для цього строка запиту обрізається наступним чином: cmdBefore - від початку строки до початку даної команди, cmdAfter - від її кінця до кінця строки. Дані ділянки можуть містити інші дужки, та на даному етапі це неважливо. Сама команда передається на обробку наступній стадії (priorityResolver), а результат, який повертається (літерал, що є результатом виконання даної команди) підставляється у вихідну строку між cmdBefore та cmdAfter, і рекурсивно передається назад у parenthesisResolver. Даний алгоритм дозволяє опрацьовувати дужки необмеженого рівня вкладеності. Точкою виходу з цієї рекурсія є умова відсутності символів «(» у строці запиту. Коли вона виконується, можна вважати, що на вхід даний метод отримав строку з операторами, без дужок, тому шукати їх не треба, достатньо просто передати дану строку на наступну стадію обробки. Такою стадією є priorityResolver. Принцип роботи даного методу схожий на принцип роботи попереднього, за винятком того, що він шукає не найглибшу пару дужок, а оператор з найвищим пріоритетом для виконання. Аналогічним чином, вхідна строка ділиться на ділянки. Та після знаходження найпріоритетнішого оператора необхідно також правильно відділити його операнди. Для цього у векторі allCmdsPos фіксуються координати абсолютно усіх операторів незалежно від їх пріоритету. Відомо, що лівий операнд найпріоритетнішого оператора починається на координаті початку строки або на координаті кінця попереднього оператора зліва, якщо такий є, закінчується ж лівий операнд власне початком найпріоритетнішого оператора. Аналогічне правило застосовується для правого операнда. Маючи координати найбільш пріоритетного оператора і його операндів, строка запиту знову розбивається по цих координатах, за принципом початок + результат виконання оператора над його операндами (забезпечується наступною стадією - interpolationsResolver) + кінець. Отримана строка рекурсивно передається цьому ж методу для подальшого пошуку пріоритетного оператора, та таким чином виконання операторів один за одним. Умовою виходу з рекурсії є відсутність у строці запиту операторів. Наступний етап – підстановка даних з масиву substitutions. За ним слідує визначення типів операндів, оскільки для чисел реалізовані окремі оператори (|| та &&), усі інші оператори є поліморфними, однак оператор ~ (є підстрокою) працює лише зі строками і не має змісту для чисел.

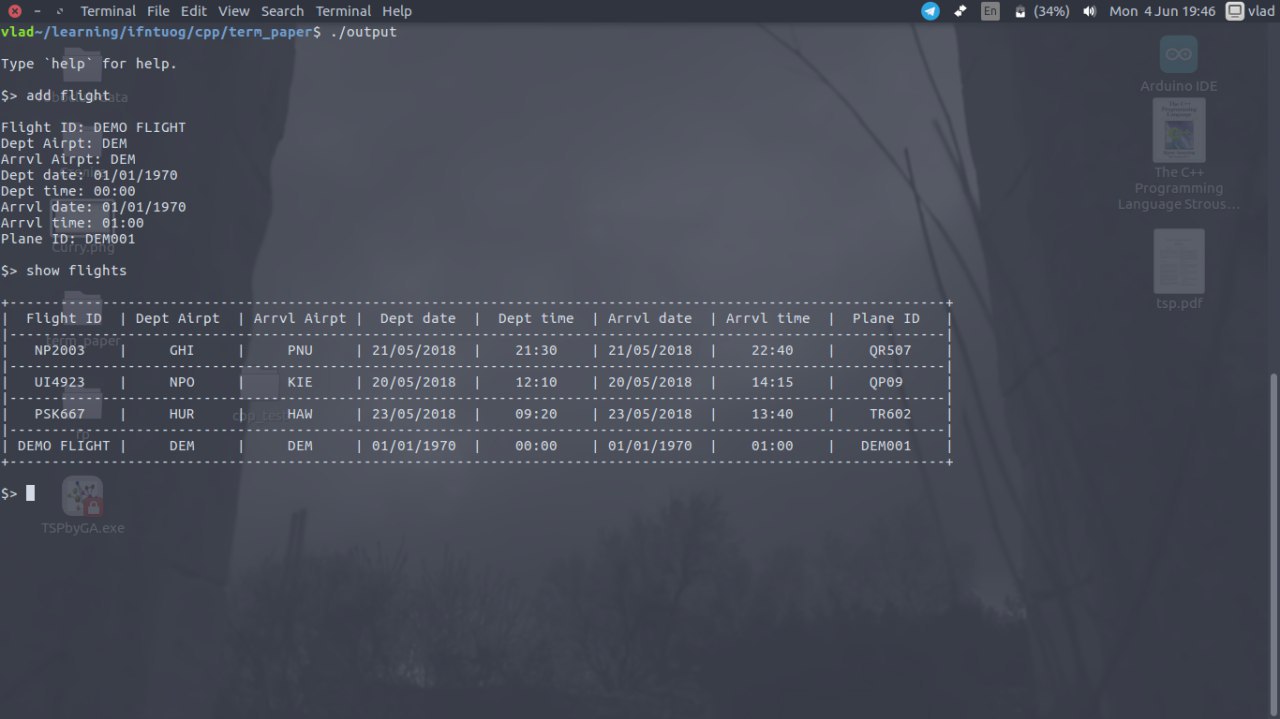
Таким чином, з використанням рекурсії, поліморфізму та логічного поділу коду на окремі методи я і реалізував Expression Builder.

## **6.6. ПРИКЛАД РОБОЧОГО ПРОЦЕСУ**



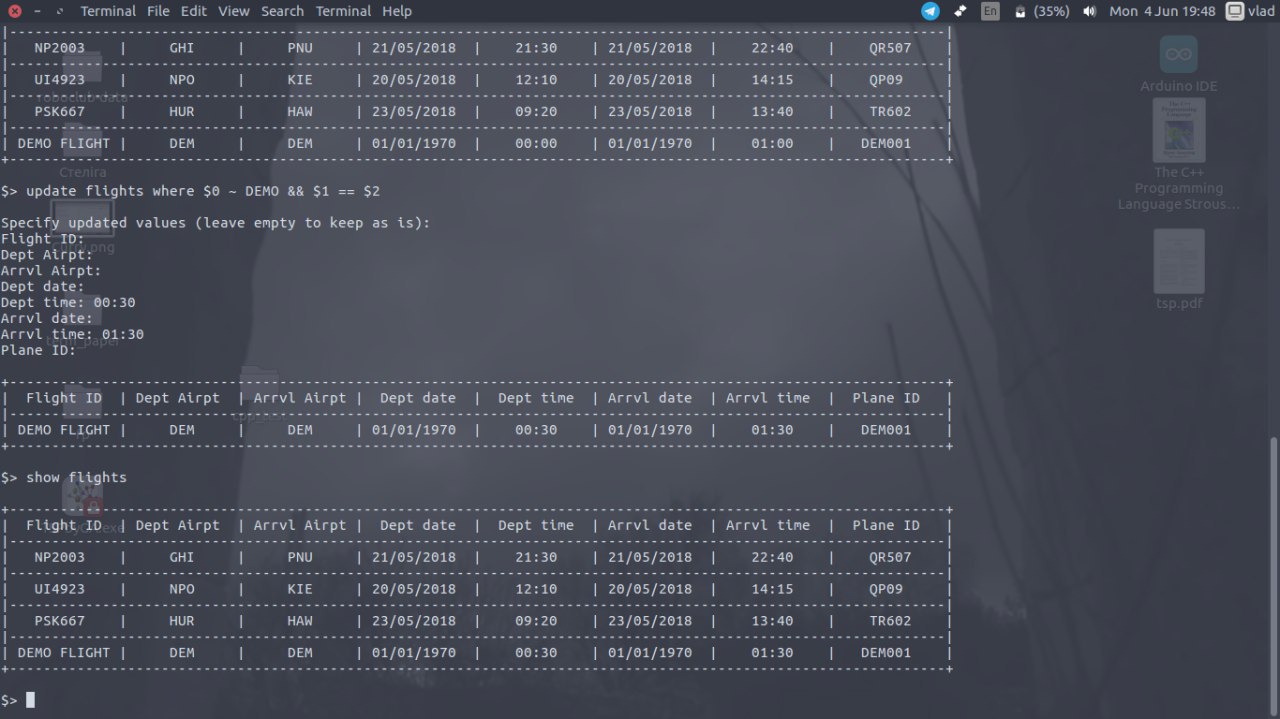
*Рисунок 6.6.1 – Виведення таблиці `flights`*

Результат виконання будь-якої команди, пов’язаної з виведенням результуючої таблиці у консоль зображено на рисунку 6.6.1. Кожна колонка таблички має ширину найширшої колонки, а текст всередині кожної клітинки відцентровано.



*Рисунок 6.6.2 – Додавання нового запису у базу*

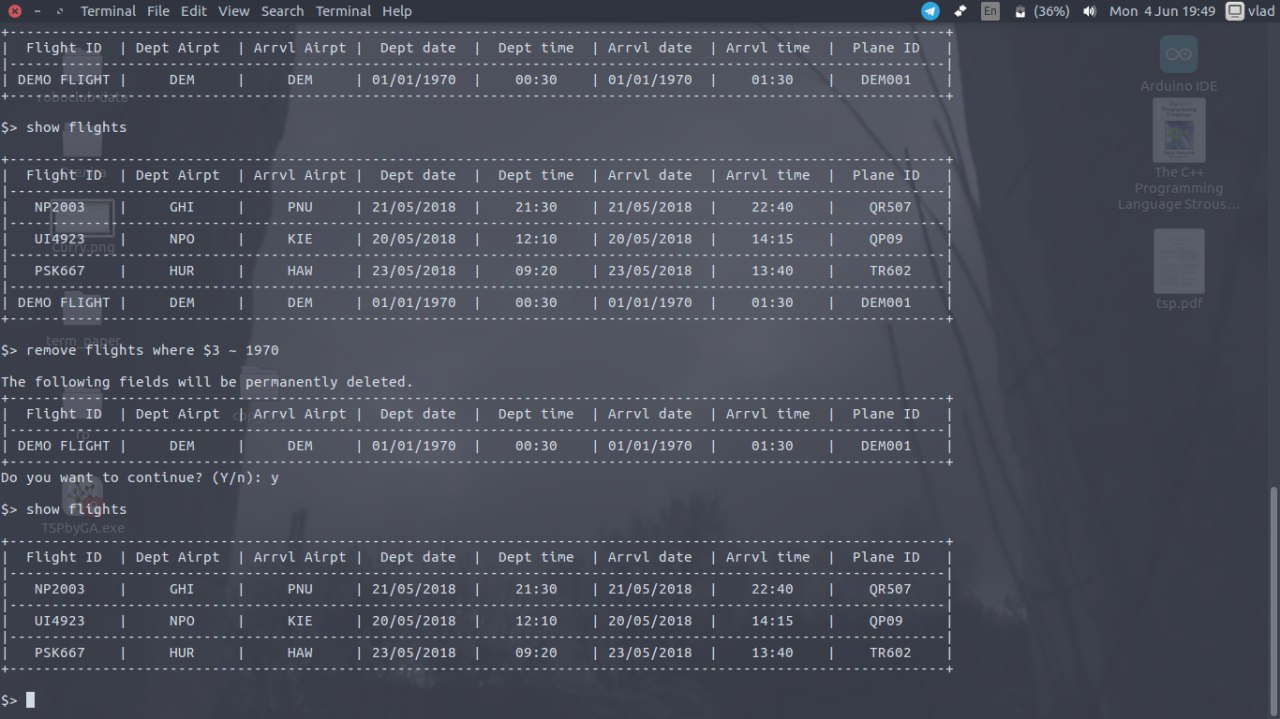
При додавання нового запису до таблиці, програма запрошує користувача ввести дані кожного поля по черзі.



*Рисунок 6.6.3 – редагування запису (записів) у базі*

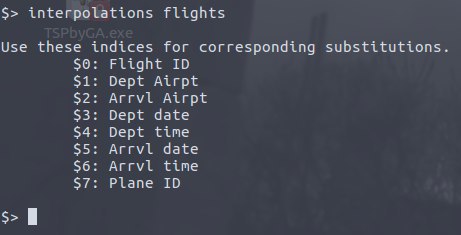
У користувача є можливість не дублювати попередньо введені дані, якщо необхідно змінити значення лише деяких полів. Для цього потрібно залишити значення поля, яке залишиться незмінним порожнім. Після виконання програма окремо виведе ділянку таблиці з оновленими значеннями.

Програма також дозволяє пакетну обробку запитів редагування та видалення записів, за замовчуванням дія виконується над усіма полями, для яких справедливою є введена умова.



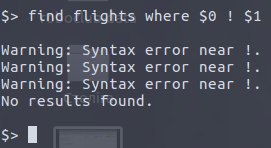
*Рисунок 6.6.4 – Видалення запису (записів) із бази*

Оскільки видалення може виконуватись пакетно і є відповідальною операцією, перед її проведенням програма виводить таблицю, вмістом якої є зведені кортежі таблиці, що відповідають введеному запиту, та запитує у користувача підтвердження їх видалення. Ця операція виконується лише після отримання позитивної відповіді.

**

*Рисунок 6.6.5 – Виведення списку підстановок даної таблиці*

Оскільки список замін не є практичним для запам’ятовування, у програмі передбачено можливість його виводу командою interpolations для кожної конкретної таблиці окремо.

**

*Рисунок 6.6.6 – Помилка аналізу виразу Expression Builder’ом*

У разі неможливості перевірити поля таблиці на відповідність заданому виразу (некоректність виразу) виводиться повідомлення про помилку (рисунок 6.6.6) та запит вважається таким, що не дав результатів. Програма продовжує виконання, очікуючи на наступну команду.

# **7. ВИСНОВОК**

Результатом виконання курсової роботи стала програма для контролю роботи аеропорту, яка не була написана з метою комерціалізації, а виключно в навчальній та пізнавальній цілях. Розроблений додаток відповідає всім вимогам предметної області.

Дана програма реалізує зручний, призначений для користувача, консольний інтерфейс. Вона дозволяє вирішувати всі завдання, сформульовані в завданні на курсову роботу. Це дозволяє зробити висновок, що завдання виконано повністю.

# **ДОДАТКИ**

## **ДОДАТОК А: ЛІСТИНГ ПРОГРАМИ**

**Cmd.h**

#ifndef CMD\_H

#define CMD\_H

#include <string>

#include "Vector.hpp"

using namespace std;

class Cmd {

private:

vector<string> getHeadersByTableName(string const&);

vector<string> getReplacements(vector<string>&);

void simpleHandler(string const&);

void complexHandler(string const&);

void show(string const&);

void add(string const&);

void update(string const&, string const&);

void remove(string const&, string const&);

void tables();

void find(string const&, string const&);

void interpolations(string const&);

void help();

void invalidMsg();

void noTableMsg(string const&);

void enoentMsg();

public:

Cmd(string const&);

};

#endif

**Cmd.cpp**

#include <iostream>

#include <string>

#include <sstream>

#include "Cmd.h"

#include "Table.h"

#include "Renderer.h"

#include "Utils.h"

#include "Flight.h"

#include "Passanger.h"

#include "Plane.h"

#include "Vector.hpp"

using namespace std;

void Cmd::invalidMsg() {

cout << "Invalid query." << endl;

}

void Cmd::simpleHandler(string const& cmd) {

if(cmd == "help") help();

else if(cmd == "tables") tables();

else invalidMsg();

}

void Cmd::complexHandler(string const& cmd) {

vector<string> operands = Utils::strSplitBySpace(cmd);

int whereClausePosition = cmd.find("where");

if(operands[0] == "show") show(operands[1]);

else if(operands[0] == "add") add(operands[1]);

else if(operands[0] == "interpolations") interpolations(operands[1]);

else if(operands.size() > 2 && whereClausePosition != string::npos) {

string expression = Utils::slice(cmd, whereClausePosition + 5, cmd.length());

if(operands[0] == "find") find(operands[1], expression);

else if(operands[0] == "update") update(operands[1], expression);

else if(operands[0] == "remove") remove(operands[1], expression);

}

else invalidMsg();

}

Cmd::Cmd(string const& cmd) {

cout << endl;

string command = Utils::trim(cmd);

if(command.length() == 0) return;

if(command.find(" ") == string::npos) simpleHandler(command);

else complexHandler(command);

cout << endl;

}

void Cmd::noTableMsg(string const& tableName) {

cout << "Table `" << tableName << "` doesn't exist." << endl;

}

void Cmd::enoentMsg() {

cout << "No results found." << endl;

}

void Cmd::show(string const& tableName) {

if(tableName == "flights") Table::TFlights.render();

else if(tableName == "passangers") Table::TPassangers.render();

else if(tableName == "planes") Table::TPlanes.render();

else noTableMsg(tableName);

}

typedef void (\*Factory)(vector<string>, bool);

void Cmd::add(string const& fieldName) {

vector<string> headers;

Factory factory;

if(fieldName == "flight") {

headers = Table::TFlights.getHeaders();

factory = Flight::factory;

}

else if(fieldName == "passanger") {

headers = Table::TPassangers.getHeaders();

factory = Passanger::factory;

}

else if(fieldName == "plane") {

headers = Table::TPlanes.getHeaders();

factory = Plane::factory;

}

else {

cout << "`" << fieldName << "` is not a valid member." << endl;

return;

}

vector<string> data;

for(int i=0; i<headers.size(); i++) {

cout << headers[i] << ": ";

string value;

getline(cin, value);

data.push\_back(value);

}

factory(data, true);

}

void Cmd::interpolations(string const& tableName) {

vector<string> headers = getHeadersByTableName(tableName);

if(headers.size()) {

cout << "Use these indices for corresponding substitutions." << endl;

for(int i=0; i<headers.size(); i++)

cout << "\t$" << i << ": " << headers[i] << endl;

} else noTableMsg(tableName);

}

vector<string> Cmd::getReplacements(vector<string>& headers) {

vector<string> replacements;

cout << "Specify updated values (leave empty to keep as is):" << endl;

for(int i=0; i<headers.size(); i++) {

cout << headers[i] << ": ";

string newValue;

getline(cin, newValue);

replacements.push\_back(newValue);

}

cout << endl;

return replacements;

}

vector<string> Cmd::getHeadersByTableName(string const& tableName) {

vector<string> headers;

if(tableName == "flights") headers = Table::TFlights.getHeaders();

else if(tableName == "planes") headers = Table::TPlanes.getHeaders();

else if(tableName == "passangers") headers = Table::TPassangers.getHeaders();

return headers;

}

void Cmd::find(string const& tableName, string const& expression) {

vector<string> headers = getHeadersByTableName(tableName);

if(headers.size()) {

vector<vector<string>> results;

if(tableName == "flights") results = Flight::find(expression);

if(tableName == "planes") results = Plane::find(expression);

if(tableName == "passangers") results = Passanger::find(expression);

if(results.size()) {

Renderer resultingTable(headers, results);

resultingTable.render();

} else enoentMsg();

}

else return noTableMsg(tableName);

}

void Cmd::update(string const& tableName, string const& expression) {

vector<string> headers = getHeadersByTableName(tableName);

if(headers.size()) {

vector<vector<string>> results;

vector<string> replacements = getReplacements(headers);

if(tableName == "flights") results = Flight::update(expression, replacements);

if(tableName == "planes") results = Plane::update(expression, replacements);

if(tableName == "passangers") results = Passanger::update(expression, replacements);

if(results.size()) {

Renderer resultingTable(headers, results);

resultingTable.render();

} else enoentMsg();

}

else return noTableMsg(tableName);

}

typedef void (\*Remover)(string);

void Cmd::remove(string const& tableName, string const& expression) {

vector<string> headers = getHeadersByTableName(tableName);

if(headers.size()) {

vector<vector<string>> results;

Remover remover;

if(tableName == "flights") { results = Flight::find(expression); remover = Flight::remove; }

if(tableName == "planes") { results = Plane::find(expression); remover = Plane::remove; }

if(tableName == "passangers") { results = Passanger::find(expression); remover = Passanger::remove; }

if(results.size()) {

cout << "The following fields will be permanently deleted." << endl;

Renderer resultingTable(headers, results);

resultingTable.render();

cout << "Do you want to continue? (Y/n): ";

string answer;

getline(cin, answer);

if(answer[0] == 'Y' || answer[0] == 'y') remover(expression);

else cout << "Aborting..." << endl;

}

else enoentMsg();

}

else return noTableMsg(tableName);

}

void Cmd::tables() {

cout << "flights" << endl;

cout << "passangers" << endl;

cout << "planes" << endl;

}

void Cmd::help() {

cout << "tables - print all existing tables' names." << endl;

cout << "find Tabl where Expr - find field(s) in table T, which match the `where` clause." << endl;

cout << "show Tabl - show table Tabl." << endl;

cout << "add Inst - add instance of table Tabl." << endl;

cout << "update Tabl where Expr - update field(s) in table T, which match Expr." << endl;

cout << "remove Tabl where Expr - remove field(s) in table T, which match Expr." << endl;

cout << " " << "Supported Expr operators:" << endl;

cout << '\t' << "priority [()];" << endl;

cout << '\t' << "interolation [$N] (N being the row index)." << endl;

cout << '\t' << "comparison [<, <=, >, >=, ==, !=, ~];" << endl;

cout << '\t' << "boolean [&&, ||];" << endl;

cout << '\t' << "Sample Expr: ($0 <= 200 && $1 != AB1234) || $2 == 100" << endl;

cout << "interpolations Tabl - print an interpolations list for table Tabl.";

cout << "help - print this manual." << endl;

cout << "exit - leave the CLI." << endl;

}

**Entry.hpp**

#ifndef ENTRY\_H

#define ENTRY\_H

#include <string>

#include "ExpressionBuilder.h"

#include "Table.h"

#include "Vector.hpp"

using namespace std;

class Entry {

protected:

template<class T> static vector<T\*> getMatchingEntries(string const&, vector<T\*>&);

template<class T> static vector<vector<string>> findEntries(string const&, vector<T\*>&);

template<class T> static vector<vector<string>> serializeEntries(Table&, vector<T\*>&);

template<class T> static void removeEntries(string const&, vector<T\*>&);

template<class T> static void entriesFactory(vector<T\*>&, vector<string>, bool);

template<class T> static vector<vector<string>> updateEntries(string const&, vector<string>, vector<T\*>&);

};

template<class T>

vector<T\*> Entry::getMatchingEntries(string const& query, vector<T\*>& entries) {

vector<T\*> result;

for(int i=0; i<entries.size(); i++) {

vector<string> line = entries[i]->serializeLn();

ExpressionBuilder expression = ExpressionBuilder(line);

bool lineMatches = expression.parse(query);

if(lineMatches) result.push\_back(entries[i]);

}

return result;

}

template<class T>

vector<vector<string>> Entry::findEntries(string const& query, vector<T\*>& entries) {

vector<vector<string>> result;

vector<T\*> matchingEntries = getMatchingEntries(query, entries);

for(int i=0; i<matchingEntries.size(); i++)

result.push\_back(matchingEntries[i]->serializeLn());

return result;

}

template<class T>

vector<vector<string>> Entry::serializeEntries(Table& parent, vector<T\*>& entries) {

vector<vector<string>> result;

result.push\_back(parent.getHeaders());

for(int i=0; i<entries.size(); i++)

result.push\_back(entries[i]->serializeLn());

return result;

}

template<class T>

void Entry::removeEntries(string const& query, vector<T\*>& entries) {

for(int i=0; i<entries.size(); i++) {

ExpressionBuilder expression = ExpressionBuilder(entries[i]->serializeLn());

bool lineMatches = expression.parse(query);

if(lineMatches) entries.erase(i--); // Decrement is needed due to entries shift when erasing occurs.

}

T::updateFS();

}

template<class T>

void Entry::entriesFactory(vector<T\*>& entries, vector<string> rawData, bool fsSync) {

T\* obj = new T(rawData);

entries.push\_back(obj);

if(fsSync) T::updateFS();

}

template<class T>

vector<vector<string>> Entry::updateEntries(string const& query, vector<string> fields, vector<T\*>& entries) {

vector<vector<string>> result;

vector<T\*> matchingEntries = getMatchingEntries(query, entries);

for(int i=0; i<matchingEntries.size(); i++) {

matchingEntries[i]->setFields(fields);

result.push\_back(matchingEntries[i]->serializeLn());

}

T::updateFS();

return result;

}

#endif

**ExpressionBuilder.h**

#ifndef EXPRESSIONBUILDER\_H

#define EXPRESSIONBUILDER\_H

#include <string>

#include "Vector.hpp"

using namespace std;

class ExpressionBuilder {

private:

static string specialSymbols;

vector<string> substitutions;

int getOperatorPriority(string);

int executor(string, string, string);

int executor(string, int, int);

int typeResolver(string, string, string);

int interpolationResolver(string, string, string);

int priorityResolver(string);

int parenthesisResolver(string);

public:

bool parse(string);

ExpressionBuilder(vector<string>);

};

#endif

**ExpressionBuilder.cpp**

#include <string>

#include "Vector.hpp"

#include <iostream>

#include "ExpressionBuilder.h"

#include "Utils.h"

using namespace std;

string ExpressionBuilder::specialSymbols = "!=><~&|";

template <typename T> int polymorphicOperators(string opp, T const lOperand, T const rOperand) {

if(opp == "==") return lOperand == rOperand;

if(opp == "!=") return lOperand != rOperand;

if(opp == "<=") return lOperand <= rOperand;

if(opp == ">=") return lOperand >= rOperand;

if(opp == "<" ) return lOperand < rOperand;

if(opp == ">" ) return lOperand > rOperand;

cout << "Warning: Syntax error near: " << opp << '.' << endl;

return -1;

}

int ExpressionBuilder::executor(string opp, string lOperand, string rOperand) {

int lSize = lOperand.size();

int rSize = rOperand.size();

if(opp == "&&") return lSize && rSize;

if(opp == "||") return lSize || rSize;

if(opp == "~" ) return Utils::contains(lOperand, rOperand);

return polymorphicOperators(opp, lOperand, rOperand);

}

int ExpressionBuilder::executor(string opp, int lOperand, int rOperand) {

if(opp == "&&") return lOperand && rOperand;

if(opp == "||") return lOperand || rOperand;

return polymorphicOperators(opp, lOperand, rOperand);

}

int ExpressionBuilder::getOperatorPriority(string x) {

if(x == "==") return 1;

if(x == "!=") return 1;

if(x == "<" ) return 1;

if(x == ">" ) return 1;

if(x == "<=") return 1;

if(x == ">=") return 1;

if(x == "~" ) return 1;

if(x == "&&") return 2;

if(x == "||") return 3;

cout << "Warning: Syntax error near " << x << '.' << endl;

return -1;

}

int ExpressionBuilder::typeResolver(string opp, string lOperand, string rOperand) {

int lInt = Utils::strToInt(lOperand);

int rInt = Utils::strToInt(rOperand);

bool isLeftInt = lInt != -1;

bool isRightInt = rInt != -1;

if(isLeftInt && isRightInt) return executor(opp, lInt, rInt);

return executor(opp, lOperand, rOperand);

}

int ExpressionBuilder::interpolationResolver(string opp, string \_lOperand, string \_rOperand) {

string lOperand = \_lOperand;

string rOperand = \_rOperand;

int lInt = Utils::strToInt(\_lOperand.substr(1, \_lOperand.length()-1));

int rInt = Utils::strToInt(\_rOperand.substr(1, \_rOperand.length()-1));

bool lIntInRange = lInt >= 0 && lInt < substitutions.size();

bool rIntInRange = rInt >= 0 && rInt < substitutions.size();

if(\_lOperand[0] == '$' && lIntInRange) lOperand = substitutions[lInt];

if(\_rOperand[0] == '$' && rIntInRange) rOperand = substitutions[rInt];

return typeResolver(opp, lOperand, rOperand);

}

int ExpressionBuilder::priorityResolver(string query) {

int queryEnd = query.length();

bool hasSpecialSymbols = false;

int urgentOperatorPriority = 100;

vector<int> urgentOperatorPos;

urgentOperatorPos.push\_back(-1);

urgentOperatorPos.push\_back(-1);

vector<vector<int>> allCmdsPos;

string currOperator;

int currOperatorStartPos = 0;

for(int i=0; i<queryEnd; i++) {

if(Utils::contains(specialSymbols, query[i])) {

currOperator += query[i];

hasSpecialSymbols = true;

} else {

if(currOperator.length()) {

int priority = getOperatorPriority(currOperator);

if(priority == -1) return false;

if(priority < urgentOperatorPriority) {

urgentOperatorPriority = priority;

urgentOperatorPos[0] = currOperatorStartPos;

urgentOperatorPos[1] = i-1;

}

vector<int> tmp;

tmp.push\_back(currOperatorStartPos);

tmp.push\_back(i-1);

allCmdsPos.push\_back(tmp);

}

currOperator.clear();

currOperatorStartPos = i+1;

}

}

if(hasSpecialSymbols) {

int prevOperatorEnd = -1;

int nextOperatorStart = queryEnd;

for(int i=0; i<allCmdsPos.size(); i++) {

int currOperatorStart = allCmdsPos[i][0];

int currOperatorEnd = allCmdsPos[i][1];

int urgentOperatorStart = urgentOperatorPos[0];

int urgentOperatorEnd = urgentOperatorPos[1];

if(

currOperatorStart == urgentOperatorStart &&

currOperatorEnd == urgentOperatorEnd

) continue; // Skip the most urgent operator

// Defining the rightmost operator before the most urgent one

if(currOperatorEnd > prevOperatorEnd && currOperatorEnd < urgentOperatorStart)

prevOperatorEnd = currOperatorEnd;

// Defining the leftmost operator after the most urgent one

if(currOperatorStart < nextOperatorStart && currOperatorStart > urgentOperatorEnd)

nextOperatorStart = currOperatorStart;

}

string lOperand = Utils::slice(query, prevOperatorEnd+1, urgentOperatorPos[0]);

string opp = Utils::slice(query, urgentOperatorPos[0], urgentOperatorPos[1]+1);

string rOperand = Utils::slice(query, urgentOperatorPos[1]+1, nextOperatorStart);

string cmdBefore = Utils::slice(query, 0, prevOperatorEnd+1);

string cmdAfter = Utils::slice(query, nextOperatorStart, queryEnd);

return priorityResolver(cmdBefore + to\_string(interpolationResolver(opp, lOperand, rOperand)) + cmdAfter);

} else {

int queryInt = Utils::strToInt(query);

if(queryInt < 1) return 0;

else return queryInt;

}

}

int ExpressionBuilder::parenthesisResolver(string query) {

int queryEnd = query.length();

vector<int> openParenthesis;

int currentLvl = -1;

int deepestLvl = currentLvl;

int deepestCmdPos[2];

if(Utils::contains(query, "(")) {

// There is at least one pair of brackets

for(int i=0; i<query.length(); i++) {

if(query[i] == '(') {

currentLvl++;

openParenthesis.push\_back(i);

}

if(query[i] == ')') {

if(currentLvl > deepestLvl) {

deepestCmdPos[0] = openParenthesis.back();

openParenthesis.erase(openParenthesis.size()-1);

deepestCmdPos[1] = i;

deepestLvl = currentLvl;

}

currentLvl--;

}

}

string cmdBefore = Utils::slice(query, 0, deepestCmdPos[0]);

string cmd = Utils::slice(query, deepestCmdPos[0]+1, deepestCmdPos[1]);

string cmdAfter = Utils::slice(query, deepestCmdPos[1]+1, queryEnd);

return parenthesisResolver(cmdBefore + to\_string(priorityResolver(cmd)) + cmdAfter);

}

else

// Linear query here

return priorityResolver(query);

}

bool ExpressionBuilder::parse(string str) {

// Remove all whitespaces

string query;

for(int i=0; i<str.length(); i++)

if(str[i] != ' ') query.push\_back(str[i]);

// -1 - An error occured

// 0 - False

// 1 - True

return parenthesisResolver(query) > 0;

}

ExpressionBuilder::ExpressionBuilder(vector<string> \_substitutions) {

substitutions = \_substitutions;

}

**FS.h**

#include <string>

#include "Vector.hpp"

#ifndef FS\_H

#define FS\_H

using namespace std;

class FS {

protected:

string tableName;

public:

vector<string> readFile();

void writeFile(vector<vector<string>>&);

};

#endif

**FS.cpp**

#include <sstream>

#include <fstream>

#include <string>

#include "Vector.hpp"

#include "FS.h"

vector<string> FS::readFile() {

string line;

vector<string> file;

ifstream fss ("./storage/" + tableName + ".txt");

while(getline(fss, line)) file.push\_back(line);

fss.close();

return file;

};

void FS::writeFile(vector<vector<string>>& data) {

ofstream table;

table.open("./storage/" + tableName + ".txt");

for(int i=0; i<data.size(); i++) {

for(int j=0; j<data[i].size(); j++) {

table << data[i][j];

if(j != data[i].size() - 1) table << '\t';

}

table << endl;

}

table.close();

}

**Flight.h**

#ifndef FLIGHT\_H

#define FLIGHT\_H

#include <string>

#include "Vector.hpp"

#include "Entry.hpp"

using namespace std;

class Flight : protected Entry {

private:

static vector<Flight\*> flights;

string flightID;

string departureAirpt;

string arrivalAirpt;

string departureDate;

string departureTime;

string arrivalDate;

string arrivalTime;

string planeID;

public:

static vector<vector<string>> serialize();

static void updateFS();

static vector<vector<string>> find(string);

static vector<vector<string>> update(string, vector<string>);

static void remove(string);

static void factory(vector<string>, bool);

vector<string> serializeLn();

void setFields(vector<string>);

Flight(vector<string>);

Flight();

};

#endif

**Flight.cpp**

#include <string>

#include "Vector.hpp"

#include "Utils.h"

#include "Flight.h"

#include "Table.h"

using namespace std;

vector<vector<string>> Flight::serialize() {

return serializeEntries(Table::TFlights, flights);

}

vector<vector<string>> Flight::find(string query) {

return findEntries(query, flights);

}

vector<vector<string>> Flight::update(string query, vector<string> fields) {

return updateEntries(query, fields, flights);

}

void Flight::remove(string query) {

removeEntries(query, flights);

}

void Flight::updateFS() {

vector<vector<string>> data = serialize();

Table::TFlights.writeFile(data);

}

void Flight::factory(vector<string> rawData, bool fsSync) {

entriesFactory(flights, rawData, fsSync);

}

void Flight::setFields (vector<string> rawData) {

if(Utils::isRawItemValid(rawData, 0)) flightID = rawData[0];

if(Utils::isRawItemValid(rawData, 1)) departureAirpt = rawData[1];

if(Utils::isRawItemValid(rawData, 2)) arrivalAirpt = rawData[2];

if(Utils::isRawItemValid(rawData, 3)) departureDate = rawData[3];

if(Utils::isRawItemValid(rawData, 4)) departureTime = rawData[4];

if(Utils::isRawItemValid(rawData, 5)) arrivalDate = rawData[5];

if(Utils::isRawItemValid(rawData, 6)) arrivalTime = rawData[6];

if(Utils::isRawItemValid(rawData, 7)) planeID = rawData[7];

}

vector<string> Flight::serializeLn() {

vector<string> record;

record.push\_back(flightID);

record.push\_back(departureAirpt);

record.push\_back(arrivalAirpt);

record.push\_back(departureDate);

record.push\_back(departureTime);

record.push\_back(arrivalDate);

record.push\_back(arrivalTime);

record.push\_back(planeID);

return record;

}

Flight::Flight(vector<string> rawData) {

setFields(rawData);

}

Flight::Flight(){};

**Main.cpp**

#include <iostream>

#include <string>

#include <typeinfo>

#include "Cmd.h"

#include "Table.h"

#include "Flight.h"

#include "Passanger.h"

#include "Plane.h"

#include "Vector.hpp"

using namespace std;

vector<Flight\*> Flight::flights = {};

vector<Passanger\*> Passanger::passangers = {};

vector<Plane\*> Plane::planes = {};

Table Table::TFlights = Table("Flights", Flight::serialize, Flight::factory);

Table Table::TPassangers = Table("Passangers", Passanger::serialize, Passanger::factory);

Table Table::TPlanes = Table("Planes", Plane::serialize, Plane::factory);

int main() {

cout << endl << "Type `help` for help." << endl << endl;

while(true) {

cout << "$> ";

string cmd;

getline(cin, cmd);

if(cmd.length() == 0) continue;

if(cmd == "exit") break;

new Cmd(cmd);

}

cout << endl;

return 0;

}

**Passanger.h**

#ifndef PASSANGER\_H

#define PASSANGER\_H

#include <string>

#include "Vector.hpp"

#include "Entry.hpp"

using namespace std;

class Passanger : protected Entry {

private:

static vector<Passanger\*> passangers;

string passangerID;

string lastName;

string firstName;

string seat;

string flightID;

public:

static vector<vector<string>> serialize();

static void updateFS();

static vector<vector<string>> find(string);

static vector<vector<string>> update(string, vector<string>);

static void remove(string);

static void factory(vector<string>, bool);

vector<string> serializeLn();

void setFields(vector<string>);

Passanger(vector<string>);

Passanger();

};

#endif

**Passanger.cpp**

#include <iostream>

#include <string>

#include "Vector.hpp"

#include "Utils.h"

#include "Passanger.h"

#include "Table.h"

using namespace std;

vector<vector<string>> Passanger::serialize() {

return serializeEntries(Table::TPassangers, passangers);

}

vector<vector<string>> Passanger::find(string query) {

return findEntries(query, passangers);

}

vector<vector<string>> Passanger::update(string query, vector<string> fields) {

return updateEntries(query, fields, passangers);

}

void Passanger::remove(string query) {

removeEntries(query, passangers);

}

void Passanger::updateFS() {

vector<vector<string>> data = serialize();

Table::TPassangers.writeFile(data);

}

void Passanger::factory(vector<string> rawData, bool fsSync) {

entriesFactory(passangers, rawData, fsSync);

}

vector<string> Passanger::serializeLn() {

vector<string> result;

result.push\_back(passangerID);

result.push\_back(lastName);

result.push\_back(firstName);

result.push\_back(seat);

result.push\_back(flightID);

return result;

}

void Passanger::setFields (vector<string> rawData) {

if(Utils::isRawItemValid(rawData, 0)) passangerID = rawData[0];

if(Utils::isRawItemValid(rawData, 1)) lastName = rawData[1];

if(Utils::isRawItemValid(rawData, 2)) firstName = rawData[2];

if(Utils::isRawItemValid(rawData, 3)) seat = rawData[3];

if(Utils::isRawItemValid(rawData, 4)) flightID = rawData[3];

}

Passanger::Passanger(vector<string> rawData) {

setFields(rawData);

}

Passanger::Passanger(){}

**Plane.h**

#ifndef PLANE\_H

#define PLANE\_H

#include <string>

#include "Vector.hpp"

#include "Entry.hpp"

using namespace std;

class Plane : protected Entry {

private:

static vector<Plane\*> planes;

string planeID;

string planeName;

string maxPassangersCount;

string maxSpeed;

string fuelCapacity;

string minRWYLength;

public:

static vector<vector<string>> serialize();

static void updateFS();

static vector<vector<string>> find(string);

static vector<vector<string>> update(string, vector<string>);

static void remove(string);

static void factory(vector<string>, bool);

vector<string> serializeLn();

void setFields(vector<string>);

Plane(vector<string>);

Plane();

};

#endif

**Plane.cpp**

#include <iostream>

#include <string>

#include "Vector.hpp"

#include "Utils.h"

#include "Plane.h"

#include "Table.h"

using namespace std;

vector<vector<string>> Plane::serialize() {

return serializeEntries(Table::TPlanes, planes);

}

vector<vector<string>> Plane::find(string query) {

return findEntries(query, planes);

}

vector<vector<string>> Plane::update(string query, vector<string> fields) {

return updateEntries(query, fields, planes);

}

void Plane::remove(string query) {

removeEntries(query, planes);

}

void Plane::updateFS() {

vector<vector<string>> data = serialize();

Table::TPlanes.writeFile(data);

}

void Plane::factory(vector<string> rawData, bool fsSync) {

entriesFactory(planes, rawData, fsSync);

}

vector<string> Plane::serializeLn() {

vector<string> result;

result.push\_back(planeID);

result.push\_back(planeName);

result.push\_back(maxPassangersCount);

result.push\_back(maxSpeed);

result.push\_back(fuelCapacity);

result.push\_back(minRWYLength);

return result;

}

void Plane::setFields (vector<string> rawData) {

if(Utils::isRawItemValid(rawData, 0)) planeID = rawData[0];

if(Utils::isRawItemValid(rawData, 1)) planeName = rawData[1];

if(Utils::isRawItemValid(rawData, 2)) maxPassangersCount = rawData[2];

if(Utils::isRawItemValid(rawData, 3)) maxSpeed = rawData[3];

if(Utils::isRawItemValid(rawData, 4)) fuelCapacity = rawData[4];

if(Utils::isRawItemValid(rawData, 5)) minRWYLength = rawData[5];

}

Plane::Plane(vector<string> rawData) {

setFields(rawData);

}

Plane::Plane(){}

**Renderer.h**

#ifndef RENDERER\_H

#define RENDERER\_H

#include <string>

#include "Vector.hpp"

using namespace std;

class Renderer {

private:

vector<vector<string>> table;

int colsCount=0;

int maxColWidth=0;

int tableWidth=0;

string padCenter(int, const string&);

string getRowSeparator(bool);

void analyse();

public:

void render();

Renderer(vector<vector<string>>);

Renderer(vector<string>, vector<vector<string>>);

};

#endif

#include <iostream>

#include "Vector.hpp"

#include <string>

#include "Renderer.h"

using namespace std;

vector<vector<string>> table = {};

string Renderer::padCenter(int width, const string& str) {

int len = str.length();

if(width < len) return str;

int diff = width - len;

int padL = diff/2;

int padR = diff - padL;

return string(padL, ' ') + str + string(padR, ' ');

}

void Renderer::analyse() {

maxColWidth = table[0][0].length();

colsCount = table[0].size();

for(int i=0; i<table.size(); i++) {

if(table[i].size() > colsCount) colsCount = table[i].size();

for(int j=0; j<table[i].size(); j++) {

if(table[i][j].length() > maxColWidth) maxColWidth = table[i][j].length();

}

}

maxColWidth += 2; //To allow printing 2 spaces around longest line

tableWidth = colsCount \* maxColWidth + colsCount + 1;

}

string Renderer::getRowSeparator(bool outermostRow) {

char cornerChar = outermostRow ? '+' : '|';

return cornerChar + string(tableWidth - 2, '-') + cornerChar;

}

void Renderer::render() {

cout << getRowSeparator(true);

cout << endl;

for(int i=0; i<table.size(); i++) {

cout << '|';

for(int j=0; j<table[i].size(); j++) {

cout << padCenter(maxColWidth, table[i][j]) << '|';

}

bool lastRow = i == table.size()-1;

cout << endl;

cout << getRowSeparator(lastRow);

cout << endl;

}

}

Renderer::Renderer(vector<vector<string>> \_table) {

table = \_table;

analyse();

}

Renderer::Renderer(vector<string> headers, vector<vector<string>> data) {

vector<vector<string>> \_table;

\_table.push\_back(headers);

for(int i=0; i<data.size(); i++) \_table.push\_back(data[i]);

// \_table.insert(\_table.end(), data.begin(), data.end());

table = \_table;

analyse();

}

**Table.h**

#ifndef TABLE\_H

#define TABLE\_H

#include <string>

#include "Vector.hpp"

#include "FS.h"

using namespace std;

typedef vector<vector<string>> (\*Serializer)();

class Table : public FS {

private:

Serializer serializer;

vector<string> headers;

public:

static Table TFlights;

static Table TPassangers;

static Table TPlanes;

void parse(vector<string>, void (\*)(vector<string>, bool));

void render();

vector<string> getHeaders();

Table(string, vector<vector<string>>(\*)(), void(\*)(vector<string>, bool));

};

#endif

**Table.cpp**

#include <iostream>

#include <sstream>

#include <fstream>

#include <string>

#include "Vector.hpp"

#include "Table.h"

#include "Renderer.h"

using namespace std;

void Table::parse(vector<string> file, void (\*factoryCB)(vector<string>, bool)) {

for(int i=0; i<file.size(); i++) {

vector<string> items;

string item;

istringstream iss(file[i]);

while(getline(iss, item, '\t')) items.push\_back(item);

if(!i) headers = items;

else factoryCB(items, false);

}

}

void Table::render() {

Renderer table(serializer());

table.render();

}

vector<string> Table::getHeaders() {

return headers;

}

Table::Table(string \_tableName, vector<vector<string>> (\*\_serializer)(), void (\*\_factory)(vector<string>, bool)) {

tableName = \_tableName;

serializer = \_serializer;

parse(readFile(), \_factory);

}

**Utils.h**

#ifndef UTILS\_H

#define UTILS\_H

#include "Vector.hpp"

#include <string>

using namespace std;

class Utils {

public:

static string enterPrompt;

static string trim(const string&);

static string replaceChar(string, char, char);

static string slice(string, int, int);

static bool isNumber(const string&);

static bool contains(string, string);

static bool contains(string, char);

static int strToInt(string);

static vector<string> strSplitBySpace(string);

static bool isRawItemValid(vector<string>&, int);

};

#endif

**Utils.cpp**

#include <iostream>

#include <sstream>

#include <string>

#include "Vector.hpp"

#include "Utils.h"

using namespace std;

string Utils::enterPrompt = "Please, enter ";

string Utils::trim(const string& str) {

size\_t first = str.find\_first\_not\_of(' ');

if (string::npos == first) return str;

size\_t last = str.find\_last\_not\_of(' ');

return str.substr(first, (last - first + 1));

}

string Utils::replaceChar(string str, char from, char to) {

for(string::iterator it = str.begin(); it != str.end(); it++)

if(\*it == from) \*it = to;

return str;

}

bool Utils::isNumber(const string& s) {

string::const\_iterator it = s.begin();

while (it != s.end() && isdigit(\*it)) ++it;

return !s.empty() && it == s.end();

}

bool Utils::contains(string str, string substr) {

return str.find(substr) != string::npos;

}

bool Utils::contains(string str, char ch) {

return str.find(ch) != string::npos;

}

string Utils::slice(string str, int start, int end) {

return str.substr(start, end-start);

}

int Utils::strToInt(string str) {

if(isNumber(str)) return stol(str);

return -1;

}

vector<string> Utils::strSplitBySpace(string str) {

istringstream iss(str);

vector<string> words;

string word;

while(getline(iss, word, ' ')) words.push\_back(word);

return words;

}

bool Utils::isRawItemValid(vector<string>& rawData, int i) {

return rawData.isValidIndex(i) && rawData[i].length();

}

**Vector.hpp**

#ifndef VECTOR\_H

#define VECTOR\_H

#include <iostream>

#include <typeinfo>

using namespace std;

template<typename T> class vector {

private:

T\* arr;

int curr\_size;

void initArray (int);

public:

vector(int);

vector();

void push\_back(T);

void erase(int);

int size();

int isValidIndex(int);

T& back();

T& operator[](int);

};

template<typename T> void vector<T>::initArray(int size) {

arr = new T[size];

}

template<typename T> int vector<T>::isValidIndex(int index) {

return index >= 0 && index < curr\_size;

}

template<typename T> vector<T>::vector(int size) {

curr\_size = size;

initArray(size);

}

template<typename T> vector<T>::vector() {

curr\_size = 0;

initArray(0);

}

template<typename T> void vector<T>::push\_back(T value) {

T\* tmp = new T[curr\_size + 1];

for(int i=0; i<curr\_size; i++) tmp[i] = arr[i];

tmp[curr\_size] = value;

delete[] arr;

arr = tmp;

curr\_size++;

}

template<typename T> void vector<T>::erase(int n) {

if(!isValidIndex(n)) return;

T\* tmp = new T[curr\_size];

for(int i=0; i<curr\_size-1; i++) tmp[i] = i < n ? arr[i] : arr[i+1];

arr = tmp;

curr\_size--;

}

template<typename T> T& vector<T>::operator[](int i) {

if(!isValidIndex(i)) return arr[0];

return arr[i];

}

template<typename T> int vector<T>::size() {

return curr\_size;

}

template<typename T> T& vector<T>::back() {

return arr[curr\_size-1];

}

#endif

**build.sh**

#!/bin/bash

g++ -std=c++11 \*.cpp -o ./output && ./output