2018/2019 н.р.
Лабораторна робота 3

Структури даних та файли, або Text = 10 балів

Передмова

Мета

Метою лабораторної є побудова власних систем класів, робота з структурами даних на початковому рівні, обробка командного рядка, елементарна робота з текстовими файлами.

Зрозуміло, що цю задачу можна розв'язати засобами баз даних. Зрозуміло, що можна побудувати більш ефективне за пам'яттю та часом роботи розв'язання. Але не завжди більш ефективний розв'язок буває економічно доцільним з точки зору вартості/часу його розробки. Семестр також має цілком конкретні межі. Тому вимог щодо ефективності обробки тут майже не ставиться.

Лабораторна робота розбита на кілька етапів та логічних частин. Планується автоматичне тестування (частин, етапів та в цілому) та **автоматична перевірка на запозичені фрагменти коду**.

Умова дуже коротко

На вхід подається текстовий файл з інформацією. Інформацію необхідно завантажити в пам'ять, перевірити коректність вхідного файлу. Далі необхідно записати інформацію з пам'яті в текстовий файл, а також надати певну статистику за отриманими даними.

Приклад

Файл містить інформацію щодо складання студентами цієї лабораторної роботи (студент, порядковий номер листа з лабораторною та кількість балів за спробу). В якості статистики треба знайти найбільш працьовитих студентів (що зробили найбільшу кількість спроб здати).

Загальна структура вхідного файлу

Вхідна інформація подається в текстовому файлі. Вхідний текстовий файл має вигляд (з точністю до «білих» рядків):

перший рядок – заголовок (header)

далі йдуть рядки з основною інформацією – «записи»

останній рядок – нижній колонтитул (footer).

Поля та роздільники

Рядки діляться на частини — «поля», — за допомогою роздільників. Частина рядка між сусідніми роздільниками, або між початком рядка та першим роздільником, або між останнім роздільником та кінцем рядка задає вміст поля. Рядок, що не містить роздільників, складається з єдиного поля. Роздільники до полів не входять. Білі символи на початку та в кінці полів мають ігноруватися (отже, до поля не входять). Порожні поля теоретично можливі (Наприклад, якщо між сусідніми роздільниками нема нічого крім білих символів. Або, нехтуючи білими символами, якщо рядок починається чи закінчується роздільником.). Різні рядки файлу можуть мати різну кількість полів.

Використовувані роздільники визначаються варіантом. В одному файлі (і навіть в одному рядку) можуть використовуватися різні роздільники з множини роздільників, що задана у варіанті.

Білі рядки

«Білими» вважаються порожні рядки та рядки з білих символів.

«Білі» рядки можуть зустрічатися у файлі в довільній кількості та в довільному місці; під час обробки мають ігноруватися.

Останній рядок текстового файлу може не містити символ кінця рядка.

Послідовність інформації

Заголовок має бути рівно один, нижній колонтитул має бути рівно один. Записів може бути скільки завгодно (зокрема, жодного). Перед заголовком не може бути ані записів, ані нижнього колонтитулу. Після нижнього колонтитулу не може бути ані заголовку, ані записів.

Заголовок

Першим полем заголовку є послідовність символів **`header`** , решта полів містять вміст заголовку згідно варіанту.

Записи

Першим полем записів (рядків з основною інформацією) є порядковий номер рядка (наскрізна нумерація з 1 у порядку розташування у файлі, білі рядки ігноруються), решта полів містять інформацію, що задається варіантом.

Нижній колонтитул

Першим полем нижнього колонтитулу є послідовність символів **`footer`** , решта полів містять вміст нижнього колонтитулу згідно варіанту.

Приклад вхідного текстового файлу

Припустимо, що роздільником є символ ;

header; поле1 заголовку ; поле 2 заголовку;поле 3 заголовку

1; поле 1;поле 2; поле 3; поле 4

2; поле 1;поле 2; поле 3; поле 4; поле 5

3; поле 1;поле 2
footer; поле 1 колонтитулу

Вимоги

1 Програма повинна вміти

B1) завантажувати вміст текстового файлу з інформацією в пам'ять та перевіряти при цьому наявність синтаксичних та семантичних помилок у вхідних даних;

B2) вивантажувати інформацію в текстовий файл (порядок записів не регламентується), команда **output**;

B3) знаходити та виводити певну статистику (згідно варіанту), команда **stat**.

(**×0**) Вхідний файл має читатися не більше одного разу. У B2 вивантаження інформації має відбуватися в тому самому форматі, в якому дані завантажуються. Якщо варіантом передбачено кілька можливих роздільників для вхідної інформації, то для виведення можна використовувати довільний з них. Банально переписувати вхідний файл на вихід заборонено. У B3 роздільниками інформації, що записується на одному рядку, мають бути табуляції. Символ табуляції в кінці рядка є неприпустимим. Оскільки планується автоматичне тестування, то нічого непередбаченого умовою виводитися не повинно.

2 (×0) Командний рядок програми

Уся інформація надходить в програму з рядка виклику (=командного рядка).

Програма має обробляти командний рядок такого вигляду.

<виконуваний файл> <ім'я вхідного файлу> {–<команда> <ім'я вихідного файлу>}

<команда> - одне з **output**, **stat**

Якщо в якості імені вихідного файлу вказано **#con**, то виведення має йти на консоль.

Фігурні дужки означають, що відповідна частина командного рядка може зустрічатися довільну кількість разів (зокрема, жодної).

Наприклад:

prog.exe in.txt –output out.txt –output out2.txt –sort s.txt –output out1.txt –stat #con

3 (×0) Консольний вихід програми

Увага! Послідовність символів ***** має для тестера особливе значення.

Виконання починається, як зазвичай, з виведення інформації про виконавця та умову. Далі програма виводить стислу довідку по аргументах свого командного рядка.

Далі на окремому рядку програма виводить

*****

Після чого починає завантажувати дані з вхідного файлу. Перед початком завантаження програма виводить

input <ім'я вхідного файлу> :

далі починає завантаження. За успішного завантаження далі програма виводить

OK

У результаті маємо бачити

input <ім'я вхідного файлу> : OK

За наявності помилок програма виводить **UPS**, у результаті маємо бачити

input <ім'я вхідного файлу> : UPS

далі з нового рядка виводить діагностичне повідомлення про помилку, з нового рядка ***** і завершує роботу. Вимоги до діагностичного повідомлення див. у підрозділі Обробка помилок.

За успішного завантаження програма продовжує роботу і далі виводить рядки з результатами виконання кожної запитуваної команди у форматі

<команда> <ім'я вихідного файлу> : <результат>

<результат> - одне з

OK (якщо вдало і виведення в текстовий файл),

UPS (якщо невдало і виведення в текстовий файл),

за виведення на консоль виводить з нового рядка на консоль очікувану інформацію.

Після завершення обробки всього командного рядка програма виводить на окремому рядку

*****

Невідомі команди мають ігноруватися з повідомленням **ignored** (див. приклад нижче). Наприклад, якщо за командного рядка

Prog.exe in.txt –output out.txt qwerty –output out2.txt –stat s.txt –output out1.txt –stat #con
вдалося виконати все, крім запису у файл *out2.txt*, то має бути виведено
*****
input in.txt : OK
output out.txt : OK
qwerty : ignored
output out2.txt : UPS
stat s.txt : OK
output out1.txt : OK
stat #con :
якась інформація
*****
Якщо в командному рядку наявне тільки ім'я виконуваного файлу, то має бути виведено
*****
nothing to do
*****
Якщо в командному рядку не вистачає параметрів, то неповна команда має ігноруватися з повідомленням (див. приклад нижче). Наприклад, за командного рядка
Prog.exe in.txt –output
має бути виведено
*****
input in.txt : OK
output undefined
*****

Якщо якесь виведення у файл виконалося невдало, то виконання програми не завершується. Натомість виконується виконання наступної команди (за наявності).

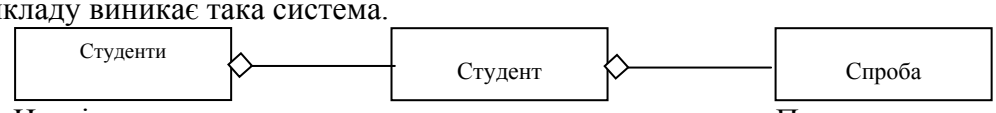
Довідка з рядку виклику програми не повинна виводитися між маркерами *****.

Порушення вимог до виходу програми призведе до того, що під час тестування тестер вирішить, що програма зовсім не відповідає вимогам. Кількість проміжків після команди, біля двокрапки та біля результату обробки значення не має.

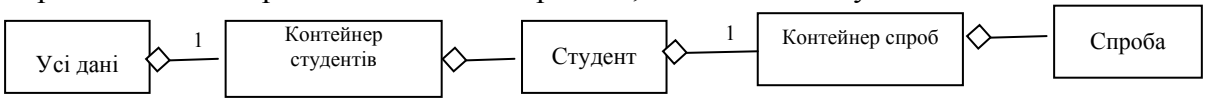
4 (×0) Структури даних для збереження інформації

Варіанти лабораторних робіт побудовано так, що інформацію можна зобразити у вигляді системи класів. Для розв'язання задачі необхідно побудувати структуру даних (контейнер), що буде зберігати інформацію з файлу. Якщо розв'язувати задачу з прикладу з точки зору ООП (а не баз даних) одиницею зберігання в структурі даних верхнього рівня має бути студент. Отже, контейнер верхнього рівня має зберігати студентів – сутності . Відповідно, контейнер *Студенти* зберігає об'єкти класу *Студент*. Для кожного студента треба знати його спроби здати лабораторну роботу. Тоді об'єкти класу *Студент* мають зберігати інформацію про спроби – деталі сутностей. Тоді контейнер *Спроби* має бути частиною *Студента* та зберігати об'єкти класу *Спроба*(=Деталі).

У результаті отримуємо дворівневу систему контейнерів. Контейнер верхнього рівня зберігає інформацію про сутності, кожна сутність має контейнер з інформацією про власні деталі. Стосовно прикладу виникає така система.

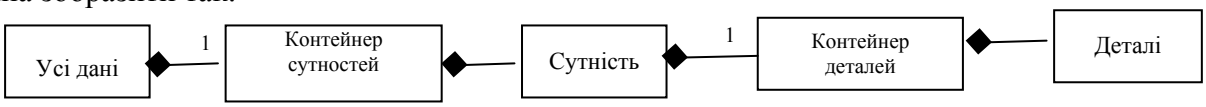


Ця діаграма описує структуру даних концептуально. При цьому не стверджується, що має бути рівно три класи для збереження даних. Наприклад, можливе й так уточнення.

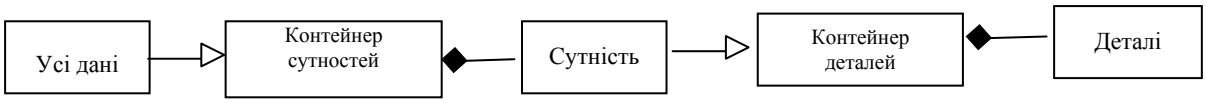


Також деякі агрегування на цій діаграмі можуть виявитися насправді композиціями. А деякі інші можуть бути замінені на успадкування. Крім того, у деяких варіантах буде доцільним мати допоміжні класи.

Якщо відійти від конкретного прикладу, то концептуально діаграму класів, що зберігають дані, можна зобразити так.



Або так .



У більшості варіантів сутностями є те, чого стосується статистика.

а) Реалізовано систему контейнерів, що містять інформацію, надану у файлі. Діаграма класів, що зберігають дані, концептуально відповідає наведеній вище діаграмі (довільний з двох останніх). Можуть бути присутні додаткові класи, але повинна зберігатися дворівнева система контейнерів. Клас «Усі дані», що містить усю отриману інформацію, має ім'я **Info**. Після завантаження всі завантажені дані зберігаються в його об'єкті.

За бажання зробити більш економну за використанням пам'яті структуру класів з використанням допоміжних класів, які в базах даних грають роль класифікаторів, можливі порушення деяких подальших вимог та вимог до етапів. Для кожного класифікатора мають бути виконані вимоги п. б)-ж) цього підрозділу. Для уникнення можливих непорозумінь під час оцінювання відповідна діаграма класів має бути представлена на узгодження не пізніше 15 березня.

б) Обмеження контейнерів на розмір вхідних даних залежать тільки від поточного доступного обсягу вільної пам'яті.

в) Внутрішнє зображення інформації якомога більше приховано від користувача. Зокрема, клас *Деталі* має бути прихованим або захищеним членом класу *Сутність* (або **Info**, якщо *Сутність* успадковується від *КонтейнераДеталей*).

г) У класах **Info** та *Сутність* наявні відкриті методи, що додають до контейнерів та їх елементів інформацію приведену до відповідних скалярних або рядкових типів. Рядкові типи для числових або логічних полів не використовуються.

Наприклад, *додатиСтудента (П, І, Б)*, *додатиСтудентуЛабораторну(П, І, Б, номер, кількістьБалів)*. Номер, кількість балів повинні задаватися значеннями числових типів; прізвище повинно задаватися як рядок.

Зверніть увагу: вивантажений файл має успішно завантажуватися навіть якщо в об'єкт **Info** не завантажувався файл, а було зроблено якусь іншу низку викликів його методів. У деяких варіантах він має зберігати та підтримувати в актуальному стані деяку інформацію, що далі виводиться в заголовку та/або нижньому колонтитулі.

д) Усі відкриті методи, до яких є доступ у клієнтів класів **Info** та *Сутність*, в якості аргументів повинні приймати дані скалярних типів або рядки.

Насправді, це випливає з п. в, бо інформація про внутрішній формат збереження інформації для клієнтів недоступна. Такий підхід підвищує гнучкість коду, бо тоді клієнти не залежать від деталей реалізації.

Методи, що приймають об'єкти не забороняються, але вони **не мають бути відкритими**. Якщо вони потрібні назовні, то для них **слід побудувати відповідні обгортки**.

Наприклад, метод *знайти(Студент)* не повинен бути відкритим. Якщо він для чогось потрібний, то відкритим має бути метод *знайти(П,І,Б)*. Останній конструює об'єкт студента та використовує далі прихований або захищений метод *знайти(Студент)*.

е) У жодному класі, призначеному для збереження даних, не повинно бути методів, що додають інформацію, що задається як вміст усього файлу, або вхідний потік, або нерозібрана частина тексту, або якийсь інший контейнер з елементами інформації (наприклад, з полями).

ж) Для класів *Сутність* та *Деталі* (та деяких інших, якщо це явно зазначено в умові варіанту) перевантажено оператори ==, !=, <, <=, >, >=. Означення цих операторів не конфліктують між собою (у математичному сенсі).

5 (×0) Зображення вхідних даних відділено від структур даних

а) Класи контейнерів, що зберігають інформацію, нічого не знають про зовнішній формат збереження даних у текстовому файлі.

б) Розбір тексту виконується **за один прохід та за ледачою стратегією**. Енергійна стратегія не використовується.

Ледача стратегія передбачає, що поки поточний рядок текстового файлу не розібрано та інформацію з нього не завантажено в результуючий контейнер, наступний рядок не читається. Оброблені рядки (як рядки) в пам'яті не зберігаються.

Припустимо, у рядку міститься інформація щодо результатів тестування: назва тесту та результат проходження:

тест1; ОК; тест2;ОК; тест3; UPS

Ледача стратегія також передбачає, що під час обробки рядка результати тестування виділяються з рядка по черзі за запитом і записуються в результуючий контейнер (поки не записано поточний результат тестування наступний не виділяється; тобто прочитали, записали, прочитали, записали,…).

Енергійна стратегія передбачає, що спочатку все прочитали, все розбили на частини (зберегли в якійсь додатковій структурі даних), а потім думаємо як це все записати в результуючий контейнер. Енергійна стратегія заборонена для використання.

в) Логіку завантаження вхідного файлу в контейнер виділено в окремий клас **Builder**, що використовуючи об'єкт класу **Lexer**, реалізує завантаження вмісту текстового файлу в контейнер. У класі **Builder** наявний відкритий метод, що завантажує вміст файлу в контейнер.

Клас **Lexer** реалізує лексичний розбір рядка. Він відповідає за отримання послідовності полів рядка, а саме за виділення полів рядка, можливо, що без урахування формату (число, рядок, дата, тощо). Об'єкт класу **Lexer** отримує рядок для розбору. Далі за кожним наступним запитом він видає клієнту вміст наступного поля рядка.

г) Наявна функція **loadData(Info&, const char*)**, що виконує завантаження файлу в контейнер «під ключ». Для завантаження використовується об'єкт класу **Builder**. Використання переваг **friend** під час завантаження даних в контейнер заборонено. Завантаження реалізовано через відкритий інтерфейс класів.

д) Коректний вхідний файл має правильно розбиратися та завантажуватися в контейнер. Якщо в контейнері щось було, то його вміст на початку завантаження скидається. У випадку наявності помилок контейнер має залишатися порожнім.

6 (F) Отримання вихідної інформації

а) Обхід контейнерів під час збирання/виведення **вихідної** інформації здійснюється виключно за допомогою ітераторів.

б) (**×0**) Контейнери нічого не знають про вихідний формат.

Наслідок: функції, що взаємодіють з вихідними потоками, реалізовано виключно за межами класів, що зберігають дані.

в) Перевірка помилок виведення виконується.

7 (×0) Performance

Квадратична оцінка часу роботи під час завантаження даних в пам'ять є допустимою.

Як тільки дані завантажено (функція **loadData** завершила роботу), решта операцій (за винятком сортувань) мають виконуватися за лінійний час.

У пам'яті зберігається тільки одна копія вхідних даних. Часткові копії неконстантного розміру також зберігати не можна.

Якщо у варіанті явно не вказано інше, запитувана статистика в окремих допоміжних структурах даних не зберігається, а виводиться «зльоту».

(**F**) Явних неоптимальностей, яких можна легко уникнути, бути не повинно. Наприклад, під час завантаження одно вхідного файлу за одним тим самим константним С-рядком багаторазово утворюється рядок типу string.

8 Завантаження вхідного некоректного файлу та перевірка помилок

а) Завантаження файлу виконується до першої помилки.

б) Під час завантаження мають перевірятися помилки вводу/виводу.

в) Під час завантаження мають перевірятися всі можливі синтаксичні та семантичні помилки (некоректні зображення чисел, невідповідність формату, неправильна кількість полів у рядку, неправильна нумерація записів, неправильні контрольні суми, тощо).

Відповідальність за перевірку помилок розподіляється між класом **Builder** та класами, що безпосередньо зберігають дані. Клас **Lexer** помилки не перевіряє (якщо він виконує більш повноцінний лексичний аналіз і підтримує виділення лексем різних видів (має бути узгоджено заздалегідь), то цей момент також узгоджується окремо).

Клас **Builder** відповідає за конвертацію рядкових значень до числових з урахуванням типу (але не специфічних обмежень на дані, що визначаються умовою варіанту; може, навіть повинен, використовувати функції стандартної бібліотеки), за перевірку нумерації записів, за перевірку послідовності заголовок-записи-нижній_колонтитул, за перевірку відповідності заголовку та нижнього колонтитулу вмісту файлу, за перевірку кількості полів у рядку.

За перевірку обмежень, що накладаються на дані в умові варіанту (наприклад, кількість балів, отримана за лабораторну має бути в межах від 0 до 10), відповідає клас, що безпосередньо буде зберігати ці дані (у прикладі за кількість балів відповідає клас Спроба). Ці помилки мають виникати під час спроби записати інформацію в контейнер.

г) За наявності помилок у вхідному файлі на консоль виводиться діагностичне повідомлення, яке повідомляє код помилки, номер рядка текстового файлу, на якому сталася помилка (рядки нумеруються з 0, порожні та білі враховуються), та суть помилки. Якщо помилка стосується відповідності вмісту файлу в цілому заголовку та нижньому колонтитулу, то в якості номера рядка вказувати -1.

Символ кінця рядка всередині діагностичного повідомлення не використовувати! Код помилки та номер рядка відділяти виключно проміжками!

Мають підтримуватися такі і тільки такі коди помилок:

100 відсутній заголовок
101 є більше одного заголовку
102 заголовок не відповідає вмісту файлу
103 невідповідна кількість полів у заголовку
104 інші помилки в заголовку

200 відсутній нижній колонтитул
201 є більше одного нижнього колонтитулу
202 нижній колонтитул не відповідає вмісту файлу
203 невідповідна кількість полів у нижньому колонтитулі
204 інші помилки в нижньому колонтитулі

300 порушено порядок заголовки- записи-нижній колонтитул
301 неправильна нумерація записів
302 невідповідна кількість полів у рядку
303 інші помилки у рядку

400 помилка роботи з фізичним файлом
500 bad_alloc

Якщо помилку можна класифікувати по-різному, то віддавати перевагу коду з найменшим номером.

Наприклад, на консоль може бути виведено таке
301 13 incorrect line number

Вся обробка помилок під час завантаження даних має бути реалізована через механізм винятків. Усі використовувані типи винятків мають бути похідними від класу **exception**.

9 (F) Інше

M1. У заголовних файлах прототипи функцій, методів належно задокументовані. Призначення класів також наявні. Функції, оголошення яких не винесено в заголовні файли, також належно задокументовані (у точці, де вперше оголошуються).

M2. У власних заголовних файлах нема нічого зайвого. Вони не розкривають подробиці реалізації оголошених у них функцій.

M3. Коментарі не повинні бути надлишковими (особливо у cpp-файлах).

М4. (**×0**) Коментарі мають на 100% відповідати коду, який коментують.

М5. **(×0)** Ситуації «*control reaches end of non-void function*», «*no return statement in function returning non-void*» виникати не повинns.

Таке буває, коли виклик функції, що повертає не void, завершується у звичайний спосіб (тобто не через появу винятку), але без повернення значення. У такому випадку результатом функції є сміття. Якщо повезе, то «правильне», якщо не повезе, то ні і далі код може вже працювати не так, як заплановано розробником.

М6. Код не містить «магічних» констант та власних глобальних змінних (та макросів, що іменують константи або мають параметри).

М7. Код має бути структурованим (за деякими виключеннями, пов'язаними з використанням винятків). **(×0)** Використання у власному коді **exit** та **goto** заборонено.

М8. Слід притримуватися функціонального проектування: один фрагмент_коду/ функція – один обов'язок; класи також не повинні мати різнопланових обов'язків; аналогічне стосується одиниць трансляції – зібрані в них частини повинні мати щось логічно спільне, наприклад, спільно виконують певний обов'язок, але обов'язків не повинно бути багато та різних. (Крайнощів вигляду: одна функція – одна одиниця трансляції, теж слід уникати; якщо щось можна логічно об'єднати, то це слід робити.)

М9. Функції/методи не повинні бути великими. Якщо функція стає великою, то майже завжди її можна сформулювати в термінах менших функцій. Жоден клас не повинен мати різнопланові обов'язки. Якщо клас стає великим, частину його логіки слід виділяти в інший клас.

М10. Код не повинен містити дублювання (у тому числі файлів та класів) та невикористовувані фрагменти (навіть закоментовані). Дублювання коментарів (наприклад, коментується оголошення в заголовному файлі, а потім цей коментар дублюється біля означення) також неприпустиме. Замість схожих функцій/методів варто розглянути можливість винесення спільної частини в допоміжну функцію/метод. Замість схожих за внутрішнім устроєм контейнерів варто розглянути можливість використання шаблонів контейнерів (або реалізувати принципово різні контейнери).

Однакова логіка обробки команд -output, -stat також є дублюванням. Якщо вона призведе до дублювання тільки дуже коротких високорівневих функцій, то як помилка розглядатися не буде.

М11. Класи мають відповідати принципу інкапсуляції та підтримувати цілісність даних. Стиль програмування має відповідати принципам ООП.

Код має захищати свої дані від несанкціонованих змін.

Відкриті методи не повинні дозволяти зіпсувати дані та передбачати наявність перевірок перед викликом.

Не слід використовувати без потреби посилання та вказівники в якості параметрів. Якщо параметр технічно передається функції адресою/посиланням (й така передача є доцільною та/або єдиною синтаксично можливою) і якщо призначення функції не передбачає змін цього параметру, то треба блокувати зміни такого параметра на рівні компіляції (користуємось const).

Методи розроблених класів реалізовано за межами означень класів.

М12. **Робота з ресурсами**. При роботі з динамічними даними усі створені динамічні змінні мають коректно знищуватися під час виконання програми. Робота з пам'яттю ведеться коректно. Чужа пам'ять не використовується ані на запис, ані на читання. Динамічні змінні коректно знищуються.

Наявність/відсутність конструкторів копії/переміщення, операторів присвоювання (із стандартною семантикою), деструкторів не регламентується, але класи мають забезпечувати коректну роботу з пам'яттю і запобігати витокам ресурсів.

М13. **(×0)** Код має відповідати рекомендаціям підрозділу **9.5 Кілька слів про стиль коду та константи true та false** (див. файл **09 Керування порядком обчислень.pdf**)

М14. **(×0)** Інструкції **using namespace** у заголовних файлах записувати заборонено.

М15. **(×0)** Назви файлів з означеннями функцій та відповідних заголовних файлів мають бути однаковими з точністю до розширення.

М16. **(×0)** Включення власних файлів з означеннями функцій НЕ допускається (за винятком файлів з шаблонами).

М17. **(×0)** За необхідності використання **system("pause");** записувати цю інструкцію саме так, як наведено тут.

М18. **(×0)** Використання pragma у власному коді дозволяється тільки у випадку #pragma once

M19. Під час компіляції відсутні попередження від компілятора. (Під час перевірки буде використовуватися компілятор mingw версії не нижче 8.1.0.)

M20. **(×0)** Стиль програмування не повинен бути «C-стиль з класами». Відкриті змінювані члени-дані заборонені. Через специфіку тестера оголошення власних **namespace** заборонено.

10 Використання контейнерів STL

Використання контейнерів STL не забороняється і не вимагається. Кожен вирішує сам для себе використовувати чи ні. **Але**. У випадку використання бібліотечних контейнерів питання співбесіди будуть стосуватися не тільки використаних у коді методів, але й тієї частини загальної специфікації контейнеру згідно діючого стандарту C++, що знаходиться в деякому ε-околі використаних засобів.

11 (×0) Оформлення коду

Вміст файлів з текстом програми та імена файлів записано з використанням символів з кодами від 0 до 127 (за винятком коментарів). Коментарі можна писати українською або англійською. В іменах файлів проміжків бути не повинно.

Для файлів з текстом програми кодування **UTF-8 without signature (without BOM)** не використовується. Можливі кодування:

UTF-8 with signature – codepage 65001
Cyrillic (Windows) – codepage 1251
(наступні два можливі, але з огляду на збільшення розміру файлів, дуже небажані)
Unicode – codepage 1200
Unicode (big-endian) – codepage 1201

Включення бібліотек, відсутніх у стандарті C++17, не допускається. (Зокрема, заборонено включення специфічних для сім'ї ОС Windows бібліотек conio.h та інших.) Усі спірні питання типу «а в мене все компілювалося» будуть вирішуватися за допомогою стандарту C++17 та компілятору. Якщо використані синтаксичні елементи, що не відкомпілювалися під час тестування, не відповідають стандарту, код буде вважатися таким, що не компілюється.

12 (×0) Академічна порядність

Студент вільно орієнтується в коді лабораторної роботи, яку він здає, розуміє використані синтаксичні елементи мови, зміст та призначення частин коду. Розуміння усіх використаних синтаксичних конструкцій та елементів мови має бути 100%. Студент вільно орієнтується в проектах власної лабораторної роботи, вміє відкомпілювати файли, зібрати проекти (у т.ч. виконуваний файл), запустити на виконання.

На початку кожного файлу в коментарях зазначено його автора та наявні запозичення з відкритих загальнодоступних джерел (що і звідки (посилання на ресурс) взято, що перероблено/додано). Також мають бути зазначені частини, які розбиралися на лабораторних заняттях з викладачем.

Використовувати код контейнерів STL заборонено, навіть якщо це коректно зазначено. За спроби видати код контейнерів STL (або його частину) за власноруч розроблений, лабораторна відразу буде забанена, подальші спроби її здати враховані не будуть. Навіть якщо його було взято не з системної бібліотеки, а з якогось інтернет-форуму.

Усі листи з кодом лабораторної відправлено із власної поштової адреси (а не з адреси товариша!). У випадку, коли з однієї адреси відправлено лабораторні різних студентів, всі такі лабораторні будуть оцінені на **0** (**нуль**) балів.

У випадку виявлення запозичень і хто дав свій код, і хто взяв чуже отримують **0** (**нуль**) балів. Якщо кілька осіб використали спільне зовнішнє джерело, то аналогічно.

Деякі частини можуть розглядатися на заняттях. Все ОК. Але 1) вони будуть потребувати хоча б мінімальної доробки; 2) коли програма набирається з однієї дошки (а не копіюється файл), розбіжності завжди існують.

Після завершення приймання лабораторної роботи буде виконана автоматична перевірка на запозичення коду. Можливо, що будуть і проміжні перевірки.

Також хочу нагадати, що остаточно лабораторна буде зарахована тільки за підсумками її захисту.

13 Етапи роботи над кодом

Лабораторна робота розбита на кілька етапів. За кожною частиною встановлено граничний термін. Зміст та вимоги до етапів зазначено в окремій частині цього документу. Етапи розробки передбачають додаткові бали за вчасне та правильне виконання.

14 Нарахування балів

Лабораторна робота допускає часткове виконання. 10 балів за лабораторну розподіляються між її функціональністю за такою схемою:

(**5**) Для коректного вхідного файлу програма правильно виконує дії B1 та B2.

(**1**) Для коректного вхідного файлу програма правильно виконує дії B1 та B2, причому обходи структур даних під час виведення виконуються виключно з використанням ітераторів.

(**2**) Для коректного вхідного файлу програма правильно виконує дії B1 та B2. У випадку некоректного вхідного файлу програма правильно діагностує помилки (виконано **всі** вимоги підрозділу 8 Завантаження вхідного некоректного файлу та перевірка помилок).

(**2**) Для коректного вхідного файлу програма правильно виконує дії B1, B2 та B3.

Позначка (**×0**) означає, що вимога (група вимог) є **критичною**: у випадку порушення лабораторна робота оцінюється в **0** (**нуль**) балів.

Позначка (**F**) означає, що у випадку порушення вимоги (або якоїсь її складової) бали за лабораторну роботу будуть зменшені на 25%: усі набрані бали множаться на 0.75.

Захист лабораторної роботи спочатку відбувається у вигляді письмової роботи. Якщо письмова робота не підтверджує дотримання академічної порядності (див. підрозділ 12 (×0) Академічна порядність) або перевірка коду на запозичення показала порушення академічної порядності, лабораторна робота оцінюється в **0** (**нуль**) балів без проведення співбесіди. Співбесіда за лабораторною роботою відбувається тільки в пограничних ситуаціях. Апеляція можлива, але тільки за тим кодом, що був надісланий у зазначені терміни, і тільки за тією письмовою роботою, яка була написана.

Окремі етапи розробки можуть передбачати бонуси. Якщо для коректного вхідного файлу програма не почала правильно виконувати дії B1 та B2, то бали за етапи та додаткові бонуси згорають.

Якщо не зазначено інше, під наявністю функції/методу/класу/… мається на увазі наявність належно реалізованої функції/…, що відповідає умові лабораторної та вимогам до неї.

У випадку, якщо на якійсь спроб здати лабораторну в цілому або якийсь її етап були зафіксовані порушення вимог підрозділу 12 (×0) Академічна порядність, то лабораторна робота оцінюється в **0** (**нуль**) балів без розгляду подальших спроб.

15 (×0) Організаційні моменти

Не пізніше 14 травня 2019 року на адресу LabAssignment2@i.ua (далі «адреса для лабораторних робіт») надійшов лист з повним кодом лабораторної роботи (усі суттєві для проекту cpp- та h-файли і нічого іншого, як вкладення). У темі листа зазначено, що це лабораторна робота 3 та номер варіанту, у форматі **Lab3, <номер варіанту>**. Наприклад, **Lab3, 66**

У самому листі (із етапом лабораторної, і з лабораторною в цілому) зазначено виконавця та компілятор, що використовувався при виконанні лабораторної роботи. Листи з архівами та посиланнями на інтернет-ресурси не припустимі. Один лист – одна лабораторна робота, повністю.

Дозволяється відправляти код лабораторної/етапу кілька разів (наприклад, якщо було усунуто якісь недоліки). У випадку, коли код лабораторної/етапу надходив кілька разів, розглядатиметься та оцінюється тільки остання отримана версія (навіть, якщо передостання працювала, а до останньої забули додати частину файлів). Дата/час версії визначається за датою надходження на адресу для лабораторних робіт. Лабораторні/етапи, що не надішли вчасно, не розглядаються та не оцінюються.

Етапи лабораторної роботи

Опис етапів

Етап 1 Лексичний розбір рядка (від +1 до +2 балів)	остаточна версія
Етап 2 Обробка командного рядка (+1 бал)	остаточна версія
Етап 3 Класи, що зберігають дані (від +0.5 до +1.5 балів)	остаточна версія
Етап 4 Клас Builder (від +0.5 до +2 балів)	остаточна версія
Етап 5 Контейнери (від +0.5 до +1 балів)	зовсім проект
Етап 6 Обходи контейнерів, отримання вихідної інформації	зовсім проект

Етап 1 Лексичний розбір рядка (від +1 до +2 балів)

Зміст етапу. Розробити та протестувати клас **Lexer**, що відповідає вимогам лабораторної роботи та власного варіанту.

Клас **Lexer** реалізує найпростіший варіант лексичного розбору рядка за ледачею стратегією. Він відповідає за отримання послідовності полів рядка, а саме за виділення полів рядка без урахування формату (число, рядок, дата, тощо). Об'єкт класу **Lexer** отримує рядок для розбору. Далі за кожним наступним запитом він видає клієнту вміст наступного поля рядка.

Інтерфейс класу **Lexer** складається з таких методів:

– метод **load** отримує рядок, який треба розділяти на поля;

– метод **bool next(std::string &field)**, що виділяє чергове поле рядка (з вилученими білими символами на початку та в кінці); поле повертається через **field**, результат – ознака наявності чергового поля (якщо поля закінчилися, то повертається **false**);

– метод **bool eof() const noexcept** повертає істину, якщо рядок вже було розібрано до кінця.

Для класу **Lexer** допустимо (але не вимагається), щоб метод **load** визначав тип рядка (білий, заголовок (перше поле є **header**), нижній колонтитул (перше поле є **footer**), решта) як значення типу **LineType**. У такому разі перші поля рядків з заголовком та нижнім колонтитулом (вони містять **header** та **footer** відповідно) до вихідної послідовності полів не входять.

Тип **LineType** є фіксованим.

enum class LineType {Empty, Footer, Header, Line};

Демо

Написати функцію **void testLexer(const string&),** яка, використовуючи клас **Lexer**, за отриманим рядком виводить послідовність його полів по одному на рядок у квадратних дужках (і більш нічого!).

Написати головну функцію, що за допомогою функції **testLexer** дозволяє користувачу перевірити правильність функціонування класу. Виконати тестування класу **Lexer**.

Якщо реалізовано **LineType**, то для тестування правильності визначення типу рядка можна написати іншу функцію і також запускати з **main**. Тестеру це не заважатиме.

Одиниці трансляції

lexer.cpp та .h – клас **Lexer** (та **LineType**, якщо використовується)

testLexer.cpp та .h – функція `testLexer`

main.cpp – головна функція

Оцінювання етапу

Не пізніше 4 березня 2019 року на адресу LabAssignment2@i.ua надійшов лист з повним кодом етапу (усі суттєві для проекту cpp- та h-файли і нічого іншого, як вкладення). У темі листа зазначено **Lab31, <номер варіанту>**. Наприклад, **Lab31, 66**

У випадку, якщо тестеру все подобається (не тільки як працює код, але й назви файлів, класу та методів) і не порушено критичні вимоги (у частині, що не суперечить опису етапу), то за вчасне виконання етапу дається **1** бал.

Примітка. За попереднім узгодженням клас **Lexer** може виконувати більш повноцінний лексичний аналіз, зокрема виконувати класифікацію полів: зображення цілого, зображення дійсного, зображення беззнакового цілого, зображення рядка, тощо. У такому випадку 1 бал за етап теж буде даватися.

Додаткові плюшки етапу (надаються тільки за вчасного виконання етапу)

Під час оцінювання лабораторної роботи може статися, що у випадку, коли метод **load** визначає тип рядка як значення типу **LineType**, за **ефективну** та **красиву** роботу з рядками (ефективність розуміється як уникнення копіювань масивів, де цього дійсно можна уникнути; явна робота з вказівниками заради ефективності красивою не вважається) будуть даватися додаткові 1-2 бали. Але це за умови, що не буде проблем з пам'яттю, а «нюанси» будуть належно закоментовані і при цьому клас буде правильно використаний в ЛР.

Якщо **Lexer** буде виконувати не тільки виділення, але й класифікацію полів, з використанням regex або механізму FA, то додаткові бали теж можливі.

Якщо «бонусні» розв'язання будуть масово поширені (написав собі, поділився з товаришем), то жодних додаткових балів роздаватися не буде. Запозичених фрагментів у них бути не повинно (навіть з відкритих джерел). Спроба отримати бонуси з використанням чужого коду буде сприйматися вкрай негативно.

Етап 2 Обробка командного рядка (+1 бал)

Зміст етапу. Реалізувати обробку командного рядка згідно вимог лабораторної роботи. У результаті має бути головна функція, що повністю та правильно обробляє командний рядок (з урахуванням можливості «некоректних» команд, нестачі аргументів, тощо).

На даному етапі ще нема жодної структури даних. Не проблема! Пишемо «заглушки» (**stub**) на все, чого поки не вистачає. Створюємо порожній клас **Info** (файли **info.h**, **info.cpp**).

Обробка завантаження даних

Створюємо майже порожній клас **Builder** (файли **builder.h**, **builder.cpp**) з єдиним відкритим методом **loadData(Info&, const char* filename)**, який далі буде завантажувати вміст файлу в контейнер. Пишемо його реалізацію. Зараз буде достатньо, якщо він буде прикидатися, що все гаразд.

Створюємо функцію **input(Info&, const char* filename)**, що має обробляти введення.

Реалізовуємо функцію **input**: вона використовує **loadData**. Виведення рядка

input in.txt : OK

виконується з **input**.

Нагадую: обробка помилок завантаження даних має виконуватися через механізм винятків, якщо дані не завантажено, то подальша робота сенсу не має. Виняток буде надходити з виклику **loadData**. Перехоплювати виняток слід у головній функції.

Обробка команд **-output**, **-stat**

У випадку некоректної команди або неможливості виконати команду виконання має продовжуватися з повідомленням. Тому винятки тут будуть не допомагати, а заважати.

Проектуємо високорівневі функції, що мають обробляти команди **-output**, **-stat**. Частину вони мають виконувати самі, а безпосереднє виведення інформації мають делегувати іншим, низькорівневим, функціям (ці функції вже будуть розташовані зовсім не в **main.cpp**). Варіантів дуже багато.

Очевидний шлях: написати по функції на кожну команду. Але логіка обробки цих двох команд буде однакова. Виникне дублювання. Це не дуже добре, але принаймні не ускладнює розуміння коду. Якщо однакова логіка обробки команд **-output**, **-stat** призведе до дублювання тільки дуже коротких високорівневих функцій, то як помилка під час оцінювання це розглядатися не буде (у вимогах це теж зафіксовано).

Один з варіантів без дублювання використовує, наприклад, такий фрагмент.

```
using InfoResults=ostream& (ostream&, Info&);
bool do_command(Info& i, const char* fname, InfoResults f){
    if (strcmp(fname,"#con")==0) return bool(f(cout,i));
    ofstream ff(fname);
    bool res= f(ff,i); ff.close(); if (ff.fail()) res=false;
    return res;
}
```

Тоді обробка команд **–output** та **–stat** буде відрізнятися тільки третім аргументом (функцією, що безпосередньо виконує виведення інформації) у виклику функції **do_command**.

Що має бути зроблено:

– мають бути оголошені (в окремій одиниці/одиницях трансляції) низькорівневі функції, що безпосередньо виводять інформацію; жодних діагностичних повідомлень вони на консоль не виводять; на даному етапі вони є заглушками, які прикидаються, що успішно виконали своє завдання;

– усі високорівневі функції, що обробляють команди **-output**, **-stat**, мають бути остаточно означені і розміщені в **main.cpp**; ці функції безпосередньо інформацію не виводять; деякі з них повідомляють успішність виконання команди (OK або UPS). Гілка, що обробляє неуспішне виконання команд, має існувати! Деякі з них використовують функції більш низького рівня, зокрема ті, що вже безпосередньо займаються виведенням інформації.

Інформативні функції

У файлі **main.cpp** розташовуємо функції **help** (виводить стислу довідку по аргументах командного рядка) та, можливо, ще одну, що виводить умову (але не виконавця!).

Головна функція

Записуємо головну функцію, яка виводить інформацію про виконавця (пам’ятаємо про статичну бібліотеку з минулого семестру), умову задачі, стислу довідку по аргументах командного рядка, а потім починає обробляти командний рядок згідно вимог лабораторної роботи, використовуючи функції **input**, дуже можливо/бажано, деякі інші. Спочатку перевіряється кількість аргументів і виконується зчитування інформації. Далі виконується решта команд.

Оскільки обробка помилок завантаження даних має виконуватися через винятки, то майже все тіло **main** має охоронятися (**try**). Діагностичне повідомлення про помилку буде виводитися з пасток. Отже, не забуваємо про **catch**. Зараз можна обмежитися перехопленням тільки винятків типу **exception,** пастка для винятків усіх типів також не завадить.

Увага! Якщо **main** стає «довгою», то це є ознакою того, що було зекономлено на високорівневих функціях. Тоді варто критично переглянути розроблений код.

(*) Отриманий код має бути таким, що як тільки далі будуть реалізовані низькорівневі функції, що безпосередньо виводять дані, та функція **loadData**, то все вже буде працювати як слід. Жодна низькорівнева функція не повинна виводити діагностичні повідомлення на консоль.

У поточному стані програма повинна вміти видавати вихід такого вигляду

```
*****
input in.txt : OK
output out.txt : OK
qwerty : ignored
stat #con :
output out\2.txt : OK
stat s.txt : OK
output out1.txt : OK
*****
```

А також такого

```
*****
nothing to do
*****
```

І такого

```
*****
input in.txt : OK
output out.txt : OK
qwerty : ignored
stat #con :
output out\2.txt : OK
stat s.txt : OK
output undefined
*****
```

Перед тим, як відправляти, це все самостійно тестуємо.

Оцінювання етапу

Не пізніше 5 березня 2019 року на адресу LabAssignment2@i.ua надійшов лист з повним кодом етапу (усі суттєві для проекту cpp- та h-файли і нічого іншого, як вкладення). У темі листа зазначено **Lab32, <номер варіанту>**. Наприклад, **Lab32, 66**

У самому листі зазначено виконавця та компілятор, що використовувався при виконанні лабораторної роботи. Листи з архівами та посиланнями на інтернет-ресурси не припустимі. Один лист – одна лабораторна робота, повністю.

У випадку, якщо тестеру все подобається (не тільки як працює код, але й назви файлів, класу та методів), виконано (*) і не порушено критичні вимоги (у частині, що не суперечить опису етапу), то за вчасне виконання етапу дається **1** бал.

Етап 3 Класи, що зберігають дані (від +0.5 до +1.5 балів)

Зміст етапу. Поповнюємо клас **Info** («Усі дані»), започатковуємо класи *КонтейнерСутностей* та *КонтейнерДеталей*, майже реалізуємо класи *Сутність* та *Деталі*. Назви, звісно, мають бути інші, але змістовні та відповідні умові варіанту.

Зауваження щодо альтернативних розв'язань. У поточну множину варіантів не було додано задач, де без додаткових класів з даними не обійтися. Але існує ситуація, якщо хтось захоче зробити розв'язання лабораторної роботи, краще за наведене мінімальне. Тоді класів буде більше. І класів, що безпосередньо зберігають дані, і контейнерів. І діаграма класів буде трохи іншою та складнішою одночасно.

Кращі за мінімальні розв'язання не забороняються. (Можливо, що за них, якщо будуть доведені до кінця, будуть даватися додаткові «плюшки».) У такому випадку слід спроектувати необхідні додаткові класи та зобразити актуальну діаграму класів. Усі додаткові класи, що не є контейнерами, на цьому етапі мають бути так само реалізовані, як і класи з умови.

Для уникнення можливих непорозумінь під час оцінювання відповідна діаграма класів має бути представлена на узгодження не пізніше 15 березня.

Розробка контейнерів на даному етапі вестися не буде. Тому рішення щодо того, чи писати кілька власних контейнерів, чи писати один шаблон контейнера, чи використати стандартні контейнери STL, можна відкласти на потім. Якщо контейнери будуть відповідати тільки за збереження відповідних даних (та їх кількості), то їх реалізація до переробки існуючого коду не призведе.

Надалі розглядається побудова структури даних з використанням композиції, а не успадкування. Використання успадкування теоретично можливе, хоча на сьогодні не вважається кращим шаблоном для такого сорту задач. Якщо надалі буде прийнято рішення використати не композицію, а успадкування (наприклад, зробити *Сутність* нащадком *КонтейнеруДеталей*), то це великих змін потребувати не буде, хоча деякі вкладені класи будуть змушені «переїхати».

Згадка про схему збереження даних. На верхньому рівні знаходиться клас **Info**, який зберігає всю надану інформацію про сутності та їх деталі. Самі сутності зберігаються в контейнері сутностей. Кожна сутність зберігає певну інформацію про себе (як надану ззовні, так і деяку агреговану), а також множину своїх деталей. Множина деталей окремої сутності зберігається в контейнері деталей, що є частиною об'єкта сутності. Клас *Деталі* зберігає якийсь одиничний факт про сутність: у прикладі (див. розділ 4) він має зберігати дані про одну спробу здати лабораторну.

Порожній клас **Info** був створений на минулому етапі. Зараз створюємо класи *Сутність*, *Деталі*, *КонтейнерДеталей*, *КонтейнерСутностей*. Класи *Деталі, КонтейнерДеталей* мають бути вкладеними у клас *Сутність*. Далі будемо поступово додавати члени до класів **Info**, *Сутність*, *Деталі*. Класи *КонтейнерДеталей*, *КонтейнерСутностей* поки порожні (і можуть залишатися порожніми до кінця цього етапу).

Слід уважно прочитати умову власного варіанту задачі та сформувати перелік того, що надалі може стати членами-даними класів **Info**, **Builder**, *Сутність*, *Деталі*, *КонтейнерСутностей*, *КонтейнерДеталей*.

Перш за все у перелік має потрапити все, що вказано в переліку полів записів файлу.

Далі в перелік потрапляють складові частини заголовку та нижнього колонтитулу (якщо вони стосуються даних чи кількості рядків).

Далі необхідно ще раз уважно прочитати що треба знайти та вивести згідно умови. Там використовується не тільки надана у файлі інформація, але ще певна агрегована чи отримана в результаті обчислень. Частина з неї може бути знайдена та запам'ятована під час завантаження. У деяких варіантах частина агрегованої інформації має отримуватися під час виведення (не може бути обчислена заздалегідь, бо додаткові структури даних з копіями вхідної інформації заборонені умовою лабораторної). Додати відповідні частини до переліку.

Далі ще раз прочитати, що треба знайти та вивести. Те, що треба знайти і чого стосується перший рядок виведення, з ймовірністю 99.9% має зберігатися або легко обчислюватися (тобто не потребувати обходу даних) в об'єкті класу *Сутність*. Те, що треба вивести після першого рядка (див. умову варіанту), відноситься до деталей: або безпосередньо задається у вхідному файлі, або легко

обчислюється (за об'єктом класу *Деталі*, та, можливо, *Сутність*), або отримується в результаті обходу під час виведення.

!!!Сформулювати для себе, що є *Сутність*.

Далі, за допомогою умови варіанту, розподілити складений перелік на частини (варто поставити якісь відмітки):

те, що стосується даних/файлу в цілому

те, що стосується сутності в цілому

те, що стосується деталі і не отримується в результаті обходу під час виведення

те, що стосується деталі і отримується в результаті обходу під час виведення

За допомогою україно-англійського словника підібрати ідентифікатори для всього, що потрапило в перелік (можливо крім того, що буде отримуватися в результаті обходу).

Те, що стосується вхідного файлу, але не стосується даних (наприклад, кількість рядків вхідного файлу із білими включно), має стати відповідними полями класу **Builder**.

Примітка. Кількість небілих рядків файлу = кількість записів файлу +2. Тому це відноситься не до файлу, а до даних у цілому.

Те, що стосується даних у цілому, має стати членами класу **Info**.

Увага. Якщо йдеться про кількість сутностей або про кількість деталей сутності, то це надалі стане методом **size()** відповідного контейнера. За кількість об'єктів, що зберігаються в контейнері, відповідає контейнер. Дублювати інформацію не треба. Відповідні поля в об'єкт, до складу якого входить контейнер додавати не треба. Якщо буде необхідний геттер, то він має делегувати свою роботу методу **size()** контейнера.

Те, що стосується сутності в цілому, має стати членом класу *Сутність*.

Те, що стосується деталі і не отримується в результаті обходу під час виведення, має стати членом класу *Деталі*.

Утворені члени класів переважно мають бути полями.

Те, що легко обчислюється, у більшості випадків має стати методом, що обчислює відповідне значення. **Але**: якщо воно обчислюється для деталі і потребує певну агреговану інформацію від сутності, якій ця деталь належить, то має сенс від такого методу відмовитися, щоб не поскладнювати загальну складність проекту лабораторної. З одного боку, це буде дуже зручно, якщо один виклик методу поверне необхідну інформацію; з іншого боку, реалізація може виявитися дуже неелементарною. Також можливі якісь проміжні варіанти.

Відразу: ідея, коли деталь «знає» свою сутність, вимагає того, щоб під час обробки та завантаження сутності не переміщувалися в пам'яті. Не кожен контейнер буде задовольняти цій вимозі.

Зараз мають існувати поля під всі дані, що надходять у текстовому вхідному файлі, і, можливо, деякі інші. До класу *Сутність* додаємо поле, що є об'єктом класу *КонтейнераДеталей*. До класу **Info** додаємо поле, що є об'єктом класу *КонтейнерСутностей*.

Для усіх полів класів *Сутність* та *Деталі* оголошуємо та означуємо геттери: по одному геттеру для кожного поля. Додаємо цих класів конструктори з аргументами, що повністю встановлюють значення всіх полів (крім полів *Сутності*, що містять обчислювану за деталями інформацію, і, звісно, крім поля-контейнера з деталями). За неможливості створити об'єкт, конструктори мають кидати виняток. Під час реалізації конструкторів ***не обмежуємо себе*** в додаванні до класу допоміжних невідкритих методів, що виконують необхідні перевірки значень. Конструктори мають залишатися короткими та зрозумілими!

Відкритих сеттерів у класу *Деталі* бути не повинно. Перевантажуємо для класу *Деталі* оператори ==, !=, <, <=, >, >= (можна як методи, можна за межами класу). Під час визначення порядку звертаємо увагу на те, як мають бути відсортовані вторинні рядки під час отримання інформації. У деяких варіантах може так статися, що будуть потрібні різні впорядкування *Деталей* (одне під час завантаження й інше для виведення). Тоді буде більш зручно означити порівняння за межами класу.

У класі *Сутність* відкритих сеттерів також бути не повинно. Але має бути метод, що додає деталі. Додаємо до класу *Сутність* відкритий метод **load**, що додає деталі в конкретну сутність.

Метод **load** класу *Сутність* отримує аргументи (числових типів для числових значень, рядки для рядкових значень); отримувати об'єкт класу *Деталі* він не повинен. За отриманими аргументами він конструює об'єкт класу *Деталі* та додає його в контейнер, використовуючи відповідні методи (знайти, додати) класу *КонтейнерДеталей*; на даному етапі клас *КонтейнерДеталей* порожній, тому **load прикидається**, що об'єкт було успішно додано в контейнер; за успішного додавання має оновлювати поля сутності, що зберігають агреговану інформацію щодо деталей. Для тестування наступного етапу буде зручно, якщо **load** на початку роботи буде виводити наладочну інформацію типу «я метод такий-то такого-то класу виконую завантаження таких-то даних туди-то».

Перевантажуємо для класу *Сутність* оператори ==, !=, <, <=, >, >= (можна як методи, можна за межами класу). Під час визначення порядку звертаємо увагу на те, як мають бути відсортовані сутності під час виведення. У деяких варіантах може так статися, що знадобляться два різних упорядкування (одне під час завантаження і інше для виведення). Тоді буде більш зручно означити порівняння за межами класу.

До класу **Info** варто додати геттери для його полів (крім поля-контейнера). Додаємо до класу **Info** два відкритих методи:

load – отримує дані, що стосуються сутності (не відносяться до окремих деталей), і додає відповідну сутність у контейнер (використовує методи класу *КонтейнерСутностей*);

load – отримує дані, що ідентифікують сутність та її деталі, і додає деталі до відповідної сутності.

Кожен з них створює об'єкт класу *Сутність*, далі може виконувати пошук та/або додавання сутності в *КонтейнерСутностей* (використовуючи відкриті методи контейнеру). Другий використовує відкритий метод **load** класу *Сутність*, щоб додати деталі в конкретну сутність.

Аналогічно методу **load** класу *Сутність*, буде зручно, якщо обидва методи **load** класу **Info** будуть спочатку виводити наладочну інформацію типу «я метод такий-то виконую завантаження таких-то даних туди-то», потім конструювати об'єкт класу *Сутність* і далі прикидатися, що все успішно завантажено. Якщо в класі **Info** є поля, що зберігають агреговану інформацію, то після успішного додавання інформації у структури даних вони мають оновлюватися.

Увага! Аргументи конструкторів та відкритих методів **load**: порядок аргументів має в точності відповідати порядку полів, заданому в умові.

Тестування та демо

До класів дозволяється додати відкритий метод **operator string** за умови, що він буде використовуватися виключно під час наладки/тестування.

Якщо проектування переважно ведеться згори до низу, то тестування має виконуватися знизу до гори.

Спочатку слід протестувати клас *Деталі*. Він не є доступним широкому загалу. На час розробки можна створити тестуючий його клас і цей тестуючий клас *ТестуюКласДеталі* зробити другом (friend) того класу, в який вкладено *Деталі*. Щоб не забути його прибрати в кінці роботи над проектом лабораторної у цілому, можна зробити приблизно таке

```
#ifdef DEBUG
   friend class ТестуюКласДеталі;
#endif
```

І далі компілювати проекти з DEBUG, записаним у віконці #defines Build Options проекта (CodeBlocks). Аналогічні можливості є і у Visual Studio.

Окремі методи цього класу можуть тестувати якісь частини коду. Зокрема, слід перевірити, чи правильно конструюється об'єкт з допустимими значеннями аргументів конструктора. Слід перевірити, чи правильно конструктор визначає неможливість створення об'єкта. Також слід перевірити виконання та узгодженість між собою операторів ==, !=, <, <=, >, >=.

Далі слід протестувати клас *Сутність*. Наведений вище підхід з #ifdef DEBUG не забороняється (ні тут, ні далі). Для класу *Сутність* слід протестувати конструктор та порівняння. А також метод **load** у частині, що оновлює значення полів сутності, що зберігають агреговану інформацію про деталі.

Далі слід протестувати клас **Info**. Зокрема його методи **load** у частині, що оновлюють поля об’єкта **Info**.

Написати головну функцію, що чесно та сумлінно виконує всі зазначені тестування, викликаючи функції/методи, що тестують окремі частини. Окремі сценарії мають бути реалізовані у вигляді окремих функцій/методів: перевірка правильності конструювання для кожного класу (hint: варто пам’ятати, що за неможливості побудувати об’єкт конструктор кидатиме виняток); перевірка правильності оновлення для кожного класу (hint: слід програмно порівнювати очікуване значення із значенням, отриманим від геттера), а також перевірка операторів ==, !=, <, <=, >, >=.

Головна функція викликає розроблені функції-сценарії і в кінці виводить **чесний** вердикт: були помилки чи ні (виводить OK на останньому рядку, який друкує).

Одиниці трансляції

cpp та h –файли, що містять означення класів структур даних та їхні поточні реалізації

main.cpp – головна функція

Оцінювання етапу

Не пізніше 26 березня 2019 року на адресу LabAssignment2@i.ua надійшов лист з повним кодом етапу (усі суттєві для проекту cpp- та h-файли і нічого іншого, як вкладення). У темі листа зазначено **Lab33, <номер варіанту>**. Наприклад, **Lab33, 66**

У самому листі зазначено виконавця та компілятор, що використовувався при виконанні лабораторної роботи. Листи з архівами та посиланнями на інтернет-ресурси не припустимі. Один лист – одна лабораторна робота, повністю.

Дозволяється відправляти код етапу кілька разів (наприклад, якщо було усунуто якісь недоліки). У випадку, коли код лабораторної надходив кілька разів, розглядатиметься та оцінюється тільки остання версія (навіть, якщо передостання працювала, а до останньої забули додати частину файлів). Дата/час версії визначається за датою надходження на адресу для лабораторних робіт.

У випадку, якщо тестеру все подобається (не тільки як працює код, але й назви файлів, класу та методів) і не порушено критичні вимоги (у частині, що не суперечить опису етапу), то за етап дається **0.5** балів. Якщо при цьому наявні у демо-частині сценарії **належно** тестують розроблений код, головна функція виносить **чесний** вердикт, що помилок не знайдено, то за це додається ще **1** бал (і загалом буде **1.5** бали за етап).

Етап 4 Клас Builder (від +0.5 до +2 балів)

Зміст етапу. Розбір вхідного файлу та завантаження його вмісту в об'єкт класу **Info**.

Зараз клас **Builder** (файли **builder.h**, **builder.cpp**) має єдиний відкритий метод **loadData(Info&, const char*)**, який реалізовано як заглушку. Поповнюємо **Builder** прихованими полями та методами, щоб він почав виконувати повноцінне завантаження. Деякі міркування щодо того, як це все зробити, можна знайти в методичці. Варто:

вести проектування та розробку методів **Builder** виключно згори до низу;

не соромитися коротких методів;

щоб зробити діагностику помилок, що відповідає умові, можна перехоплювати винятки, що надходять від структур даних, і кидати їх далі (додаючи інформацію про номер рядка, тощо); винятки, що генеруються в методах класу **Builder**, варто відразу кидати з усією необхідною інформацією.

Демо

Скласти приклади поганих (bad_???.txt) та гарних (good_???.txt) вхідних даних (загальним розміром до 50 кБ). Перевірити обробку білих рядків, зокрема, незалежність нумерації записів від наявності білих рядків. Перевірити чи правильно **Builder** обробляє помилки, за які має відповідати. Використати головну функцію етапу 2.

Додаткові «плюшки» етапу (надаються тільки за вчасного виконання етапу)

Якщо належно реалізовано власний клас винятків/систему класів, що є похідним від **exception** і:

знає код помилки,

знає рядок, на якому вона сталася,

знає повідомлення про її суть,

має перевантажений оператор приведення до типу **string**, що повертає той рядок, який слід вивести на консоль,

саме об'єкти цього класу кидаються з методів розроблених класів та власних функцій (під час завантаження винятки бібліотечних функцій мають перехоплюватися у методі **loadData** і далі кидатися як виняток власного типу; винятки від конструкторів можуть поповнюватися необхідною інформацією і кидатися далі; конструктори раніш розроблених класів даних прийдеться змінити, щоб вони кидали винятки власного типу винятків)

то може бути наданий **0.5** додаткові бали.

Якщо «бонусні» розв'язання будуть масово поширені (написав собі, поділився з товаришем), то жодних додаткових балів роздаватися не буде. Запозичених фрагментів у них бути не повинно (навіть з відкритих джерел чи з лабораторних занять). Спроба отримати бонуси з використанням чужого коду буде сприйматися вкрай негативно.

Оцінювання етапу

Не пізніше 01 квітня 2019 року на адресу LabAssignment2@i.ua надійшов лист з повним кодом етапу (усі h- та cpp- коди лабораторної, приклади поганих та гарних вхідних файлів, на яких відбувалося тестування класу **Builder**, загальним розміром до 50кБ). У темі листа зазначено **Lab34, <номер варіанту>**. Наприклад, **Lab34, 66**

У самому листі зазначено виконавця та компілятор, що використовувався при виконанні лабораторної роботи. Листи з архівами та посиланнями на інтернет-ресурси не припустимі. Один лист – одна лабораторна робота, повністю.

Дозволяється відправляти код етапу кілька разів (наприклад, якщо було усунуто якісь недоліки). У випадку, коли код лабораторної надходив кілька разів, розглядатиметься та оцінюється тільки остання версія (навіть, якщо передостання працювала, а до останньої забули додати частину файлів). Дата/час версії визначається за датою надходження на адресу для лабораторних робіт.

У випадку, якщо тестеру все подобається (не тільки як працює код, але й назви та розміри файлів, класу, методів та тестових файлів) і не порушено критичні вимоги (у частині, що не суперечить опису етапу), то за вчасне виконання етапу дається **0,5** бали. Якщо було складено належну систему тестів з коректними даними, на якій програма веде себе очікувано (додати текстовий файл **readme.txt** з описами тестових файлів: що саме в кожному тесті перевіряється/його особливості, тощо; тобто для перевірки чого саме цей тестовий файл потрібний), то за це надається ще **0,5** бали. Якщо було складено належну систему тестів з некоректними даними, програма розпізнає їх як некоректні і в **readme.txt** наведено опис цих тестів, то за це надається ще **0,5** бали.

Етап 5 Контейнери (від +0.5 до +1 балів) ЗОВСІМ ПРОЕКТ

Зміст етапу. Розробка контейнерів.

Поміркувати над тим, які саме структури даних будуть використовуватися в лабораторній (динамічний масив, список, ….), як буде зберігатися інформація (відсортовано чи ні), чи можливе зберігання однакових значень.

План:

1) розробити контейнер, наприклад, що зберігає дійсні, зі стандартним мінімальним набором операцій (empty, size, add, find, sort, …); щодо засобів доступу на читання, то на початковому етапі можна обмежитися виведенням вмісту контейнера на консоль (далі цей метод треба буде прибрати);

2) ретельно протестувати;

3) далі можна або робити з нього шаблон, або явно підставляти потрібний для лабораторної тип елементів;

Якщо необхідно використовувати кілька однаково влаштованих структур даних, то варто зробити шаблон. Інший варіант: розробити інший (=інакше влаштований) контейнер. (У випадку наявності дублювання класів бали будуть знижуватися.)

Підключити реалізовані контейнери до лабораторної роботи. Тобто підмінити ними класи *КонтенерСутностей*, *КонтейнерДеталей*. Hint: можна використати оголошення синоніма

using *КонтенерСутностей*=ююю;

Замість усіх заглушок, що прикидалися начебто виконуються методи контейнерів, підставити належні виклики.

За сценаріями етапу 3 додати відповідні тестові файли.

Протестувати результуючий код на отриманих тестових файлах та на тестових файлах етапу 4.

На цьому етапі ітераторів може ще не бути.

Створити великий тестовий файл і протестувати на ньому.

Оцінювання етапу

Не пізніше 9 квітня 2019 року на адресу LabAssignment2@i.ua надійшов лист з повним кодом етапу (усі суттєві для проекту cpp- та h-файли, тестові файли; загальний обсяг тестових файлів не повинен перевищувати 50кБ, тестовий великий файл надсилати **заборонено**). У темі листа зазначено **Lab35, <номер варіанту>**. Наприклад, **Lab35, 66**

У самому листі зазначено виконавця та компілятор, що використовувався при виконанні лабораторної роботи. Листи з архівами та посиланнями на інтернет-ресурси не припустимі. Один лист – одна лабораторна робота, повністю.

Дозволяється відправляти код етапу кілька разів (наприклад, якщо було усунуто якісь недоліки). У випадку, коли код лабораторної надходив кілька разів, розглядатиметься та оцінюється тільки остання версія (навіть, якщо передостання працювала, а до останньої забули додати частину файлів). Дата/час версії визначається за датою надходження на адресу для лабораторних робіт.

У випадку, якщо тестеру все подобається (не тільки як працює код, але й назви файлів, класу, методів та тестових файлів), STL не використовувався (і далі не використовується в заключній версії), і не порушено критичні вимоги (у частині, що не суперечить опису етапу), то за вчасне виконання етапу дається **0,5** бали. Якщо було складено належну систему тестів, на якій програма веде себе очікувано, то за це надається ще **0,5** бали. (Без працюючого коду бали за тести не даються.)

Етап 6 Обходи контейнерів, отримання вихідної інформації ЗОВСІМ ПРОЕКТ

До контейнерів слід додати ітератори.

Далі реалізувати вивантаження всієї інформації та статистики згідно варіанту.

Варто розробити функції/методи, що вивантажують інформацію в потік, а потім побудувати для них обгортки, що працюють з іменем файлу. Деякі деталі можна знайти в методичці.

Щодо тестування. Варто перевірити чи дійсно завантажується вивантажений файл.

А також виконати певну кількість «ручних» додавань інформації в контейнер, а потім вивантажити її та знов завантажити.

Також слід перевірити формат виведення запитуваної статистики.

Щодо доступу до даних вкладених контейнерів (*КонтенерДеталей*). Тут або треба «дружити» (мінімізує час розробки), або в класі сутність будувати ітератор, що розіменовується в щось інше, ніж об’єкт класу *Деталі*.

Цей етап вже фактично останній. Додаткових балів за нього не передбачено. Але додаткові плюшки заробити можна.

Додаткові плюшки етапу.

Якщо вся лабораторна буде реалізована без використання «друзів», то у залежності від того, наскільки все буде красиво зроблено, можуть бути надані **1** або **2** додаткових бали.

Якщо «бонусні» розв’язання/ідеї будуть масово поширені (написав собі, поділився з товаришем), то жодних додаткових балів роздаватися не буде. Запозичених фрагментів у них бути не повинно (навіть з відкритих джерел чи з лабораторних занять). Спроба отримати бонуси з використанням чужого коду/чужої ідеї буде сприйматися вкрай негативно.