



Programming Assignment: Электронная таблица

Passed · 1/1 points

Deadline Pass this assignment by Jan 25, 11:59 AM +04

Instructions My submission Discussions

В этой задаче мы используем накопленные знания для создания движка электронной таблицы, такой как Microsoft Excel или Google Spreadsheets (мы будем называть эти продукты существующими решениями).

Интерфейс таблицы описывается классом ISheet из файла common.h. Сама таблица создаётся функцией CreateSheet() из того же файла. Вам нужно будет предоставить её реализацию. Также нужно будет предоставить реализацию всех остальных методов и функций из файла common.h (например, методов классов Position, Size и FormulaError).

В качестве решения можно будет сдавать несколько файлов, которые содержат все необходимые реализации.

Общее описание

Ячейки

Таблица состоит из ячеек. Для пользователя ячейки задаются своим *индексом*, т.е. строкой вида "A1" или "C14", или "R2Z". Программно положение ячейки описывается *позицией*, т.е. номерами её строки и столбца, которые представлены классом Position. Причём номера строк и столбцов начинаются с нуля, как это принято в C++. Например, индексу "A1" соответствует позиция (0, 0), а индексу "AB15" - позиция (14, 27). Преобразования между индексом и позицией осуществляются методами Position::FromString() и Position::ToString() (их вам тоже нужно будет реализовать).

Для определённости мы будем полагать, что количество строк и столбцов в нашей таблице не будет превышать 16384. То есть максимальная позиция ячейки составит (16383, 16383) с индексом "XFD16384".

Для получения ячейки с заданной позицией используется метод ISheet::GetCell(). Он возвращает указатель на объект класса ICell. С его помощью можно получить текст и значение ячейки. Текст - это "сырое" содержимое ячейки, а значение - это "отображаемое" содержимое. В существующих решениях текст доступен только во время редактирования ячеек, а значения отображаются всё остальное время.

Ячейка считается пустой, если вызов GetCell() вернёт nullptr, либо если её текст пуст.

В существующих решениях ячейка может содержать данные множества разных форматов: простые текстовые данные, численные значения, валюта, дата, и т.п. В данной задаче мы ограничимся только двумя типами: простой текст (std::string) и числа с плавающей запятой (double). Причём последние могут появиться только как результат вычисления значения формулы.

Формулы

Ячейка трактуется как формула в случае, если её текст начинается со знака "=" (и на этом не заканчивается, то есть текст, который содержит только этот знак, формулой не считается). То, что следует после знака "=", называется *выражением* формулы. Выражение формулы - это простое арифметическое выражение, которое может содержать следующие элементы:

- Числа (5, 3.14)
- Бинарные операции (1*3+4/2-5)
- Ссылки на другие ячейки (A1+B2*C3)

В отличие от существующих решений, результатом вычисления формулы может быть только число. То есть формула не может быть использована, например, для конкатенации строк. Если формула ссылается на другую ячейку, которая не содержит формулу, то её текст трактуется как число. Если эта ячейка пустая, то её значение полагается равным нулю.

Получается, что, если в ячейке записана формула, то текстом ячейки является знак равенства и выражение формулы, а значением ячейки является число - вычисленное значение формулы. Если в ячейке записан простой текст, то значение ячейки обычно совпадает с её текстом. Кроме случая, когда текст начинается с символа "'" (апострофа). Тогда в значении ячейки этот символ не присутствует. Это можно использовать, если нужно начать текст со знака "=", но чтобы он не интерпретировался как формула.

Содержимое ячейки задаётся методом ISheet::SetCell(), в который передаётся текст ячейки. Далее реализация сама решает, как интерпретировать этот текст.

Вставка и удаление строк и столбцов

Важно отметить, что в таблицу можно вставлять строки и удалять их. Поддержка такой операции делает реализацию значительно более интересной. Рассмотрим такой пример (здесь приведены тексты ячеек):

| | | |
|--|---|---|
| | A | B |
|--|---|---|

| | | |
|---|-----|--------|
| 1 | 42 | 43 |
| 2 | =A1 | =A2+B1 |

Если вызвать для такой таблицы InsertCols(1) (то есть вставить столбец перед столбцом "B", который является первым столбцом, если считать с нуля), то мы получим таблицу

| | A | B | C |
|---|-----|---|--------|
| 1 | 42 | | 43 |
| 2 | =A1 | | =A2+C1 |

Обратите внимание, что ячейка "B2" теперь "переехала" в "C2", и её текст изменился. Имеет смысл пользоваться ментальной моделью, в соответствии с которой "B1" до вставки и "C1" после вставки - суть одна и та же ячейка, у которой просто изменился индекс. И подобные изменения индекса должны быть отражены во всех релевантных формулах.

Таким образом, при вставке строк или столбцов могут изменяться тексты ячеек с формулами, но значения ячеек не меняются.

С удалением ячеек дело обстоит чуть сложнее. Рассмотрим тот же пример, только вызовем для него DeleteCols(0) (то есть удалить столбец "A"). Тогда мы получим следующую таблицу:

| | A |
|---|-----------|
| 1 | 43 |
| 2 | =#REF!+A1 |

Второе слагаемое в бывшей ячейке "B2" обновилось корректно, а первое заменилось на "#REF!". Это сообщение об ошибке, которое значит, что мы пытаемся обратиться к недоступной ячейке. Что логично, она ведь была удалена. Вычисление значения такой формулы вернёт нам не численное значение, а эту самую ошибку. Это отражено в типе ICell::Value - он может содержать либо простой текст, либо число, либо ошибку.

Таким образом, при удалении строк или столбцов могут изменяться как тексты ячеек с формулами, так и их значения. При этом, в качестве новых значений могут фигурировать сообщениями об ошибках.

Печать таблицы

С помощью методов ISheet::PrintValues() и ISheet::PrintTexts() таблицу можно распечатать целиком в некоторый поток. При этом будет выведена минимальная прямоугольная область, включающая в себя все непустые ячейки. Размер этой области можно запросить в явном виде с помощью метода ISheet::GetPrintableSize().

Для печати формул предназначен метод ISheet::GetExpression(), который возвращает текстовое представление формулы без пробелов и лишних скобок. Мы считаем, что определить нужность скобки можно, смотря на типы двух вершин — текущую и дочернюю, что есть некоторое упрощение. Например, в выражении A1 / (A2 / A3) скобки нужны (без них значение формулы может изменяться), а скобки в выражении (A1 / A2) / A3 не нужны. Таким образом, чтобы понять, нужно ли вставить между вершиной и ее дочерней вершиной скобки, нужна следующая информация:

- Тип текущей вершины,
- Тип дочерней вершины,
- Является ли дочерняя вершина левой или правой (актуально, если родительская вершина — бинарный оператор).

Мы ожидаем, что вы будете использовать следующую матрицу нужности скобок, где строчки соответствуют типам родительских, а столбцы — типам дочерних вершин:

| | ++* | *-* | *-* | */* | ±* | атом |
|-----|--------|--------|--------|--------|----|------|
| ++* | | | | | | |
| *-* | справа | справа | | | | |
| *-* | нужны | нужны | | | | |
| */* | нужны | нужны | справа | справа | | |
| ±* | нужны | нужны | | | | |

В этой таблице пустая ячейка означает, что скобки не нужны, "справа" означает, что скобки нужны, если ребенок правый, "нужны" означает, что скобки нужны всегда. Под атомами в таблице мы понимаем числа и ссылки на ячейки.

Заметим, что с такими правилами преобразование AST \rightarrow expression \rightarrow AST может приводить к AST, отличному от исходного. Например, скобки в формуле -(A1 * A2) не нужны, хотя выражения -(A1 * A2) и -A1 * A2 имеют разные AST: UnaryMinus(Mul(Cell(A1), Cell(A2))) и Mul(UnaryMinus(Cell(A1)), Cell(A2)) соответственно.

Обработка ошибок

Электронная таблица - достаточно гибкая структура, и её легко можно привести в неконсистентное состояние. Реализация должна корректно вести себя в случае всех возможных ошибок.

Могут быть выброшены следующие исключения:

- **Некорректная позиция.** Программно есть возможность создать

экземпляр класса Position, в котором хранится некорректная позиция. Например (-1, -1). Попытка передать такую позицию в методы предоставляемых интерфейсов должна приводить к исключению InvalidPositionException. Гарантируется, что позиции, возвращаемые методами интерфейсов (например, ICell::GetReferencedCells()) всегда корректны.

- **Некорректная формула.** Если в ячейку с помощью метода ISheet::SetCell() пытаются записать синтаксически некорректную формулу (например, =A1+"), то реализация должна выбросить исключение FormulaException, а значение ячейки не должно измениться. Формула считается синтаксически некорректной, если она не удовлетворяет предоставленной грамматике.
- **Циклическая зависимость между ячейками.** Если в ячейку с помощью метода ISheet::SetCell() пытаются записать формулу которая привела бы к циклической зависимости между ячейками, то реализация должна выбросить исключение CircularDependencyException, а значение ячейки не должно измениться.
- **Слишком большая таблица.** Напрямую нельзя создать ячейку с позицией превышающей максимальную, равно как и указать такую ячейку в формуле. Однако можно заполнить ячейку близкую к максимальной, или согнаться на таковую в формуле, а затем вставить в таблицу ещё несколько строк/столбцов, тем самым "выдвинув" ячейку за пределы максимальной позиции. Программа не должна позволять так делать, а вместо этого должна бросать исключение TableTooBigException.

Значение ячейки с формулой может принимать следующие ошибочные состояния, описанные в классе FormulaError:

- #REF! - Ссылка на несуществующую ячейку. См. пример выше про удаление ячеек.
- #VALUE! - Ссылка на ячейку, которая не является формулой и не может быть трактована как число; необходимо, чтобы вся ячейка трактовалась как число, то есть ячейка "11PM" приведёт к данной ошибке.
- #DIV/0! - В процессе вычисления значения формулы возникло деление на ноль, или переполнился тип double. Эту проверку можно выполнить с помощью функции `std::isfinite()`.

При этом, если формула в ячейке зависит от формулы в другой ячейке, вычисление которой привело к ошибке, то текущая формула вернёт ту же ошибку. То есть ошибки "распространяются" наверх по зависимостям. Если формула в ячейке зависит от формул в нескольких ячейках, и они возвращают разные ошибки, то текущая формула может вернуть любую из этих ошибок.

Обработка ошибок - один из наиболее сложных аспектов разработки ПО. В частности потому, что техническое задание очень часто не рассматривает каких-то особых случаев и не уточняет, как должна себя в них вести программа. В данном условии мы могли пропустить описание каких-то особых случаев. Зато мы предоставляем множество юнит-тестов в файле main.cpp. Если у вас возникают вопросы, как должна себя вести программа в каком-то случае, проверьте - быть может, он в явном виде описан в тестах.

Если ваша реализация проходит все юнит-тесты, этого должно быть достаточно для того, чтобы тестирующая система посчитала её корректной. Однако, тесты не проверяют быстродействия реализации. Об этом позаботьтесь сами.

Быстродействие и сложность

Здесь мы сформулируем желаемую сложность работы нашей реализации. Для удобства её оценки, давайте сделаем несколько предположений. На практике таких ограничений мы делать не будем, это нужно исключительно для удобства оценки:

- Количество ячеек, на которые ссылается произвольная формула, не превосходит некоторой константы
- Общее количество ссылок на ячейки вне печатной области ограничено некоторой константой

Далее давайте скажем, что K - количество ячеек в печатаемой области таблицы. Тогда от реализации мы ожидаем следующих асимптотических сложностей работы в разных сценариях:

| # | Сценарий | Сложность |
|---|---|-----------|
| 1 | Вызов любого метода интерфейса ISheet | O(K) |
| 2 | Вызов метода ISheet::GetCell() | O(1) |
| 3 | Вызов метода ICell::GetValue() | O(K) |
| 4 | Вызов метода ICell::GetText() | O(1) |
| 5 | Вызов метода ICell::GetReferencedCells() | O(1) |
| 6 | Повторный вызов метода ISheet::SetCell() с теми же аргументами | O(1) |
| 7 | Повторный вызов метода ICell::GetValue() при условии, что значения ячеек, от которых данная ячейка зависит напрямую или опосредованно, не менялись с момента первого вызова (вставка строк и столбцов не приводит к изменению значений ячеек) | O(1) |

Эти требования подобраны не случайно, а чтобы отдать предпочтение наиболее часто встречающимся сценариям, и чтобы сделать возможной подходящую реализацию. Давайте посмотрим на них более подробно.

Структура данных для хранения ячеек

Предоставленный интерфейс ISheet содержит как методы вроде GetCell(), которые требуют произвольной индексации по хранилищу ячеек, так и методы вроде InsertCells(), которые требуют вставки элементов в произвольное место хранилища. Очевидно, что для обеспечения эффективной работы обоих видов

запросов нужна какая-то нетривиальная структура данных.

Однако, требования по сложности говорят нам, что метод `GetCell()` должен обрабатывать за константное время, в то время как ограничения на время работы `InsertCells()` самые либеральные. То есть мы предполагаем, что операции произвольного доступа на практике нужны чаще, чем операции вставки. Такое ограничение позволяет нам использовать простую структуру данных вроде векторов.

Кеширование вычислений

Давайте рассмотрим интересный пример, где с помощью формул мы вычисляем в таблице значения [треугольника Паскаля](#):

| | A | B | C | D |
|-----|-----|--------|--------|--------|
| 1 | 1 | | | |
| 2 | =A1 | =A1+B1 | | |
| 3 | =A2 | =A2+B2 | =B2+C2 | |
| 4 | =A3 | =A3+B3 | =B3+C3 | =C3+D3 |
| ... | ... | ... | ... | ... |

Каждая (ну, почти каждая) ячейка зависит от двух ячеек в предыдущей строке. Это значит, что каждая ячейка опосредованно зависит от $O(k)$ других ячеек. Теперь пусть мы вызываем метод `ISheet::PrintValues()`, который вычисляет значения всех ячеек. При наивной реализации без кеширования вычисление одной ячейки будет иметь сложность $O(k)$. Таким образом, общая сложность работы метода составит $O(k^2)$, что противоречит требованию #1.

Кроме того, из требования #7 напрямую следует необходимость сохранять кеш значений ячеек.

На практике кеширование любого рода обычно требует повышенной аккуратности в связи с возможностью многопоточного доступа. В данной же задаче для упрощения мы не требуем корректности работы реализации интерфейса в многопоточной среде, что существенно упрощает нам жизнь.

Тем не менее, нужно затратить некоторые усилия на обеспечение эффективности инвалидации кешированного значения. Действительно, если значение ячейки изменилось, то кеш значений всех ячеек, что от неё зависит непосредственно или опосредованно, должен быть инвалидирован. Причём требование #6 говорит нам, что это должно происходить быстро.

Рекомендации по реализации

Ниже мы дадим несколько рекомендаций, которые, на наш взгляд, позволяют выполнить наиболее простую реализацию, удовлетворяющую обозначенным требованиям.

Ячейки в памяти

Как было сказано выше, сформулированные ограничения позволяют хранить таблицу как вектор векторов. При этом, удобно думать, что при вставке и удалении строк и столбцов ячейки "двигаются" в соседние позиции. То есть удобно отвязать ячейку от её индекса. Для этого можно выделять ячейки в динамической памяти, а в векторах хранить лишь указатели (желательно умные) на объекты ячеек.

При этом нет смысла выделять память под пустые ячейки. То есть наши векторы вполне могут содержать нулевые указатели. Спецификация метода `ISheet::GetCell()` позволяет нам это делать.

Парсинг формул

В предыдущем материале для чтения мы рассказали вам, как пользоваться системой ANTLR для генерации парсера. Здесь мы предлагаем вам воспользоваться этим знанием. В шаблон решения уже входит готовая грамматика и все нужные для работы ANTLR файлы, мы также интегрировали генерацию всех необходимых C++-файлов из грамматики в `CMakelists.txt`. Вам остается только реализовать бизнес-логику формул. Для этого мы рекомендуем реализовать `Listener`, который будет строить AST формулы из ваших собственных классов, и дальнейшие манипуляции уже проводить с этим деревом.

Граф зависимостей

Для эффективной инвалидации кешированных значений можно использовать граф зависимостей. То есть для каждой ячейки нам нужно знать, от каких ячеек зависит она (*исходящие ссылки*), и какие ячейки зависят от неё (*входящие ссылки*).

При изменении значения ячейки достаточно пройтись рекурсивно по всем входящим ссылкам и инвалидировать кеши соответствующих ячеек. Причём, если кеш какой-то ячейки уже был инвалидирован, нет смысла продолжать рекурсию дальше. Именно эта оптимизация и позволит достичь константной сложности в требовании #6. Кроме того, граф зависимостей упростит предотвращение циклических зависимостей.

Вершинами данного графа являются ячейки, и хранить его удобно как список ребер, входящих в исходящих из конкретной ячейки. При изменении ячейки необходимо будет обновлять список исходящих ссылок (ребер), а также списки входящих ссылок (ребер) для всех ячеек, от которых данная ячейка зависела и станет зависеть.

Столт помнить о ситуации, когда в ячейку записывают формулу, которая указывает на пустую ячейку. Поскольку для этой пустой ячейки нам нужно сохранить список входящих ссылок, придётся создать фиктивную ячейку без содержимого.

Подготовка решения

spreadsheet_starter_files.zip

Вам дан архив, в котором содержится несколько сущностей:

- Файлы common.h и formula.h содержат основные интерфейсы и функции, которые вам нужно будет реализовать
- Файл test_runner.h это наш старый добрый юнит-тест фреймворк
- Файл main.cpp содержит юнит-тесты
- В файле Formula.g4 описана грамматика формул для ANTLR
- В папке antlr4_runtime лежит библиотека ANTLR
- Файл antlr-4.7.2-complete.jar это java-пакет, который использует ANTLR для кодогенерации
- Файл FindANTLR.cmake нужен для подключения ANTRL с помощью CMake
- Файл CMakeLists.txt это точка входа для CMake

Распакуйте архив и используйте CMake, чтобы сгенерировать файлы для вашей любимой билд-системы или IDE. Запустите сборку. Всё должно успешно скомпилироваться, но будут ошибки линкера для проекта spreadsheet. Это нормально, т.к. в нём пока нет необходимых реализаций.

Добавьте файлы со своими реализациями (после этого может потребоваться перезапустить CMake). Когда сборка станет успешной, можно запустить программу, она выполнит юнит-тесты. Чтобы быстрее получить компилирующиеся программу, рекомендуется начать с "затычек" в качестве реализаций. Очевидно, юнит-тесты будут для них падать. Далее постепенно добавляйте реализации, пока все юнит-тесты не пройдут.

В обучающих видео вам рассказали, как устанавливать и использовать cmake под Windows/MinGW. Ниже вы найдёте инструкции по использованию CMake для нескольких других систем, на которых мы тестиировали.

Windows, Visual Studio 2017

Запустите консоль из корня распакованного архива. В консоли выполните:

```
mkdir build && cd build
```

```
cmake .. -G "Visual Studio 15 2017" -A x64 -Wno-dev
```

Откройте файл build/spreadsheet.sln.

При добавлении новых файлов никаких дополнительных действий не требуется. Просто нажмите Build | Build Solution и файлы проектов будут автоматически обновлены, чтобы включить добавленные файлы.

macOS

Установите системные утилиты для сборки программ:

```
xcode-select --install
```

Опционально, установите XCode из App Store.

Установите [Homebrew](#) и установите CMake:

```
brew install cmake
```

Опционально, установите Ninja:

```
brew install ninja
```

Создайте директорию для билд-системы и сгенерируйте файлы для ее настройки:

```
mkdir build && cd build
```

```
cmake .. -Wno-dev -DCMAKE_BUILD_TYPE=Debug
```

В такой конфигурации будет использована билд-система make. Она старая и лучше использовать что-то другое, например:

```
cmake .. -Wno-dev -DCMAKE_BUILD_TYPE=Debug -G Ninja cmake .. -Wno-dev -DCMAKE_BUILD_TYPE=Debug -G Xcode
```

Далее, в зависимости от того, какую билд-систему вы выбрали:

- По умолчанию (make): запустите make
- Xcode: откройте сгенерированный проект в Xcode
- Ninja: ninja. Его билд всегда параллельный и сам определяет подходящее число параллельных процессов

Debian & Ubuntu

Установите необходимые пакеты (ninja-build опционален):

```
sudo apt-get install cmake ninja-build pkgconf uuid-dev
```

Создайте директорию для билд-системы и сгенерируйте файлы для ее настройки:

```
mkdir build && cd build
```

```
cmake .. -Wno-dev -DCMAKE_BUILD_TYPE=Debug -DWITH_LIBCXX=Off
```

В такой конфигурации будет использована билд-система make. Она старая и лучше использовать что-нибудь другое, например Ninja:

```
cmake .. -Wno-dev -DCMAKE_BUILD_TYPE=Debug -G Ninja -DWITH_LIBCXX=Off
```

Обратите внимание, что мы отключаем форсированное

использование libc++ в clang++. Если в вашей системе нет libstdc++ (например, если вы используете FreeBSD, но все же читаете этот гайд), можно убрать эту опцию.

Далее, в зависимости от того, какую билд-систему вы выбрали:

- По умолчанию (make): запустите make
- Ninja: ninja. Его билд всегда параллельный и сам определяет подходящее число параллельных процессов

Отправка файлов

В качестве решения сдайте архив со всеми .cpp и .h файлами, в которых содержатся ваши реализации. Не включайте в него файлы, которые были в исходном архиве.

