Лабораторная работа 3
Длинная арифметика

Цель работы

Изучить особенности работы классами и операторами в C++.

Стандарт языка

C++17 или новее. Требования к работам и Оформление исходных текстов.

Описание

В программе должен быть реализован **класс LN**, позволяющий выполнять арифметические операции над целыми числами произвольной точности. С его использованием должна проводиться работа с данными, получаемыми из входного файла.

Внутреннее представление чисел: упакованное двоичное, то есть, число хранится как массив uint8_t, uint16_t, uint32_t или uint64_t, где каждый элемент массива хранит по 8, 16, 32 или 64 бита числа соответственно.

Текстовое представление числа (при конвертации и в файлах) – в шестнадцатеричной системе.

Класс должен иметь конструкторы:

1. из long long со значением по умолчанию: 0;
2. из строки символов C (const char *);
3. из строки std::string_view;
4. конструктор копирования;
5. конструктор перемещения.

Для класса должны быть реализованы операторы:

1. оператор копирующего присваивания;
2. оператор перемещающего присваивания;
3. арифметические: +, -, *, /, %, ~, - (унарный);
4. комбинация арифметических операций и присваивания (например, +=);
5. сравнения: <, <=, >, >=, ==, !=;
6. преобразования типа в: long long (с генерацией исключения в случае, когда значение не умещается), bool (неравенство нулю);
7. создания из литерала произвольной длины с суффиксом _ln (например, должно работать выражение: LN x; x = 123_ln;).

Если какой-то оператор не реализован – возвращаем UNSUPPORTED.

Результат операций сравнения: 0 (для false) или 1 (для true).

Число -0 должно быть полностью эквивалентно 0 (включая вывод).

Реализации операций умножения (оператор ‘*’), деления (оператор ‘/’), остаток от деления (оператор ‘%’) и квадратный корень (‘~’) **должны работать с адекватной скоростью** (то есть требуется алгоритм уровня “в столбик”, а не “умножение на n путём сложения n раз”).

Квадратный корень и деление **округляют к 0**. Результат при взятии корня из отрицательного числа или делении на ноль: **NaN**. Результат арифметических действий и сравнения с NaN: в соответствии со стандартом IEEE-754.

Необходимо реализовать вспомогательные функции/методы (сложение, вычитание, универсальное сравнение (возвращает -1, 0, 1)), которые будут использоваться в функциях/методах: оператор ‘+’, оператор ‘<’ и пр. Например, при реализации и оператора ‘+’, и оператора ‘-’ необходимо выполнять сложение или вычитание, в зависимости от знаков входных чисел.

В классе ***рекомендуется*** отдельно **хранить знак числа и признак NaN**. Разряды числа в массиве ***рекомендуется*** **начинать с младшего** (аналогично **little endian**).

Реализация по файлам

Объявление класса необходимо поместить в заголовочный файл LN.h, реализация крупных методов (больше 2 строк) класса должна быть в LN.cpp. Код функции main располагается в main.cpp.

Создавать свои файлы (.h/.cpp) со вспомогательным кодом не запрещено.

Работа с памятью

Нехватка памяти в методах класса должна обрабатываться через **исключения** C++.

Кол-во перевыделений памяти в конструкторах и операциях не должно быть порядка длины числа (или больше…).

Ограничение

Использовать STL, string или другие стандартные классы C++ с нетривиальными деструкторами (= отличными от деструктора, который вы получаете по умолчанию с классом, который ничего не делает, Destructors - cppreference.com) ***внутри*** класса LN не запрещено, НО оценивается в **-5 баллов**. Соответственно, можно получить отрицательное кол-во баллов за эту работу в целом.

Использовать STL, string или другие стандартные классы C++ ***вне*** класса LN можно без ограничений.

Замечание: методы класса LN, реализованные вне его описания, операторы работы с LN во внешней форме и прочие функции, используемые для реализации функциональности класса LN – считаются ***внутри*** LN.

Формат аргументов командной строки

Аргументы программе передаются через командную строку:

<имя_входного_файла> <имя_выходного_файла>

Входной файл содержит выражение в форме обратной польской записи. Каждое число и знак операции (‘+’, ‘-’, ‘*’, ‘/’, ‘%’ – остаток от деления, ‘~’ – квадратный корень, ‘_’ – унарный минус, ‘<’, ‘<=’, ‘>’, ‘>=’, ‘==’, ‘!=’) располагаются на отдельной строке. Каждая строка оканчивается символом перевода строки.

Выходной файл должен содержать состояние стека на момент завершения работы программы. Каждое значение находится на новой строке, начиная с вершины, каждая строка оканчивается символом перевода строки.

Корректность входных данных гарантируется. Также гарантируется отсутствие антипереполнения стека.

Во входном файле данные могут как в lower-, так и в uppercase (пример в репозитории курса). В выходном файле – в uppercase.

Пример входных и выходных данных:

input	**output**
2 -3 2 + F - 10 ==	1 2

Автотесты

Автотесты на Github можно запустить только вручную через вкладку Actions. Функционал с переоткрытием Pull Request отключен.

Как запускать тесты описано в памятке:

https://skkv-itmo.gitbook.io/c-cpp-cource/organization/labs/cicd-github#run-autotests

Поле	**Возможные значения**	**Значение по умолчанию**	**Описание**
`example`	`true` `false`	`true`	Тест из примера
`plus_op`	`true` `false`	`true`	Тест с длинными числами и +
`compare_op`	`true` `false`	`true`	Тест с длинными числами и >

Пример запуска (терминал открыт из папки корня репо):

```
gh workflow run BuildTest -f example=false -f plus_op=true -f compare_op=false
```

Несколько полей могут принимать значение `true`.

Подробнее о том, как можно смотреть запуски из вашей IDE:

https://skkv-itmo.gitbook.io/c-cpp-cource/ide-features/gh-actions-ide