Лабораторная работа № 10. Класс string, с-строки

В качестве решения лабораторной работы требуется сдать в **iRunner** файл **.ZIP**, в котором будет решение всех задач из лабораторной работы. Если Вы сдаете неполное решение, то всё равно необходимо создать функции и методы на все пункты (при этом они могут ничего не делать и возвращать просто любую константу). В противном случае, как и в случае использования неверных сигнатур, при компиляции на сервере произойдет ошибка.

Имена файлов с реализацией:

```
my_string.{h,cpp}
my_string_view.{h,cpp}
```

Для файлов с тестами используйте постфикс "`_tests`".

Задача 1 . String. Создайте класс String, аналогичный по поведению классу std::string из стандартной библиотеки. Он должен быть основан на С-строках (строковых массивах) и быть динамически расширяемым.	**Задача 2. StringView.** Создать класс StringView, аналогичный по поведению классу std::string_view из стандартной библиотеки. Класс должен иметь два приватных поля - указатель на строковый массив и его длину. Класс не должен владеть строковым массивом (то есть управлять временем его существования).
Использовать класс std::string и его методы запрещается.	Использовать класс std::string_view и его методы запрещается.
Необходимо реализовать следующую функциональность:	Необходимо реализовать следующую функциональность:
	• Константное статическое поле StringView::npos, принимающее целочисленное значение, равное std::numeric_limits<int>::max().
• Конструктор по умолчанию, инициализирующий объект пустой строкой. • Конструктор от С-строки (предполагается, что строка будет нуль-терминированной). • Конструктор String(int count, char ch), инициализирующий объект строкой из count символов ch. • Конструкторы и операторы копирования и перемещения. • Деструктор, корректно освобождающий выделенную память.	• Конструктор по умолчанию, инициализирующий объект пустой строкой. • Конструктор StringView(const char* str, int count = npos) от С-строки. Необходимо использовать префикс переданной строки длины min(count, length(str)).
• Метод int length(), возвращающий текущую длину строки. • Метод empty(), возвращающий true, если строка пуста, и false в противном случае. • Метод c_str(), возвращающий текущую строку в виде указателя на константную нуль-терминированную С-строку.	• Метод int length(), возвращающий текущую длину строки. • Метод empty(), возвращающий true, если строка пуста, и false в противном случае. • Метод data(), возвращающий внутренний указатель на С-строку.
• Оператор обращения по индексу [], возвращающий ссылку на соответствующий символ строки, а также его константный аналог. Считать, что обращение по индексу, превышающему	• Оператор обращения по индексу [], возвращающий константную ссылку на соответствующий символ строки. Считать, что обращение по индексу, превышающему длину строки, является неопределенным поведением.

длину строки, является неопределенным поведением. • Методы front() и back(), возвращающие ссылки на первый и последний элемент символы строки соответственно, а также их константные аналоги. Считать, что вызов этих методов для пустой строки является неопределенным поведением. • Метод reserve(int capacity), расширяющий внутренний массив так, чтобы в него поместилась строка длины capacity. Если текущая вместимость массива уже достаточна для этого, метод может ничего не делать. • Метод push_back(char ch), добавляющий переданный символ в конец строки. При недостатке памяти следует произвести релокацию внутреннего строкового массива и увеличить его размер в два раза. • Метод pop_back(), удаляющий последний символ из строки. Считать, что вызов метода для пустой строки является неопределенным поведением. • Метод clear(), удаляющий все символы из строки. • Метод insert(int index, const String& str), вставляющий строку str на позицию index. • Метод insert(int index, const char* str, int count), вставляющий первые count символов строки str на позицию index. Гарантируется, что среди этих count символов будут отсутствовать нулевые. • Метод erase(int index, int count = 1), удаляющий из строки min(count, length() - index) символов, начиная с index (включительно). • Обеспечить возможность конкатенации двух строк при помощи оператора +. • Обеспечить возможность эффективной конкатенации двух строк при помощи оператора +=.	• Метод at(int index), аналог оператора [], работающий так же, но выбрасывающий исключение типа std::out_of_range с произвольным сообщением, если переданный индекс некорректен. • Методы starts_with(StringView v) и ends_with(StringView v), возвращающие true, если текущая строка начинается переданным префиксом / заканчивается переданным суффиксом, и false в противном случае. • Методы remove_prefix(int count) и remove_suffix(int count), удаляющие count символов с начала / конца строки. Считать, что если count > length(), то поведение не определено. • Метод StringView substr(int pos, int count = npos), возвращающий подстроку длины min(count, length() - pos), начинающуюся с символа pos. • Метод int find(StringView v, int pos = 0), осуществляющий поиск переданной подстроки в строке. Поиск необходимо начинать с символа pos. Если подстрока найдена, необходимо вернуть индекс ее первого символа, в противном случае требуется вернуть StringView::npos. • Метод int find(char ch, int pos = 0), осуществляющий поиск переданного символа в строке, начиная с позиции pos. Логика такая же, как у предыдущей функции.
• Метод int compare(const String& str) для лексикографического сравнения строк. Он должен работать аналогично соответствующему методу класса std::string. • Метод int compare(const char* str) с аналогичной логикой. • Обеспечить возможность сравнения двух строк при помощи любого из операторов <, <=, >, >=, ==, !=. Логика сравнения такая же, как у функции compare. • Обеспечить возможность сравнения String и const char*, const char* и String.	• Метод int compare(StringView v) для лексикографического сравнения строк. Он должен работать аналогично соответствующему методу класса String. • Обеспечить возможность сравнения двух строк при помощи любого из операторов <, <=, >, >=, ==, !=. Логика сравнения такая же, как у функции compare.

Замечания:	**Замечания:**
• Обратите внимание, что метод c_str() должен корректно работать даже тогда, когда внутри String хранится пустая строка (иными словами, данный метод всегда должен возвращать валидный указатель, а не nullptr). • В наиболее популярных реализациях стандартной библиотеки C++ методы pop_back() и clear() не уменьшают размер внутреннего массива (они изменяют size, но не изменяют capacity). Хотя это не будет проверяться, с вашей стороны будет логично использовать такую же логику в своем классе. • При реализации операторов сравнения разрешается использовать spaceship operator <=>, добавленный в C++20. Он может сократить объем кода, который вам нужно написать. На сервере ваше решение будет собираться компилятором clang-10 (с флагом -std=c++20) и линковаться с реализацией стандартной библиотеки libc++-10. **UPD**: пример работы с данным оператором. • **UPD**: Если это возможно, предпочитайте использовать библиотечные функции strncpy, strncmp, а не реализовывать их логику самостоятельно. Благодаря определенным оптимизациям на уровне ассемблерного кода (с которыми вы познакомитесь в следующем семестре), библиотечные функции могут работать во много раз быстрее вашего вручную написанного цикла.	• Обратите внимание, что логика работы метода data() у std::string_view отличается от логики метода c_str() у std::string. Требуется лишь, чтобы диапазон [data(), data() + length()) был валидным, и каждый элемент в этом диапазоне был равен соответствующему элементу оригинальной строки. Нет никаких гарантий того, что, например, данный диапазон будет нуль-терминированным. Вы можете использовать такую же логику при реализации методов своего класса. • Как и в случае со String, при реализации операторов сравнения разрешается использовать spaceship operator <=>, добавленный в C++20.

Подробности для интересующихся: при выполнении определенных условий современному x86_64-процессору неважно, копирует (сравнивает) он 1 байт или 8 байт, это занимает одинаковое количество времени. Кроме того, опять же, при выполнении определенных условий процессор может использовать специальные модули SSE и AVX, которые позволяют очень быстро работать с блоками памяти размером до 128 байт. В реализациях strncpy, strncmp в популярных компиляторах предусмотрено использование таких возможностей процессора. Ссылки на stackoverflow: Do strncpy/memcpy/memmove copy the data byte by byte or in another efficiently way?, Why is this slower than memcmp.