Отчёт по индивидуальному заданию №3 «Класс String»

Стойко Елисей, 5130904/50005

Вариант 11

Содержание

1. Задание

Разработайте класс `String`, аналогичный классу `std::string`, с основными операциями для работы со строками.

2. Требования к реализации

Использование строки в стиле C, считая, что символы имеют коды от 0 до 127
Правильное управление памятью (правило трех/пяти)
Использование идиомы `copy-and-swap`
Безопасность исключений
Возможно использование библиотечных функций из `<cctype>`
Запрещается использование функций из `<cstring>`
Тестирование можно выполнить в функции `main`.

3. Особенности реализации

3.1. Оператор ввода (>>)

1. очищаем строку перед записью (метод `clear()`)
2. пропускаем все пробелы до первого *непробельного* символа
3. проверяем состояние потока
4. добавляем к строке по символу из потока до пробела / конца строки
5. символ добавляем в формате `... += { ch, '\0' }` для корректной работы оператора `+=`

```
std::istream& operator>>(std::istream& stream, String& str) {
    str.clear();
    char ch = 0;

    while (stream.get(ch) && std::isspace(ch)) {

    }

    if (!stream) {
        return stream;
    }

    char buf[2] = { ch, '\0' };
    str += buf;

    while (stream.get(ch) && !std::isspace(ch)) {
        char buf[2] = { ch, '\0' };
        str += buf;
    }

    return stream;
}
```

3.2. Оператор < (операторы сравнения)

1. строки сравниваются лексикографически, как в `std::string`
2. проверяем каждый символ до конца минимальной по длинне строки
3. если все символы равны, сравнимаем длины строк
4. оператор `==` написан аналогично
5. остальные операторы сравнения реализованы через `<` и `==`

```
bool String::operator<(const String& other) const {
    size_t minLen = std::min(length_, other.length_);
    for (size_t i = 0; i < minLen; ++i) {
        if (data_[i] < other.data_[i]) {
            return true;
        }
        if (data_[i] > other.data_[i]) {
            return false;
        }
    }
    return length_ < other.length_;
}
```

3.3. Конкатенация строк (операторы + и +=)

1. оператор `+` реализован через оператор `+=`
2. чтобы избежать проблем с копированием символов через `my::strcpy` при самоприсваивании вида `a += a`, используется временная переменная `temp` и рекурсивный вызов функции
3. оператор `+=` для `const char*` реализован через оператор `+=` для `const String&`, путём приведения C-строки к типу `String` через реализованый по условию конструктор

```
String& String::operator+=(const String& other) {
    if (other.length_ == 0 || other.data_ == nullptr) {
        return *this;
    }

    if (this == &other) {
        String temp(other);
        return *this += temp;
    }

    size_t newLen = length_ + other.length_;
    if (newLen + 1 <= capacity_) {
        my::strcpy(data_ + length_, other.data_);
    } else {
        capacity_ = newLen + 1;
        char* newData = new char[capacity_];

        my::strcpy(newData, data_);
        my::strcpy(newData + length_, other.data_);

        delete[] data_;
        data_ = newData;
    }
    length_ = newLen;
    return *this;
}

String String::operator+(const String& other) const {
    String result(*this);
    result += other;
    return result;
}

String& String::operator+=(const char* str) {
    return *this += String(str);
}
```

4. Дополнительное задание (вариант 11)

11. Сформировать новую строку, заменив в исходной строке все прописные латинские буквы на строчные.

```
String String::lower() const {
    String newString{*this};
    int delta = 'a' - 'A';

    for (size_t i = 0; i < newString.length_; ++i) {
        if (newString[i] >= 'A' && newString[i] <= 'Z' && std::isupper(newString[i]))
{
            newString[i] = newString[i] + delta;
        }
    }

    return newString;
}
```

Метод возвращает новую строку, в которой все символы `'A'..'Z'` заменены на соответствующие `'a'..'z'`. Остальные символы остаются без изменений.

5. Тест-план

№	Тестируемый элемент	Описание теста	Входные данные	Ожидаемый результат
1	`String()` (констр. по умолчанию)	Пустая строка	`String s;`	`s.length() == 0`, `s.empty() == true`

2	`String()`	Множественное создание, вычисление длины строки	`String s2a;`
			`String s2b;`
			`String s2c;`
			`s2a = "a";`
			`s2b = "bb";`
			`s2c = "ccc";`
				1
				2
				3

3	`String(const char*)`	Пустой литерал	`String s("");`	`length() == 0`, `data_[0] == '\0'`
4	`String(const char*)`	Обычная строка	`"Hello"`	`length() == 5`, последние символы совпадают
5	`String(const char*)`	Длинная строка	Строка длиной 1000+ символов	Корректная работа, нет переполнений
6	`String(const String&)`	Копия непустой строки	`String a("abc");` `String b(a);`	`b == a`, но `b.data_ != a.data_`
7	`String(const String&)`	Копия пустой строки	`String a; String b(a);`	`b.empty() == true`
8	`String(String&&)`	Перемещение временного объекта	`String b(String("abc"));`	`b == "abc"`, у источника `length_ == 0`, `data_` в безопасном состоянии
9	`String(String&&)`	Перемещение непустой строки	`String a("abc");` `String b(std::move(a));`	`b == "abc"`, `a` в валидном «пустом» состоянии
10	`length()`	Пустая строка	`String s;`	`length() == 0`
11	`length()`	Непустая строка	`"Test"`	`length() == 4`
12	`empty()`	Пустая строка	`String s;` или `String s("");`	`empty() == true`
13	`empty()`	Непустая строка	“a”	`empty() == false`

14	`operator=(const String&)`	Присваивание между разными объектами	`a = "abc";`
			`b = "x";`
			`b = a;`
				`b == "abc"`, старый буфер `b` освобождён

15	`operator=(const String&)`	Самоприсваивание	`a = "abc";` `a = a;`	Состояние не меняется, нет падения

16	`operator=(String&&)`	Присваивание перемещением	`a = "abc"; b = std::move(a);`	`b == "abc"`, а в «пустом» состоянии
17	`operator=(const char*)`	Присваивание пустой С-строки	`s = "hello"; s = "";`	`s.empty() == true`
18	`operator=(const char*)`	Длинная строка	`s = "a"; s = длинная_строка`	Буфер перераспределён, строка корректна
19	`operator[](size_t)`	Первый символ	`String s("abc"); s[0] = 'z';`	Строка `"zbc"`
20	`operator[](size_t)`	Последний символ	`s[length() - 1]`	Меняется только последний символ
21	`operator[](size_t) const`	Чтение	`const String s("abc"); char c = s[1];`	`c == 'b'`
22	`operator+`	Оба непустые	`"ab" + "cd"`	`"abcd"`
23	`operator+`	Левая пустая	`String() + "abc"`	`"abc"`
24	`operator+`	Правая пустая	`"abc" + String()`	`"abc"`
25	`operator+=(const String&)`	Добавление к непустой	`"ab" += "cd"`	`"abcd"`
26	`operator+=(const String&)`	Добавление пустой строки	`"ab" += ""`	`"ab"`
27	`operator+=(const char*)`	Добавление С-строки	`"ab" += "cde"`	`"abcde"`
28	`clear()`	Очищение непустой строки	`s = "abc"; s.clear();`	`s.empty() == true`, `data_[0] == '\0'`
29	`clear()`	Многократный вызов	`s = "abc"; s.clear(); s.clear();`	Корректная работа, без падения
30	`operator==`	Равные строки	`"abc" == "abc"`	`true`
31	`operator==`	Разные строки	`"abc" == "abd"`	`false`
32	`operator!=`	Разные строки	`"abc" != "abd"`	`true`
33	`operator<`	Лексикографическое сравнение	`"abc" < "abd"`	true
34	`operator<`	Префикс	`"ab" < "abc"`	true
35	`operator<=, >=`	Равные строки	`"abc" <= "abc"`	`true`
36	`operator>>`	Ввод одного слова	ввод: `hello`⏎	Строка `"hello"`
37	`operator>>`	Ввод до пробела	ввод: `hello world`⏎	Строка `"hello"`, остаток во входном потоке
38	`operator>>`	Пустой ввод	нажатие `Enter` сразу	Строка пустая или предыдущее значение

39	`operator<<`	Вывод непустой	`String("abc")`	На экране `"abc"` без лишних символов
40	`operator<<`	Вывод пустой	`String()`	Пустая строка, только перевод строки (если добавить)
41	`lower()`	Смешанный регистр	`"AbC1!"`	`"abc1!"`
42	`lower()`	Нет латинских букв	`"123_+-"`	Строка без изменений
43	`lower()`	Уже нижний регистр	`"abc"`	Строка без изменений
44	`my::strlen`	Пустая С-строка	`""`	`0`
45	`my::strlen`	Обычная строка	`"abcd"`	`4`
46	`my::strcpy`	Копирование обычной строки	`dest` достаточно велик, `src "abc"`	В `dest "abc\0"`
47	`my::strcpy`	Копирование пустой строки	`src ""`	В `dest` сразу `'\0'`
48	`my::strcpy`	Самокопирование (`dest == src`)	Особый случай, обычно не используется	Поведение должно быть корректным или явно запрещено