

СОДЕРЖАНИЕ

ПОСТАНОВКА ЗАДАЧИ	3
ВВЕДЕНИЕ	3
СТРУКТУРА ПРОЕКТА	4
РЕАЛИЗАЦИЯ	4
1 struct StringValue	4
1.1 Приватные поля	4
1.2 Конструктор и деструктор	5
1.3 Методы управления данными	5
2 class String	6
2.1 Приватные поля	6
2.2 Конструкторы и деструктор	6
2.3 Операции доступа к элементам	7
2.4 Операции изменения размера и емкости	7
2.5 Другие методы и операции	8
2.6 Итераторы	8
UNIT-ТЕСТИРОВАНИЕ	9
ЗАКЛЮЧЕНИЕ	12
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	13
ПРИЛОЖЕНИЕ	15
1 Исходный код заголовочного файла (string.hpp)	15
2 Исходный код unit-тестов (string.cpp)	19

ПОСТАНОВКА ЗАДАЧИ

Целью данного проекта было разработать класс String на языке программирования C++, который поддерживает механизм copy-on-write.

Основные задачи включали в себя:

1. Реализация класса String, обеспечивающего хранение строковых данных и поддерживающего операции копирования, перемещения и изменения.
2. Внедрение механизма copy-on-write для оптимизации операций копирования и передачи строк между объектами.
3. Создание unit-тестов для проверки корректности работы класса String и его методов.

ВВЕДЕНИЕ

Класс String представляет собой динамический массив символов, который используется для хранения строк. Он обеспечивает эффективное управление памятью и поддерживает основные операции работы со строками, такие как конкатенация, изменение размера, вставка и удаление символов.

Copy-on-write (COW) - это техника оптимизации памяти, которая позволяет избежать лишних копирований данных при операциях копирования объектов. Вместо того, чтобы немедленно создавать копию данных, когда объект копируется, используется механизм разделения данных, и копирование происходит только в случае, если объекты начинают изменяться.

СТРУКТУРА ПРОЕКТА

Предложенная структура проекта соответствует структуре **The Pitchfork Layout (Merged header placement + Separate test placement)**.

```
rgr-grigorev-is142s03/
├── external/
│   └── googletest/...
├── src/
│   └── libcsc/
│       └── string.hpp    // Реализация класса String
└── test/
    └── libcsc/
        └── string.cpp    // Unit-тесты для класса String
```

РЕАЛИЗАЦИЯ КЛАССА STRING

Для реализации строки с механизмом copy-on-write была определена вложенная структура `StringValue`, указатель на которую лежит в секции `private` класса `String`.

1 struct `StringValue`

1.1 Приватные поля

1. **int refCount**: Хранит количество ссылок на данную строку. Используется для реализации механизма copy-on-write: если `refCount` больше 1, то

данные считаются общими и не могут быть изменены напрямую. Вместо этого создается копия данных.

2. **char *data**: Указатель на массив символов, который содержит собственно данные строки. Данные строки хранятся как массив символов типа `char`.
3. **size_t capacity**: Поле, которое хранит емкость выделенной памяти для строки. Оно определяет максимальное количество символов, которое может содержать строка без повторного выделения памяти.
4. **size_t size**: Поле, которое хранит текущий размер строки, то есть количество символов, которые на данный момент используются в строке. Поле `size` помогает отслеживать реальное количество символов в строке без необходимости прохода по всему массиву данных при каждом обращении к строке.

1.2 Конструктор и деструктор

1. **explicit StringValue(const char *d = "", size_t cap = 0)**: Конструктор класса `StringValue`, который инициализирует данные строки. Принимает указатель на строку `d` и ее емкость `cap`. Если емкость не указана, она вычисляется автоматически на основе длины строки.
2. **~StringValue()**: Деструктор класса `StringValue`, который освобождает выделенную память для данных строки.

1.3 Методы управления данными

1. **void detach()**: Метод, который отвечает за копирование данных в случае, если их счетчик ссылок больше 1. При этом создается новый экземпляр данных с собственной копией строки.

2. **void resize(size_t newSize, char filled = '\0')**: Метод изменения размера данных строки. При необходимости выделяется новая память, данные копируются в нее, а остаток заполняется символом filled.

2 class String

2.1 Приватные поля

StringValue *value: Указатель на экземпляр структуры StringValue, который содержит собственно данные строки. Используется для управления данными и реализации механизма copy-on-write. При создании нового объекта String он ссылается на существующий объект StringValue, а при изменении данных создается копия StringValue.

2.2 Конструкторы и деструктор

1. **explicit String(const char *d = "")**: Конструктор, создает объект класса String из переданной строки d, инициализируя его данными из StringValue. Если аргумент не передан, создается пустая строка.
2. **String(String &&rhs) noexcept**: Конструктор перемещения переносит ресурсы из одного объекта String в другой, обновляя указатель value на StringValue, и обнуляя указатель rhs.value для предотвращения двойного освобождения памяти.
3. **String &operator=(String &&rhs) noexcept**: Оператор перемещающего присваивания позволяет переместить ресурсы из одного объекта String в другой, обновляя указатель value на StringValue.
4. **String(const String &rhs)**: Конструктор копирования создает глубокую копию объекта String, увеличивая счетчик ссылок на общие данные в StringValue.
5. **String &operator=(const String &rhs)**: Оператор копирующего присваивания выполняет глубокое копирование объекта String, обновляя

счетчик ссылок и освобождая старые данные, если необходимо, в `StringValue`.

6. **~String()**: Деструктор освобождает выделенную память, уменьшая счетчик ссылок на общие данные в `StringValue` и освобождая их при необходимости.

2.3 Операции доступа к элементам

1. **char &operator[](size_t index)**: Оператор доступа к символу по индексу для изменения строки. Метод вызывает `value->detach()`, чтобы гарантировать уникальность данных, а затем возвращает ссылку на символ по заданному индексу.
2. **const char &operator[](size_t index) const**: Оператор доступа к символу по индексу для чтения строки, аналогичный предыдущему, но для константного объекта.
3. **char &at(size_t index)**: Метод доступа к символу с проверкой границ для изменения строки. Вызывает `value->detach()` для обеспечения уникальности данных и проверяет, не выходит ли индекс за границы строки. Если выходит, вызывается исключение `std::out_of_range`.
4. **const char &at(size_t index) const**: Метод доступа к символу с проверкой границ для чтения строки, аналогичный предыдущему, но для константного объекта.

2.4 Операции изменения размера и емкости

1. **void resize(size_t newSize, char filled = '\0')**: Метод изменения размера строки до `newSize`. При необходимости заполняет новые элементы символом `filled`.

2. **void reserve(size_t newCap):** Метод резервирования памяти для заданной емкости строки.
3. **void push_back(char c):** Метод добавления символа в конец строки. При необходимости увеличивает емкость строки.
4. **void insert(size_t pos, const char *str):** Метод вставки строки str в указанную позицию pos в строке.
5. **void erase(size_t pos, size_t len):** Метод удаления len символов, начиная с позиции pos в строке.

2.5 Другие методы и операции

1. **bool empty() const:** Метод проверки на пустоту строки.
2. **size_t size() const:** Метод получения размера строки.
3. **size_t capacity() const:** Метод получения емкости строки.
4. **void shrink_to_fit():** Метод уменьшения емкости строки до ее фактического размера.
5. **const char *c_str() const:** Метод получения указателя на массив символов строки.
6. **iterator begin():** Метод получения итератора на начало строки.
7. **const_iterator begin() const:** Метод получения константного итератора на начало строки.
8. **iterator end():** Метод получения итератора на конец строки.
9. **const_iterator end() const:** Метод получения константного итератора на конец строки.

2.6 Итераторы

Класс **String::iterator** реализует функционал итератора для обхода элементов строки. Поддерживаются операции инкремента, декремента,

сравнения, а также арифметика указателей. Категорией итератора для класса String был выбран **std::random_access_iterator_tag**, так как итератор строкового класса обладает всеми свойствами, которые характерны для итераторов произвольного доступа:

1. **Произвольный доступ к элементам:** Итератор класса String позволяет выполнять операции произвольного доступа, такие как смещение вперед и назад на произвольное количество элементов.
2. **Арифметика указателей:** Итератор класса String поддерживает арифметические операции над итераторами, такие как сложение и вычитание, что характерно для итераторов произвольного доступа.
3. **Сравнение итераторов:** Итератор класса String может быть сравнен с другими итераторами для определения их относительного порядка в строке.
4. **Быстрый доступ к элементам:** Обеспечен доступ к каждому отдельному элементу строки за время $O(1)$ (с помощью оператора []).
5. **Поддержка операций инкремента и декремента:** Поддерживает как префиксные, так и постфиксные операции инкремента и декремента, что позволяет удобно перемещаться по элементам строки.

UNIT-ТЕСТИРОВАНИЕ

Для проекта были написаны unit-тесты с использованием библиотеки **GoogleTest**. Далее идет перечисление и описание всех реализованных тестов.

1. Конструкторы (**StringTest.Constructors**): Проверяет корректность работы конструкторов класса String. В условиях теста создаются объекты String с различными аргументами конструктора. После создания объектов проверяется их корректное состояние: пустота, размер и содержимое.

2. Операции перемещения (**StringTest.MoveOperations**): Проверяет правильность работы операций перемещения (move constructor и move assignment) в классе String. В тесте создаются объекты String, которые затем перемещаются в другие объекты. Условия проверки включают проверку размера и содержимого перемещенных объектов, а также корректность их состояния после перемещения.
3. Операции копирования (**StringTest.CopyOperations**): Проверяет корректность работы операций копирования (copy constructor и copy assignment) в классе String. В тесте создаются объекты String, которые затем копируются в другие объекты. Проверяется правильность копирования данных, размер и содержимое объектов, а также их состояние после копирования.
4. Оператор индекса (**StringTest.IndexOperator**): Проверяет корректность работы оператора индексации (operator[]) в классе String. Создается объект String, после чего проверяется доступ к символам строки по индексу и их корректность.
5. Метод at() (**StringTest.AtMethod**): Проверяет работу метода at() класса String, который обеспечивает доступ к символам с проверкой границ массива. В условиях теста производятся попытки доступа к символам строки по различным индексам, включая индексы за пределами строки, и проверяется возникновение исключения std::out_of_range.
6. Методы size(), capacity(), reserve() (**StringTest.SizeCapacityReserve**): Проверяет методы size(), capacity() и reserve() класса String. Создается объект String, изменяется его емкость с помощью метода reserve(), после чего проверяется корректность емкости и размера строки.
7. Метод empty() (**StringTest.Empty**): Проверяет корректность метода empty() класса String, который определяет, является ли строка пустой. Создается объект String, затем проверяется его пустота и добавляется символ, чтобы проверить изменение состояния после добавления.

8. Метод `shrink_to_fit()` (**StringTest.ShrinkToFit**): Проверяет метод `shrink_to_fit()` класса `String`, который уменьшает емкость строки до фактического размера. Создается объект `String` с избыточной емкостью, затем применяется метод `shrink_to_fit()`, и проверяется, что емкость уменьшилась до фактического размера.
9. Метод `resize()` (**StringTest.ResizeMethod**): Проверяет метод `resize()` класса `String`, который изменяет размер строки и заполняет новые символы указанным значением. Создается объект `String`, изменяется его размер с помощью метода `resize()`, после чего проверяется корректность размера и заполнение новых символов.
10. Метод `push_back()` (**StringTest.PushBack**): Проверяет метод `push_back()` класса `String`, который добавляет символ в конец строки. Создается объект `String`, добавляется символ, после чего проверяется корректность размера и содержимого строки.

ЗАКЛЮЧЕНИЕ

В ходе данной работы был разработан класс String на языке C++, реализующий строковый контейнер с механизмом copy-on-write. Реализация этого класса включает в себя основные операции работы со строками, включая конструкторы, операции копирования и перемещения, доступ к символам, изменение размера и емкости строки, вставку и удаление символов, а также итеративный доступ к элементам. Механизм copy-on-write предназначен для оптимизации работы с копиями строк и уменьшения накладных расходов при операциях копирования и присваивания. Также он позволяет избежать лишних копирований данных, когда строки разделяют общие данные до их изменения.

В ходе разработки были созданы также unit-тесты для класса String, которые покрывают основные операции и подтверждают корректность работы реализованных методов. Также важно отметить, что разработанный класс String успешно проходит весь pipeline на GitLab кафедры ВС СибГУТИ, что подтверждает его работоспособность и соответствие установленным требованиям. Это гарантирует, что класс можно успешно использовать в реальных проектах, обеспечивая эффективное и надежное управление строковыми данными. В pipeline также осуществляются проверки на утечки памяти с помощью инструмента Valgrind.

Таким образом, разработанный класс String представляет собой гибкий и эффективный инструмент для работы со строками в языке C++, который обеспечивает высокую производительность и надежность операций над строками.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Copy-On-Write example // CPlusPlus.com URL:
<https://cplusplus.com/forum/beginner/81321/> (дата обращения: 28.02.2024).
2. C++ String Handling // Wikipedia URL:
https://en.wikipedia.org/wiki/C%2B%2B_string_handling (дата обращения: 29.02.2024).
3. Implementing a copy on write String class using reference counting in C++ // StackExchange Code Review URL:
<https://codereview.stackexchange.com/questions/175286/implementing-a-copy-on-write-string-class-using-reference-counting-in-c> (дата обращения: 29.02.2024).
4. The Dark Side of C++ - Copy-On-Write // YouTube URL:
<https://www.youtube.com/watch?v=9b9hWIH8-hE> (дата обращения: 01.03.2024).
5. Confusion about Copy-On-Write and shared_ptr // StackOverflow URL:
<https://stackoverflow.com/questions/6245235/confusion-about-copy-on-write-and-shared-ptr> (дата обращения: 01.03.2024).
6. Copy-On-Write Pointer // GitHub URL:
<https://github.com/HadrienG2/copy-on-write-ptr> (дата обращения: 02.03.2024).
7. Implementing Copy-On-Write // StackExchange - Software Engineering URL:
<https://softwareengineering.stackexchange.com/questions/360130/implementing-copy-on-write> (дата обращения: 02.03.2024).
8. C++ - Велосипедостроение для профессионалов // Habr URL:
<https://habr.com/ru/companies/oleg-bunin/articles/352280/> (дата обращения: 04.03.2024).
9. Why COW was deemed ungood for std::string // GitHub Gist URL:
<https://gist.github.com/alf-p-steinbach/c53794c3711eb74e7558bb514204e755> (дата обращения: 03.03.2024).

10. Legality of COW std::string implementation in C++11 // StackOverflow URL:
<https://stackoverflow.com/questions/12199710/legalty-of-cow-stdstring-impleme-ntation-in-c11> (дата обращения: 04.03.2024).
11. Conceptual COW string implementation // GitHub Gist URL:
<https://gist.github.com/Manu343726/02287de75bb24f2cef00> (дата обращения: 05.03.2024).
12. Optimize String Use: A Case Study // O'Reilly URL:
<https://www.oreilly.com/library/view/optimized-c/9781491922057/ch04.html>
(дата обращения: 07.03.2024).

ПРИЛОЖЕНИЕ

1 Исходный код заголовочного файла (string.hpp)

```
#pragma once

#include <algorithm>
#include <cstdint>
#include <cstring>
#include <stdexcept>

namespace csc {

class String {
private:
    struct StringValue {
        int refCount{1};
        char *data;
        size_t capacity;
        size_t size;

        explicit StringValue(const char *d = "",
            size_t cap = 0)
            : capacity(cap), size(strlen(d)) {
            capacity = std::max(cap, size + 1); // +
            null-terminator
            data = new char[capacity];
            strcpy(data, d, capacity);
        }

        ~StringValue() { delete[] data; }

        void detach() {
            if (refCount > 1) {
                --refCount;
                char *newData = new char[size + 1];
                strcpy(newData, data, size + 1);
                data = newData;
            }
        }

        void resize(size_t newSize, char filled =
            '\0') {
            if (newSize > capacity) {
                char *newData = new char[newSize + 1];
                strcpy(newData, data, size);
                delete[] data;
                data = newData;
                for (size_t i = size; i < newSize;
                    ++i) {
                        newData[i] = filled;
                    }
            }
        }

        // Erase
        void erase(size_t pos, size_t len) {
            if (pos >= value->size) {
                throw std::out_of_range("Position out of
                    range");
            }
            if (pos + len > value->size) {
                len = value->size - pos; // Correct len
            }
            value->detach();
            // Push elements from right to the
            beginning of str
            for (size_t i = pos; i + len <
                value->size; ++i) {
                value->data[i] = value->data[i + len];
            }
            value->size -= len;
            value->data[value->size] = '\0';
        }

        // Empty
        bool empty() const { return value->size ==
            0; }

        // Size
        size_t size() const { return value->size; }

        // Capacity
        size_t capacity() const { return
            value->capacity; }

        // Shrink to fit
        void shrink_to_fit() {
            if (value->capacity > value->size) {
                value->detach();
                char *newData = new char[value->size +
                    1];
                strcpy(newData, value->data,
                    value->size + 1);
                delete[] value->data;
                value->data = newData;
                value->capacity = value->size;
            }
        }

        // C string
        const char *c_str() const { return
            value->data; }
    };
};
```

```

        capacity = newSize;
    } else if (newSize < size) {
        for (size_t i = newSize; i < size;
++i) {
            data[i] = filled; // fill to new
size
        }
    }
    size = newSize;
    data[size] = '\0';
}

} *value;

public:
    // Constructor from const char*
    explicit String(const char *d = "") { value =
new StringValue(d); }

    // Move constructor
    String(String &&rhs) noexcept :
value{rhs.value} { rhs.value = nullptr; }

    // Move assignment
    String &operator=(String &&rhs) noexcept {
        if (this != &rhs) {
            delete value; // Free cur object
            value = rhs.value; // Move data
            rhs.value = nullptr;
        }
        return *this;
    }

    // Copy constructor
    String(const String &rhs) : value(rhs.value)
{
    ++value->refCount; // Just new reference,
no allocators
}

    // Copy assignment
    String &operator=(const String &rhs) {
        if (this != &rhs) {
            if (--value->refCount == 0) {
                delete value;
            }
            value = rhs.value;
            ++value->refCount;
        }
        return *this;
    }

    // Destructor
    ~String() {

```

```

// Iterator class
class iterator {
private:
    char *ptr;

public:
    using iterator_category =
std::random_access_iterator_tag;
    using difference_type = std::ptrdiff_t;
    using value_type = char;
    using pointer = char *;
    using reference = char &;

    explicit iterator(char *p) : ptr(p) {}

    // Dereferencing
    reference operator*() const { return *ptr;
}

    // Increment
    iterator &operator++() {
        ++ptr;
        return *this;
    }

    // Post-increment
    const iterator operator++(int) {
        iterator temp = *this;
        ++ptr;
        return temp;
    }

    // Decrement
    iterator &operator--() {
        --ptr;
        return *this;
    }

    // Post-decrement
    const iterator operator--(int) {
        iterator temp = *this;
        --ptr;
        return temp;
    }

    // Compound assignment +=
    iterator &operator+=(difference_type n) {
        ptr += n;
        return *this;
    }

    // Compound assignment -=
    iterator &operator-=(difference_type n) {

```



```

    if (value != nullptr && --value->refCount
== 0) {
        delete value;
    }
}

// Operator[]
char &operator[](size_t index) {
    value->detach();
    return value->data[index];
}

// Const operator[]
const char &operator[](size_t index) const {
    return value->data[index]; }

// At
char &at(size_t index) {
    if (index >= value->size) {
        throw std::out_of_range("Index out of
range");
    }
    value->detach();
    return value->data[index];
}

// Const at
const char &at(size_t index) const {
    if (index >= value->size) {
        throw std::out_of_range("Index out of
range");
    }
    return value->data[index];
}

// Resize
void resize(size_t newSize, char filled =
'\0') {
    value->detach();
    value->resize(newSize, filled);
}

// Reserve
void reserve(size_t newCap) {
    if (newCap > value->capacity) {
        value->detach();
        // Create new buf with new capacity,
copy data
        char *newData = new char[newCap + 1];
        strcpy(newData, value->data, newCap +
1);
        delete[] value->data;
        value->data = newData;
        value->capacity = newCap;
    }
}

```

```

    ptr -= n;
    return *this;
}

// Subscripting
reference operator[](difference_type n)
const { return *(ptr + n); }

// Addition
friend iterator operator+(iterator it,
difference_type n) {
    return iterator(it.ptr + n);
}

// Subtraction
friend iterator operator-(iterator it,
difference_type n) {
    return iterator(it.ptr - n);
}

// Equality
friend bool operator==(const iterator &lhs,
const iterator &rhs) {
    return lhs.ptr == rhs.ptr;
}

// Inequality
friend bool operator!=(const iterator &lhs,
const iterator &rhs) {
    return !(lhs == rhs);
}

// Less than
friend bool operator<(const iterator &lhs,
const iterator &rhs) {
    return lhs.ptr < rhs.ptr;
}

// Greater than
friend bool operator>(const iterator &lhs,
const iterator &rhs) {
    return lhs.ptr > rhs.ptr;
}

// Less than or equal to
friend bool operator<=(const iterator &lhs,
const iterator &rhs) {
    return lhs.ptr <= rhs.ptr;
}

// Greater than or equal to
friend bool operator>=(const iterator &lhs,
const iterator &rhs) {
    return lhs.ptr >= rhs.ptr;
}

```

<pre> } } // Push back void push_back(char c) { if (value->size + 1 >= value->capacity) { reserve(value->size * 2 + 2); } value->data[value->size++] = c; value->data[value->size] = '\0'; } // Insert void insert(size_t pos, const char *str) { if (pos > value->size) { throw std::out_of_range("Position out of range"); } size_t len = strlen(str); if (len + value->size > value->capacity) { reserve(len + value->size); } value->detach(); // Push existing elements to the right for (size_t i = value->size; i >= pos; --i) { value->data[i + len] = value->data[i]; } // Copy new data in pos for (size_t i = 0; i < len; ++i) { value->data[pos + i] = str[i]; } value->size += len; value->data[value->size] = '\0'; } </pre>	<pre> }; // Const iterator using const_iterator = const iterator; // Begin method iterator begin() { value->detach(); return iterator(value->data); } // Const begin method const const_iterator begin() const { return const_iterator(value->data); } // End method iterator end() { value->detach(); return iterator(value->data + value->size); } // Const end method const const_iterator end() const { return const_iterator(value->data + value->size); } }; } // namespace csc </pre>
--	--

2 Исходный код unit-тестов (string.cpp)

<pre> #include <libcsc/string.hpp> #include <gtest/gtest.h> #include <utility> using namespace csc; TEST(StringTest, Constructors) { String s1; EXPECT_TRUE(s1.empty()); String s2("hello"); EXPECT_EQ(s2.size(), 5); } </pre>	<pre> TEST(StringTest, PushBack) { String s; s.push_back('a'); EXPECT_EQ(s.size(), 1); EXPECT_EQ(s[0], 'a'); } TEST(StringTest, Insert) { String s("hello"); s.insert(5, " world"); EXPECT_EQ(s.size(), 11); EXPECT_EQ(s[6], 'w'); } </pre>
--	--

```

}

TEST(StringTest, MoveOperations) {
    // Create the original strings
    String s1("test1");
    String s2("test2");

    // Move the strings
    String moved1(std::move(s1));
    String moved2 = std::move(s2);

    // Check the properties of the moved-to
    objects
    EXPECT_EQ(moved1.size(), 5);
    EXPECT_EQ(moved2.size(), 5);
}

TEST(StringTest, CopyOperations) {
    String s1("test1");
    const String &copy1(s1);
    EXPECT_EQ(copy1.size(), s1.size());
    String s2("test2");
    const String &copy2 = s2;
    EXPECT_EQ(copy2.size(), s2.size());
}

TEST(StringTest, IndexOperator) {
    String s("hello");
    EXPECT_EQ(s[1], 'e');
}

TEST(StringTest, AtMethod) {
    String s("test");
    EXPECT_EQ(s.at(2), 's');
    EXPECT_THROW(s.at(5), std::out_of_range);
}

TEST(StringTest, SizeCapacityReserve) {
    String s1;
    s1.reserve(10);
    EXPECT_GE(s1.capacity(), 10);
    String s2("hello");
    EXPECT_EQ(s2.size(), 5);
    EXPECT_GE(s2.capacity(), 5);
}

TEST(StringTest, Empty) {
    String s;
    EXPECT_TRUE(s.empty());
    s.push_back('a');
    EXPECT_FALSE(s.empty());
}

TEST(StringTest, Erase) {
    String s("hello world");
    s.erase(5, 6);
    EXPECT_EQ(s.size(), 5);
}

TEST(StringTest, BeginEnd) {
    String s("test");
    EXPECT_EQ(*s.begin(), 't');
    EXPECT_EQ(*(s.end() - 1), 't');
}

TEST(StringTest, IteratorTraversal) {
    String s("hello");
    String::iterator it = s.begin();
    String::iterator end = s.end();

    // Test forward traversal
    int count = 0;
    while (it != end) {
        ++count;
        ++it;
    }
    EXPECT_EQ(count, 5); // 'hello' has 5
    characters
}

TEST(StringTest, IteratorDereferencing) {
    String s("hello");
    String::iterator it = s.begin();

    // Test dereferencing
    EXPECT_EQ(*it, 'h');
    ++it;
    EXPECT_EQ(*it, 'e');
}

TEST(StringTest, IteratorComparison) {
    String s("hello");
    String::iterator it1 = s.begin();
    String::iterator it2 = s.begin();
    String::iterator end = s.end();

    // Test iterator equality and inequality
    EXPECT_EQ(it1, it2);
    ++it1;
    EXPECT_NE(it1, it2);

    // Test iterator comparison
    EXPECT_LT(it2, it1);
    EXPECT_LE(it2, it1);
    EXPECT_GT(it1, it2);
}

```

```

TEST(StringTest, ShrinkToFit) {
    String s("hello");
    s.reserve(20);
    EXPECT_GE(s.capacity(), 20);
    s.shrink_to_fit();
    EXPECT_EQ(s.capacity(), 5);
}

TEST(StringTest, ResizeMethod) {
    String s1("hello");
    s1.resize(3);
    EXPECT_EQ(s1.size(), 3);
    std::cerr << s1.c_str() << '\n';
    String s2("h");
    s2.resize(5, 'i');
    EXPECT_EQ(s2.size(), 5);
    EXPECT_EQ(s2[4], 'i');
}

```

```

EXPECT_GE(it1, it2);

// Test iterator comparison with end iterator
while (it1 != end) {
    ++it1;
}
EXPECT_EQ(it1, end);
EXPECT_NE(it2, end);
}

int main(int argc, char *argv[]) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```