

Федеральное государственное бюджетное образовательное учреждение высшего  
образования  
«Сибирский государственный университет телекоммуникаций и информатики»  
(СибГУТИ)

Кафедра вычислительных систем

Расчетно-графическая работа  
по дисциплине «Современные технологии программирования»  
на тему «Разработка строкового контейнера с механизмом Copy-on-write»

Выполнил:  
ст. гр. ИС-142  
Григорьев Ю.В.

Проверил:  
доц. Пименов Е. С.

Новосибирск 2024

## Содержание

1. Постановка задачи	3
2. Мотивация	4
3. Реализация	5
3.1 Внутреннее устройство	5
3.2 Интерфейс	5
3.3 Итераторы	6
4. Модульное тестирование	8
5. Список источников	10
6. Приложение	11

## 1. Постановка задачи

Спроектировать строковый контейнер `String` с механизмом Copy-on-write. Использовать стандарт языка C++17. Реализовать итераторы, совместимые с алгоритмами стандартной библиотеки. Покрыть модульными тестами. При реализации не пользоваться контейнерами стандартной библиотеки, реализовать управление ресурсами в идиоме RAII.

В качестве системы сборки использовать CMake. Структурировать проект в соответствии с соглашениями The Pitchfork Layout (Merged header placement + Separate test placement). Всю разработку вести в системе контроля версий git. Настроить автоматическое форматирование средствами clang-format.

Проверить код анализаторами Valgrind Memcheck, undefined sanitizer, address sanitizer, clang-tidy.

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

Рисунок 1. Демонстрация структуры проекта по The Pitchfork Layout (MH + ST)

## **2. Мотивация**

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.

## 3. Реализация

### 3.1 Внутреннее устройство

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

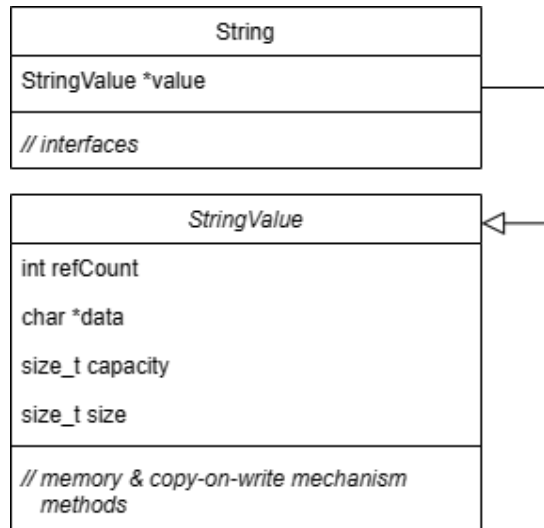


Рисунок 2. Структура внутреннего устройства реализованного строкового контейнера

### 3.2 Интерфейс

**struct `StringValue`:**

**Конструктор `StringValue(const char *d = "", size_t cap = 0)`:** Создает объект `StringValue`, инициализируя его данными из строки `d`.

Сложность:  $O(n)$ , где  $n$  - длина строки `d`.

Создает указатель, на который ссылаются итераторы класса `String`.

**Метод `detach()`:** Обеспечивает копирование данных, если на объект `StringValue` ссылается более одного объекта `String`.

Сложность:  $O(n)$ , где  $n$  - размер строки.

Приводит к изменению указателей, на которые ссылаются итераторы конкретного экземпляра класса `String`.

**Деструктор `~StringValue()`:** Освобождает выделенную память для хранения данных строки.

Сложность:  $O(1)$ .

## **class String:**

### **Конструкторы и деструктор:**

Сложность: **O(1)**, за исключением конструктора от строки, где происходит создание объекта `StringValue`, сложность которого **O(n)**.

### **Методы работы с элементами строки** (`operator[]`, `at`, `push_back`, `insert`, `erase`):

Сложность: **O(1)**, за исключением операций `insert` и `erase`, которые могут иметь сложность до **O(n)** в случае перемещения элементов внутри строки.

Могут привести к изменению размера строки и, следовательно, к изменению позиций элементов в памяти, что влияет на итераторы.

### **Методы работы с емкостью и размером строки** (`resize`, `reserve`, `shrink_to_fit`):

Сложность: **O(1)**, за исключением операции `resize`, которая может иметь сложность до **O(n)** при увеличении размера строки.

Могут изменять размер строки, что повлияет на позиции элементов в памяти и, следовательно, на итераторы.

### **Методы работы с итераторами** (`begin`, `end`, операторы инкремента/декремента):

Сложность: **O(1)**.

Позволяют эффективно перемещаться по строке, но при изменении размера или емкости строки позиции элементов в памяти могут измениться, что повлияет на итераторы.

## **3.3 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 `[]`).

**Поддержка операций инкремента и декремента:** Поддерживает как префиксные, так и постфиксные операции инкремента и декремента, что позволяет удобно перемещаться по элементам строки.

## 4. Модульное 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.
11. Совместимость с `std::find()` (**StringTest.Find**): Проверяет совместимость класса `String` с функцией `std::find()` из C++ STL для нескольких символов (присутствующих и не присутствующих) в строке.

## 5. СПИСОК 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](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/legality-of-cow-stdstring-implementation-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).

## 6. Приложение

file src/libcsc/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

```

```

// 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) {

```

```

~String() {
    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;

```

```

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;
}

```

```

        value->capacity = newCap;
    }
}

// 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';
}

};

// 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

```

## file test/libcsc/string.cpp

```

#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");

    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');
    }
}

```

```

    EXPECT_EQ(s2.size(), 5);
}

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);
}

```

```

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_GT(it1, it2);
EXPECT_GE(it1, it2);

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

TEST(StringTest, Find) {
    String s("hi");
    EXPECT_NE(std::find(s.begin(), s.end(), 'i'),
s.end());
    EXPECT_EQ(std::find(s.begin(), s.end(), 'a'),
s.end());
    EXPECT_EQ(*std::find(s.begin(), s.end(),
'h'), 'h');
}

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

```