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 «Реализация шаблонного типа данных String + COW»

Выполнил:

ст. гр. ИВ-222

Рудцких В. Е.

Проверил:

доц. Пименов Е. С.

Содержание

[1. Постановка задачи 3](#__RefHeading___Toc79_531918277)

[2. Мотивация 4](#__RefHeading___Toc851_4100427750)

[3. Реализация 5](#__RefHeading___Toc300_531918277)

[3.1 Внутреннее устройство 5](#__RefHeading___Toc302_531918277)

[3.2 Интерфейс 5](#__RefHeading___Toc304_531918277)

[3.3 Итераторы 5](#__RefHeading___Toc306_531918277)

[4. Список источников 6](#__RefHeading___Toc832_4100427750)

[5. Приложение 7](#__RefHeading___Toc834_4100427750)

## **Постановка задачи**

Спроектировать шаблонный тип данных String + COW. Использовать стандарт языка С⁠+⁠+⁠20. 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. 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.

## Мотивация

Стандартная реализация строки в большинстве языков программирования использует стратегию глубокого копирования (deep copy). Это означает, что при каждой операции копирования строки (например, при присваивании, передаче в функцию по значению) создается полная физическая копия всех данных, включая сам массив символов.

Такой подход может быть крайне неэффективен с точки зрения производительности и потребления памяти, особенно когда:

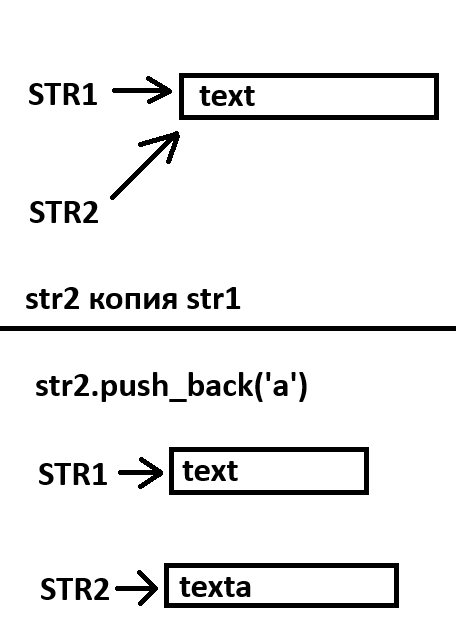
1. Строки имеют большой размер
2. Операции копирования происходят часто, но последующие модификации этих копий нет

Реализация строки с механизмом Copy-on-Write призвана решить эту проблему. При операциях копирования не создается немедленная копия данных. Вместо этого обе строки (исходная и новая) начинают разделять один и тот же массив символов в памяти. При этом Множество объектов-строк могут безопасно ссылаться на один и тот же буфер с данными. Это обеспечивает высокую эффективность по памяти и времени для операций, которые не меняют данные (чтение, поиск, передача как константной ссылки).

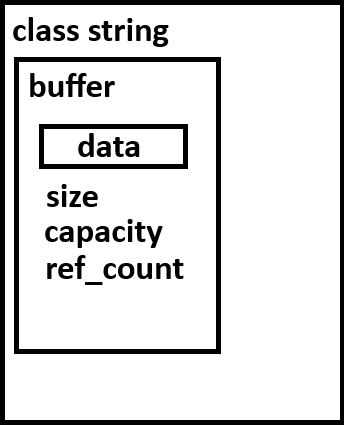
## Реализация

### Внутреннее устройство

Механизм работы copy-on-write заключается в том, чтобы при копировании не создавать полную копию объекта, а реализовать доступ к памяти копируемого предмета. Создание копии происходит только при внесении изменений. Таким образом достигается более эффективное использование ресурсов, см. рисунок 1.

Рисунок 1: Принцип работы Copy-on-write

Устройство класса представляет собой обернутую строку с параметрами для обеспечения копирования без выделения памяти, см. рисунок. 2.

Рисунок 2: Устройство класса

### Интерфейс

Опишите операции, доступные через публичный интерфейс. Укажите их асимптотическую сложность и влияние на итераторы.

**Конструкторы**:

Конструктор по умолчанию string(): O(1) - создает буфер начального размера

Конструктор из C-строки string(const char\*): O(n) - копирование данных строки

Конструктор копирования string(const string&): O(1) - увеличивает счетчик ссылок

Конструктор перемещения string(string&&): O(1) - передача владения буфером

Деструктор: O(1) или O(n) - уменьшает счетчик ссылок, при 0 - удаляет данные

**Операторы присваивания**:

Копирующее присваивание: O(1) - разделение буфера с подсчетом ссылок

Перемещающее присваивание: O(1) - передача владения буфером

**Доступ к элементам**:

operator[]: O(1) - обращение по индексу

at(): O(1) - обращение по индексу с проверкой границ

**Модификация строки**:

push\_back() / operator+=: O(1) амортизированно, может вызвать переаллокацию

insert(): O(n) - Добавление элемента и сдвиг

erase(): O(n) - Удаление элемента и сдвиг

resize(): O(n) - Присвоение размера, может вызвать переаллокацию

reserve(): O(n) - Присвоение вместимости, переаллокация при увеличении capacity

clear(): O(1) - сброс размера

**Информационные методы**:

size() / length() / capacity() / empty(): O(1) - Получение информации

c\_str(): O(1) - Указатель на начало строки (как в си)

### Итераторы

Какие итераторы предоставляет ваша структура данных? Категория, особенности реализации.

Реализованный класс предоставляет итераторы произвольного доступа (std::random\_access\_iterator\_tag). Подобный итератор позволяет получать доступ к любому элементу по индексу и использовать большинство алгоритмов стандартной библиотеки.

**Типы итераторов**:

iterator - изменяющий итератор

const\_iterator - константный итератор

reverse\_iterator - обратный изменяющий итератор

const\_reverse\_iterator - константный обратный итератор

**Особенности реализации в контексте COW**:

1. Неконстантные итераторы вызывают ensure\_unique() - потенциальное отделение буфера
2. Константные итераторы не вызывают отделение - безопасны для чтения
3. Инвалидация: Любая модификация через один итератор влияет на все копии строки

**Обеспечение корректности**:

1. При создании неконстантного итератора проверяется уникальность буфера
2. Если буфер разделен, создается копия данных

Это гарантирует, что модификация через итератор не повлияет на другие копии

## Список источников

1. Kolpackov B. Canonical Project Structure [Электронный ресурс]. URL: https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1204r0.html (дата обращения: 07.05.2022).

2. Copy-on-write [Электронный ресурс]. URL: https://habr.com/ru/articles/673372/ (дата обращения: 20.12.2024).

3. std::basic\_string [Электронный ресурс]. URL: https://en.cppreference.com/w/cpp/string/basic\_string (дата обращения: 20.12.2024).

## Приложение

// file stringCOW.cpp

#include <iostream>

#include <iterator>

#include <algorithm>

#include <stdexcept>

class string { private: static void\* copy(void\* dest, const void\* src, size\_t n) { char\* d = static\_cast<char\*>(dest); const char\* s = static\_cast<const char\*>(src); for (size\_t i = 0; i < n; ++i) d[i] = s[i]; return dest; }

static size\_t getlength(const char\* str) { size\_t len = 0; while (str[len] != '\0') ++len; return len; }

static void\* moving(void\* dest, const void\* src, size\_t n) { char\* d = static\_cast<char\*>(dest); const char\* s = static\_cast<const char\*>(src); if (d == s) return dest; if (d < s) { for (size\_t i = 0; i < n; ++i) d[i] = s[i]; } else { for (size\_t i = n; i > 0; --i) d[i - 1] = s[i - 1]; } return dest; }

static int compare(const void\* ptr1, const void\* ptr2, size\_t n) { const unsigned char\* p1 = static\_cast<const unsigned char\*>(ptr1); const unsigned char\* p2 = static\_cast<const unsigned char\*>(ptr2); for (size\_t i = 0; i < n; ++i) { if (p1[i] != p2[i]) return (p1[i] < p2[i]) ? -1 : 1; } return 0; } struct Buffer { char\* data; size\_t size; size\_t capacity; int ref\_count;

Buffer(size\_t cap = 15) : size(0), capacity(cap), ref\_count(1) {  
 data = new char[capacity + 1];  
 data[0] = '\0';  
}  
  
Buffer(const char\* str, size\_t len)  
 : size(len), capacity(len), ref\_count(1) {  
 data = new char[capacity + 1];  
 string::copy(data, str, len);  
 data[len] = '\0';  
}  
  
~Buffer() { delete[] data; }  
  
void add\_ref() { ++ref\_count; }  
void release() {  
 if (--ref\_count == 0) delete this;  
}  
  
Buffer\* detach() {  
 if (ref\_count > 1) {  
 Buffer\* new\_buf = new Buffer(size);  
 string::copy(new\_buf->data, data, size);  
 new\_buf->size = size;  
 new\_buf->data[size] = '\0';  
 release();  
 return new\_buf;  
 }  
 return this;  
}

};

Buffer\* buf;

void ensure\_unique() { if (buf->ref\_count > 1) { buf = buf->detach(); } }

public: class iterator { private: char\* ptr;

public: using difference\_type = std::ptrdiff\_t; using value\_type = char; using pointer = char\*; using reference = char&; using iterator\_category = std::random\_access\_iterator\_tag;

iterator(char\* p = nullptr) : ptr(p) {}  
  
reference operator\*() const { return \*ptr; }  
pointer operator->() const { return ptr; }  
  
iterator& operator++() {  
 ++ptr;  
 return \*this;  
}  
iterator operator++(int) {  
 iterator tmp = \*this;  
 ++ptr;  
 return tmp;  
}  
iterator& operator--() {  
 --ptr;  
 return \*this;  
}  
iterator operator--(int) {  
 iterator tmp = \*this;  
 --ptr;  
 return tmp;  
}  
  
iterator& operator+=(difference\_type n) {  
 ptr += n;  
 return \*this;  
}  
iterator& operator-=(difference\_type n) {  
 ptr -= n;  
 return \*this;  
}  
  
iterator operator+(difference\_type n) const { return iterator(ptr + n); }  
iterator operator-(difference\_type n) const { return iterator(ptr - n); }  
  
difference\_type operator-(const iterator& other) const {  
 return ptr - other.ptr;  
}  
  
reference operator[](difference\_type n) const { return ptr[n]; }  
  
bool operator==(const iterator& other) const { return ptr == other.ptr; }  
bool operator!=(const iterator& other) const { return ptr != other.ptr; }  
bool operator<(const iterator& other) const { return ptr < other.ptr; }  
bool operator>(const iterator& other) const { return ptr > other.ptr; }  
bool operator<=(const iterator& other) const { return ptr <= other.ptr; }  
bool operator>=(const iterator& other) const { return ptr >= other.ptr; }

};

class const\_iterator { private: const char\* ptr;

public: using difference\_type = std::ptrdiff\_t; using value\_type = const char; using pointer = const char\*; using reference = const char&; using iterator\_category = std::random\_access\_iterator\_tag;

const\_iterator(const char\* p = nullptr) : ptr(p) {}  
  
reference operator\*() const { return \*ptr; }  
pointer operator->() const { return ptr; }  
  
const\_iterator& operator++() {  
 ++ptr;  
 return \*this;  
}  
const\_iterator operator++(int) {  
 const\_iterator tmp = \*this;  
 ++ptr;  
 return tmp;  
}  
const\_iterator& operator--() {  
 --ptr;  
 return \*this;  
}  
const\_iterator operator--(int) {  
 const\_iterator tmp = \*this;  
 --ptr;  
 return tmp;  
}  
  
const\_iterator& operator+=(difference\_type n) {  
 ptr += n;  
 return \*this;  
}  
const\_iterator& operator-=(difference\_type n) {  
 ptr -= n;  
 return \*this;  
}  
  
const\_iterator operator+(difference\_type n) const {  
 return const\_iterator(ptr + n);  
}  
const\_iterator operator-(difference\_type n) const {  
 return const\_iterator(ptr - n);  
}  
  
difference\_type operator-(const const\_iterator& other) const {  
 return ptr - other.ptr;  
}  
  
reference operator[](difference\_type n) const { return ptr[n]; }  
  
bool operator==(const const\_iterator& other) const {  
 return ptr == other.ptr;  
}  
bool operator!=(const const\_iterator& other) const {  
 return ptr != other.ptr;  
}  
bool operator<(const const\_iterator& other) const {  
 return ptr < other.ptr;  
}  
bool operator>(const const\_iterator& other) const {  
 return ptr > other.ptr;  
}  
bool operator<=(const const\_iterator& other) const {  
 return ptr <= other.ptr;  
}  
bool operator>=(const const\_iterator& other) const {  
 return ptr >= other.ptr;  
}

};

using reverse\_iterator = std::reverse\_iterator; using const\_reverse\_iterator = std::reverse\_iterator;

string() : buf(new Buffer()) {}

string(const char\* str) { size\_t len = string::getlength(str); buf = new Buffer(str, len); }

string(const string& other) : buf(other.buf) { buf->add\_ref(); }

string(string&& other) noexcept : buf(other.buf) { other.buf = new Buffer(); }

~string() { buf->release(); }

string& operator=(const string& other) { if (this != &other) { buf->release(); buf = other.buf; buf->add\_ref(); } return \*this; }

string& operator=(string&& other) noexcept { if (this != &other) { buf->release(); buf = other.buf; other.buf = new Buffer(); } return \*this; }

char& operator[](size\_t pos) { ensure\_unique(); return buf->data[pos]; }

const char& operator[](size\_t pos) const { return buf->data[pos]; }

char& at(size\_t pos) { if (pos >= buf->size) { throw std::out\_of\_range("string::at"); } ensure\_unique(); return buf->data[pos]; }

const char& at(size\_t pos) const { if (pos >= buf->size) { throw std::out\_of\_range("string::at"); } return buf->data[pos]; }

iterator begin() { ensure\_unique(); return iterator(buf->data); }

const\_iterator begin() const { return const\_iterator(buf->data); }

const\_iterator cbegin() const { return const\_iterator(buf->data); }

iterator end() { ensure\_unique(); return iterator(buf->data + buf->size); }

const\_iterator end() const { return const\_iterator(buf->data + buf->size); }

const\_iterator cend() const { return const\_iterator(buf->data + buf->size); }

reverse\_iterator rbegin() { return reverse\_iterator(end()); }

const\_reverse\_iterator rbegin() const { return const\_reverse\_iterator(end()); }

const\_reverse\_iterator crbegin() const { return const\_reverse\_iterator(cend()); }

reverse\_iterator rend() { return reverse\_iterator(begin()); }

const\_reverse\_iterator rend() const { return const\_reverse\_iterator(begin()); }

const\_reverse\_iterator crend() const { return const\_reverse\_iterator(cbegin()); }

bool empty() const { return buf->size == 0; }

size\_t size() const { return buf->size; }

size\_t length() const { return buf->size; }

size\_t capacity() const { return buf->capacity; }

void reserve(size\_t new\_cap) { if (new\_cap > buf->capacity) { ensure\_unique(); Buffer\* new\_buf = new Buffer(new\_cap); string::copy(new\_buf->data, buf->data, buf->size); new\_buf->size = buf->size; new\_buf->data[buf->size] = '\0'; buf->release(); buf = new\_buf; } }

void resize(size\_t new\_size, char ch = '\0') { ensure\_unique(); if (new\_size > buf->capacity) { reserve(std::max(new\_size, buf->capacity \* 2)); }

if (new\_size > buf->size) {  
 std::fill(buf->data + buf->size, buf->data + new\_size, ch);  
}  
buf->size = new\_size;  
buf->data[new\_size] = '\0';

}

void shrink\_to\_fit() { if (buf->size < buf->capacity) { ensure\_unique(); Buffer\* new\_buf = new Buffer(buf->size); string::copy(new\_buf->data, buf->data, buf->size); new\_buf->size = buf->size; new\_buf->data[new\_buf->size] = '\0'; buf->release(); buf = new\_buf; } }

void clear() { ensure\_unique(); buf->size = 0; buf->data[0] = '\0'; }

iterator insert(const\_iterator pos, char ch) { size\_t offset = pos - cbegin(); ensure\_unique();

if (buf->size + 1 > buf->capacity) {  
 reserve(std::max(buf->size + 1, buf->capacity \* 2));  
}  
  
string::moving(buf->data + offset + 1, buf->data + offset,  
 buf->size - offset + 1);  
buf->data[offset] = ch;  
++buf->size;  
  
return iterator(buf->data + offset);

}

void push\_back(char ch) { insert(cend(), ch); }

iterator erase(const\_iterator pos) { return erase(pos, pos + 1); }

iterator erase(const\_iterator first, const\_iterator last) { size\_t start = first - cbegin(); size\_t count = last - first;

if (count > 0) {  
 ensure\_unique();  
 string::moving(buf->data + start, buf->data + start + count,  
 buf->size - start - count + 1);  
 buf->size -= count;  
 buf->data[buf->size] = '\0';  
}  
  
return iterator(buf->data + start);

}

string& operator+=(char ch) { push\_back(ch); return \*this; }

string& operator+=(const string& other) { if (other.buf->size > 0) { ensure\_unique(); if (buf->size + other.buf->size > buf->capacity) { reserve(buf->size + other.buf->size); } string::copy(buf->data + buf->size, other.buf->data, other.buf->size); buf->size += other.buf->size; buf->data[buf->size] = '\0'; } return \*this; }

const char\* c\_str() const { return buf->data; }

friend bool operator==(const string& lhs, const string& rhs) { return lhs.size() == rhs.size() && string::compare(lhs.c\_str(), rhs.c\_str(), lhs.size()) == 0; }

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

string operator+(const string& lhs, const string& rhs) { string result = lhs; result += rhs; return result; }

string operator+(const string& str, char ch) { string result = str; result += ch; return result; }

string operator+(char ch, const string& str) { string result; result += ch; result += str; return result; }

// file stringCOWtest.cpp

#include <gtest/gtest.h>

#include <vector>

#include <algorithm>

#include <cstring>

#include <stringCOW/stringCOW.cpp>

TEST(COWStringTest, Constructors) { string s1; EXPECT\_TRUE(s1.empty()); EXPECT\_EQ(s1.size(), 0);

string s2("hello"); EXPECT\_EQ(s2.size(), 5); EXPECT\_STREQ(s2.c\_str(), "hello");

string s3(s2); EXPECT\_EQ(s3.size(), 5); EXPECT\_STREQ(s3.c\_str(), "hello"); }

TEST(COWStringTest, AssignmentAndCOW) { string s1("original"); string s2 = s1;

EXPECT\_STREQ(s1.c\_str(), s2.c\_str());

s2[0] = 'O'; EXPECT\_STREQ(s1.c\_str(), "original"); EXPECT\_STREQ(s2.c\_str(), "Original"); }

TEST(COWStringTest, OperatorBracketAndAt) { string s("test"); EXPECT\_EQ(s[0], 't'); EXPECT\_EQ(s.at(1), 'e');

s[0] = 'T'; EXPECT\_STREQ(s.c\_str(), "Test");

EXPECT\_THROW(s.at(10), std::out\_of\_range); }

TEST(COWStringTest, Iterators) { string s("hello");

std::vector chars(s.begin(), s.end()); std::vector expected = {'h', 'e', 'l', 'l', 'o'}; EXPECT\_EQ(chars, expected);

std::vector reverse\_chars(s.rbegin(), s.rend()); std::vector expected\_reverse = {'o', 'l', 'l', 'e', 'h'}; EXPECT\_EQ(reverse\_chars, expected\_reverse);

\*s.begin() = 'H'; EXPECT\_STREQ(s.c\_str(), "Hello"); }

TEST(COWStringTest, ResizeAndReserve) { string s; s.reserve(100); EXPECT\_GE(s.capacity(), 100); EXPECT\_TRUE(s.empty());

s.resize(5, 'a'); EXPECT\_EQ(s.size(), 5); EXPECT\_STREQ(s.c\_str(), "aaaaa");

s.resize(3); EXPECT\_EQ(s.size(), 3); EXPECT\_EQ(s[0], 'a'); EXPECT\_EQ(s[1], 'a'); EXPECT\_EQ(s[2], 'a'); EXPECT\_EQ(std::strlen(s.c\_str()), 3); }

TEST(COWStringTest, InsertAndErase) { string s("world"); s.insert(s.cbegin(), 'H'); EXPECT\_STREQ(s.c\_str(), "Hworld");

s.insert(s.cbegin() + 1, 'e'); EXPECT\_STREQ(s.c\_str(), "Heworld");

s.insert(s.cbegin() + 2, 'l'); s.insert(s.cbegin() + 3, 'l'); s.insert(s.cbegin() + 4, 'o'); s.insert(s.cbegin() + 5, ' '); EXPECT\_STREQ(s.c\_str(), "Hello world");

s.erase(s.cbegin() + 5, s.cend()); EXPECT\_STREQ(s.c\_str(), "Hello");

s.erase(s.cbegin() + 1); EXPECT\_STREQ(s.c\_str(), "Hllo"); }

TEST(COWStringTest, PushBack) { string s; s.push\_back('h'); s.push\_back('e'); s.push\_back('l'); s.push\_back('l'); s.push\_back('o'); EXPECT\_STREQ(s.c\_str(), "hello"); }

TEST(COWStringTest, ShrinkToFit) { string s("test"); s.reserve(100); EXPECT\_GE(s.capacity(), 100);

std::string old\_data = s.c\_str(); size\_t old\_size = s.size();

s.shrink\_to\_fit(); EXPECT\_GE(s.capacity(), old\_size); EXPECT\_STREQ(s.c\_str(), old\_data.c\_str()); }

TEST(COWStringTest, OperatorsPlus) { string s1("hello"); string s2(" world"); string s3 = s1 + s2; EXPECT\_STREQ(s3.c\_str(), "hello world");

string s4 = s1 + '!'; EXPECT\_STREQ(s4.c\_str(), "hello!");

string s5 = '!' + s1; EXPECT\_STREQ(s5.c\_str(), "!hello"); }

TEST(COWStringTest, Comparison) { string s1("hello"); string s2("hello"); string s3("world");

EXPECT\_TRUE(s1 == s2); EXPECT\_TRUE(s1 != s3); }

TEST(COWStringTest, Clear) { string s("hello"); s.clear(); EXPECT\_TRUE(s.empty()); EXPECT\_EQ(s.size(), 0); EXPECT\_STREQ(s.c\_str(), ""); }

TEST(COWStringTest, MoveSemantics) { string s1("hello"); string s2 = std::move(s1); EXPECT\_STREQ(s2.c\_str(), "hello"); EXPECT\_TRUE(s1.empty() || std::strcmp(s1.c\_str(), "") == 0); }

TEST(COWStringTest, COWSemantics) { string s1("shared"); string s2 = s1; string s3 = s1;

EXPECT\_EQ(s1.c\_str(), s2.c\_str()); EXPECT\_EQ(s1.c\_str(), s3.c\_str());

s2[0] = 'S'; EXPECT\_NE(s1.c\_str(), s2.c\_str()); EXPECT\_EQ(s1.c\_str(), s3.c\_str()); EXPECT\_STREQ(s1.c\_str(), "shared"); EXPECT\_STREQ(s2.c\_str(), "Shared"); EXPECT\_STREQ(s3.c\_str(), "shared"); }

TEST(COWStringTest, CapacityAndEmpty) { string s1; EXPECT\_TRUE(s1.empty()); EXPECT\_GE(s1.capacity(), 0);

string s2("non-empty"); EXPECT\_FALSE(s2.empty()); EXPECT\_GE(s2.capacity(), s2.size()); }

int main(int argc, char \*\*argv) { ::testing::InitGoogleTest(&argc, argv); return RUN\_ALL\_TESTS(); }