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.

Новосибирск 2025

Содержание

1. Постановка задачи

Спроектировать шаблонный тип данных String + COW. Использовать стандарт языка C++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.

2. 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).

3. Реализация

3.1 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.

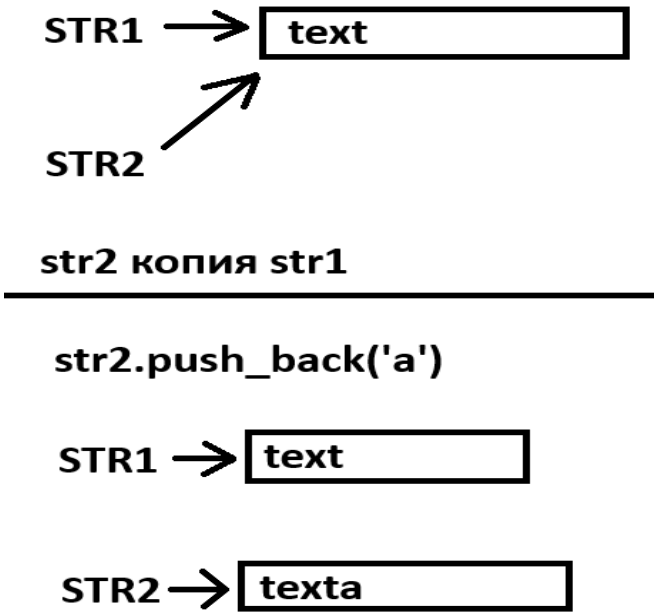

Рисунок 1: Принцип работы Copy-on-write

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

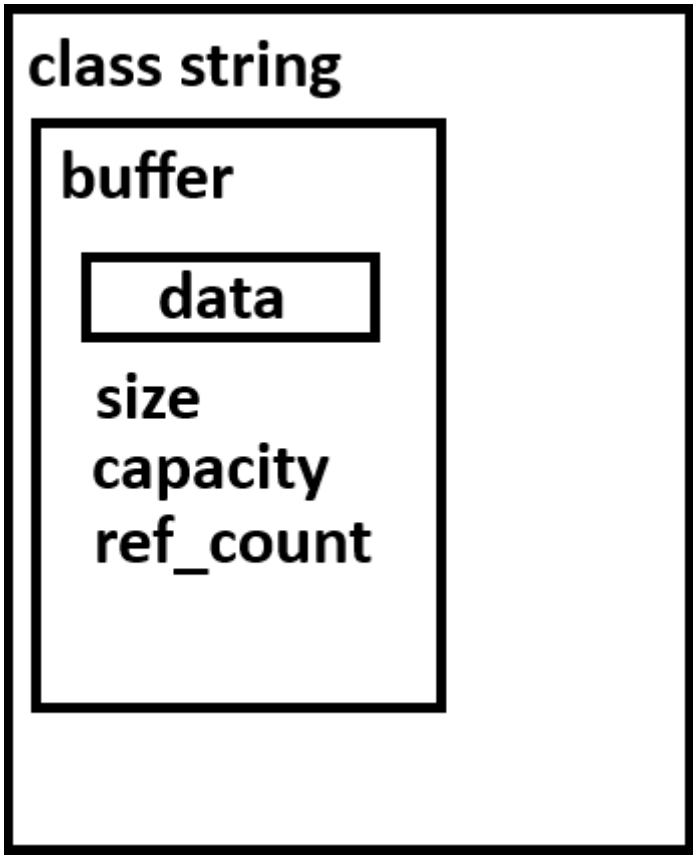

Рисунок 2: Устройство класса

3.2 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)

3.3 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

4. 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).

5. 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(); }
```