Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Курсова робота

освітній ступінь – бакалавр

на тему: «**ОПТИМІЗАЦІЯ ОБРОБКИ ТЕКСТОВОЇ ІНФОРМАЦІЇ**»

Виконав: студент 3-го року навчання,

Освітньої програми «Комп'ютерні науки», 122

Романюк Андрій Олександрович

Керівник Бублик В.В., _________

кандидат фіз.-мат. наук, доцент

Рецензент ________________________

(прізвище та ініціали)

Курсова робота захищена
з оцінкою ________________________

Київ – 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних

систем, доцент, канд. ф.-м. наук

____________ О. П. Жежерун

«____» _______________2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студента 3-го року навчання, факультету інформатики

Романюка Андрія Олександровича

Тема: Оптимізація обробки текстової інформації
Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання
Зміст
Список скорочень і термінів
Вступ
Історія імплементації рядків
Оптимізації простих рядків
Різниця імплементації SSO у STL та інших бібліотеках
Порівняння імплементацій SSO. Тести
Висновок
Список використаних джерел

Дата видачі «___» _________ 2025 р.

ЗМІСТ

ЗМІСТ 3
СПИСОК СКОРОЧЕНЬ І ТЕРМІНІВ 4
АНОТАЦІЯ 5
ВСТУП 6
1 ІСТОРІЯ ІМПЛЕМЕНТАЦІЇ РЯДКІВ 9
1.1 Зберігання довжини рядка у першому байті. Prefixed length strings 10
1.2 Використання нульового символу. Null terminated strings 11
1.3 Використання структур і класів для реалізації рядків 12
1.4 Імплементації рядків для редакторів тексту 13
1.4.1 Рядок з проміжком посередині. Gap Buffer 14
1.4.2 Представлення рядка бінарним деревом. Rope 15
1.4.3 Представлення рядка таблицею. Piece table 18
2 ОПТИМІЗАЦІЇ ПРОСТИХ РЯДКІВ 23
2.1 Оптимізація рядків способом лінивого обчислення. Copy-On-Write 23
2.2 Оптимізація коротких рядків за допомогою типу union. Short String Optimization 27
3 РІЗНИЦЯ ІМПЛЕМЕНТАЦІЙ SSO У STL ТА ІНШИХ БІБЛІОТЕКАХ 29
3.1 Оптимізація з використанням загального покажчика. GCC 30
3.2 Оптимізація виділенням окремих структур. Clang 32
3.3 Стандартна реалізація SSO. MSVC 34
3.4 Оптимізація перетворенням останнього байту. Folly 36
3.5 Поєднання імплементацій GCC та Folly. Власна 38
4 ПОРІВНЯННЯ ІМПЛЕМЕНТАЦІЙ SSO. ТЕСТИ 41
4.1 Тестування створення рядка різної довжини 42
4.2 Тестування копіювання рядка різної довжини 44
4.3 Тестування перетворення рядка з короткого у довгий 46
4.4 Тестування рядка алгоритмом Кнута-Морріса-Пратта 47
4.5 Тестування рядка парсингом CSV 49
4.6 Підсумки результатів 50
ВИСНОВОК 51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 52

СПИСОК СКОРОЧЕНЬ І ТЕРМІНІВ

1. Legacy - код або застосунок, що використовує старі технології. Зазвичай через проблеми з переходом на нові.
2. SSO (Short String Optimization) – вид оптимізації рядків, при якому використовується структура “union” для зберігання рядків малого розміру на стеці замість купи.
3. COW (Copy-On-Write) - вид оптимізації рядків, при якому використовується внутрішня підструктура для лінивого обчислення модифікацій рядка.
4. Lazy evaluation (ліниве обчислення) – стратегія оптимізації, при якій обчислення виразу відкладається до моменту, коли його результат буде безпосередньо потрібен.
5. ASCII (American Standard Code for Information Interchange) - стандарт кодування символів, що призначає числові значення від 0 до 127 для представлення англійських літер, цифр та спеціальних символів.
6. UTF (Unicode Transformation Format) - сімейство форматів кодування символів, що дозволяє зберігати та обробляти текст різними мовами світу.
7. Race condition (стан гонитви) - ситуація в багатопотоковому програмуванні, коли результат виконання коду залежить від послідовності або часу виконання різних потоків.
8. ISO/IEC (International Organization for Standardization/International Electrotechnical Commission) - міжнародні організації, що спільно розробляють та публікують стандарти в галузі інформаційних технологій та електроніки.
9. WSL2 (Windows Subsystem for Linux 2) - компонент операційної системи Windows, що дозволяє запускати Linux-програми та інструменти розробки безпосередньо в середовищі Windows.
10. Dead code elimination (видалення мертвого коду) - оптимізація компілятора, яка виявляє та усуває фрагменти програмного коду, результати виконання яких ніколи не використовуються.
11. CSV (Comma-Separated Values) - текстовий формат файлів для зберігання табличних даних, в якому значення в рядка розділяються комами.
12. MinGW (Minimalist GNU for Windows) - набір інструментів розробки, що включає компілятори GCC, утиліти та бібліотеки, адаптовані для створення нативних Windows-програм.
13. Віртуалізація - технологія, що дозволяє створювати віртуальні версії комп'ютерних ресурсів, таких як операційні системи.

АНОТАЦІЯ

У роботі розглянуто рядки та їхні імплементації впродовж історії. Оглянуто особливі реалізації, створені для оптимізації функцій редакторів тексту. Також оглянуто ідеї та загальні будови оптимізацій COW та SSO для стандартних рядків. Окрім загального огляду, детально проаналізовано найпоширеніші фактичні імплементації.

Також реалізована власна імплементація оптимізації коротких рядків, як більш ефективна у використанні пам'яті. Ця реалізація комбінує у собі SSO від GCC та Folly.

Вкінці проведено низку тестів для порівняння усіх імплементацій задля виявлення закономірностей впливу оптимізацій та їх конкретних реалізацій на швидкодію.

Ключові слова: SSO, COW, оптимізація коротких рядків, gap buffer, rope, piece table, оптимізація, lazy evaluation, union, GCC, Clang, MSVC, Folly.

ВСТУП

Рядок є одним із базових типів даних у програмуванні. З самого початку свого існування він видозмінювався, еволюціонував та розвивався. Велика кількість мов намагалася революціонізувати рядки та те, як програмісти їх використовують. Проте кожна нова ітерація на довго не затримувалась, а в рамках однієї існували різні групи розробників зі своїм баченням того, як повинні бути реалізовані рядки.

Основною проблемою цього типу даних є його замовчувана змінюваність. Інтуїтивно зрозуміло, що рядки, як репрезентація тексту, повинні постійно змінюватися: символи додаються, викреслюються і змінюються. Як тільки письмо почало існувати, людство сприймало його як щось непостійне, щось, що у будь-який момент можна змінити. Таке бачення перейшло і на рядки у програмуванні. Сама ідея комп’ютерів з’явилася із бажання людства пришвидшити та покращити процеси, які до цього виконувалися дуже неефективно. Тому у випадку, що щось настільки примітивне, як текст, не мало б кращої репрезентації, то сама ідея електронно-обчислювальних машин була б під питанням.

Саме через це виникали різні способи представлення рядків та їх оптимізації. Фундаментальною ідеєю, на якій базуються усі реалізації рядків, є те, що текст – це послідовність символів. І саме від цієї аксіоми і відштовхувалися інженери у своїй подальшій роботі. Хоча, з цим так само з’явилися і свої проблеми. А як визначити символ?

Більшість мов програмування, незалежно від парадигми, як-от C, C++, Java, Python, Haskell та багато інших сприймають символи як особливий випадок чисел. Тобто у відповідність певному символу ставиться певне число. Така система називається кодуванням. Наприклад, однією із перших є ASCII, що ставить у відповідність

- числам як рядкам
- англійським малим і великим літерам

- маленькому наборові популярних спеціальних символів

числа від 1 до 127. І це також в свою чергу принесло свої проблеми. У світі існує безліч мов та безліч різних символів. А часто, символи з різних мов зустрічаються у одному тексті, що робить таку ідею, як створення різних кодувань для різних мов доволі безперспективною.

Через таку відносну необмеженість символів і виникають усі ці проблеми. На відміну від рядків, інші типи даних, як-от цілі та дійсні числа, є краще окресленими та обмеженими типами, що дозволяє зручніше та ефективніше реалізувати їх. По-перше, в реальних умовах, є можливість наперед передбачити діапазон, в якому потрібно буде використовувати числа. В той же час, базовим методом використання рядків є його створення (написання тексту) та редагування. У більшості випадків важко визначити, як можна обмежити рядки, окрім моментів, де сама система має певні лімітації, як-от максимальна довжина імені користувача або кількість символів у коментарі. Проте навіть такі обмеження не завжди будуть ефективними, оскільки зазвичай ці діапазони є досить великими (часто, від одного до декількох сотень).

Звісно, більшість з них були вирішені. Проблема кодувань була вирішена остаточно. Була створена система UTF-8 (а також, -16, -32 та -64 для дуже великих наборів знаків), що за допомогою загального збільшення діапазону чисел (від 1 байта до 4 байтів), методів комбінації (символ ‘é’ вважається як символ ‘e’ разом з символом ‘◌́’) та постійних доповнень стандарту досягла майже повної відповідності усім можливим системам письма. Проблему розміру та змінюваності довелося заховати за структурами. Абсолютна більшість мов надають доступ до типу рядків через клас, який зазвичай називають string. Це дозволило приховати незручні моменти внутрішньої реалізації рядків за інтерфейсом, що дозволило розробникам відійти від проблем тексту і сфокусуватися на вирішенні своїх конкретних задач.

Проте саме останнє є менш стандартизованим. Світ програмування постійно покращується й адаптується до нових стандартів та ідей. Зазвичай це

обмежує інженерів у їх можливостях, оскільки, якщо стандарт з'явився, то це означає, що не усі їм підлягали. Через це існують різні реалізації одних і тих самих ідей.

В сучасному світі використовують оптимізацію коротких рядків, або ж short string optimization (SSO). Вона дозволяє використовувати пам'ять на стеці, а не у купі, для запису достатньо коротких (зазвичай до 16 символів) рядків. Як було зазначено, імплементація цієї оптимізації різниться між вендорами бібліотек (що зазвичай прив'язані до компіляторів). І хоча вони все ще мають одну і ту саму ідею, не можна повністю зрозуміти кожну окрему реалізацію, не розглянувши конкретно її.

Ця різниця також супроводжується питанням того, наскільки сильно вона впливає на швидкодію програм, що використовують рядки. Чи є між ними суттєва відмінність, чи сам факт наявності SSO вже достатній? До того ж, якщо була можливість створити таку різноманітність імплементацій, то, можливо, можна створити і власну, яка могла б увібрати в себе найкращі аспекти усіх і тим самим перевершити їх усіх.

Зважаючи на ці труднощі та неочевидність рішення, **за мету цієї роботи** були поставлені вивчення сучасних методів оптимізації рядків і визначення значущості оптимізацій та їх різниць.

Завданнями курсової роботи є –

1. Дослідити вже наявні методи оптимізації;
2. Створення власного застосунку для реалізації оптимальної обробки рядків символів;
3. Створення та виконання тестів продуктивності для усіх реалізацій за одних умов;
4. Виконання порівняльного аналізу результатів тестів.

Об'єктом дослідження є імплементація класу рядка та конкретні реалізації оптимізації SSO.

1 ІСТОРІЯ ІМПЛЕМЕНТАЦІЇ РЯДКІВ

Варто почати з історії імплементації рядків. Як вже було зазначено, текст сприймається як впорядкований набір символів. У програмуванні це означає, що потрібно обрати структуру, яка б відображала дане бачення. Очевидно, що вибір стояв між списками та масивами. І так само очевидно, що обраним був масив. По-перше, це економне використання пам'яті. Списки вимагають додаткових обсягів, оскільки кожен окремий елемент вимагає своєї підструктури, що окрім власне корисних даних буде також мати в собі мінімум додаткове посилання на наступний елемент. Масиви ж є впорядкованими комірками пам'яті, що не потребують додаткових витрат на розташування. Також це пришвидшувало доступ до елементів, що у списках має часову складність O(n), а у масивах вона була б сталою – O(1).

Хоча є і аргумент щодо списків. Дії із рядками як видалення, вставлення та редагування могли виконуватися швидше та зрозуміліше. В загальному випадку усі вони вимагали б максимум дві операції перепризначення покажчика та очистку видалених елементів. З іншого боку, масиви не мають такої гнучкості. При зміні рядків, елементи потрібно було б не тільки додавати і видаляти, але і рухати задля збереження структурної цілісності тексту. Також списки за замовчуванням динамічно змінюють розмір, в той час як для масивів необхідно створити новий і перенести туди дані, що є доволі ресурсовитратною задачею.

Оскільки з самого початку було визначено, що текст не повинен бути дуже ресурсомістким саме для обчислювань, було обрано масиви. Код пишеться лише раз і використовується величезну кількість разів. Спрощення роботи для програмістів у цьому випадку означало б погіршення у швидкодії для всіх, хто б використовував такі рядки.

Розібравшись із базовою ідеєю, розглянемо структури, що ближчі до практичної реалізації.

1.1 Зберігання довжини рядка у першому байті. Prefixed length strings

Одною із проблем для рядків у вигляді масивів є їх розмір і ємність. Необхідно, щоб в одній структурі ми могли тримати як дані, так і метадані. Тут варто згадати, що типи символу і цілого числа є взаємозамінними. Керуючись цією логікою було створено схему "префіксної довжини". Вона полягає в тому, що певна кількість перших елементів масиву буде використана не для символів, а для значення довжини самого рядка. Перевага такої імплементації в тому, що вона дуже проста і майже не потребує додаткових маніпуляцій. Якщо потрібно розмір рядка, то просто зчитуємо необхідну кількість перших байтів. Якщо потрібно отримати максимальний розмір, то він завжди сталий через те, що ми надаємо на розмір сталу і обмежену кількість байт, тобто ємність рядка буде дорівнювати максимальному значенню, якого може набувати розмір за даної кількості байт. Зчитування елементів має константну складність.

Такий спосіб відображення рядків популяризувався мовою програмування Pascal. А саме її імплементацією UCSD (University of California San Diego). У ній вказано, що рядок виділяє максимальну можливу пам'ять у розмірі 256 байтів [1]. Перший виділяється для довжини, а в наступних зберігаються символи.

Загалом така структура є доволі компактною та інтуїтивно зрозумілою. Проте вона не дуже гнучка. Очевидно, рядки обмежені по максимальному розміру. Також попри фактичну довжину тексту, ми завжди виділяємо максимальну кількість пам'яті для рядка, що не є оптимальним способом використання пам'яті.

Деякі імплементації намагалися виправити цю імплементацію. Наприклад, у стандарті Extended Pascal ISO 10206 [2] вказано, що рядки можуть бути будь-якої довжини. Зазвичай це досягалося простим збільшенням префіксу до 2 байтів, що розширювало максимальну кількість символів до 65,536.

Однозначно, така імплементація не була найкращою. Вона погано масштабувалась, потребувала більше уваги від розробників та обмежувала їх.

1.2 Використання нульового символу. Null terminated strings

Іншою схемою, яку створили приблизно у той самий час, є “рядки з нульовим закінченням”, або ж “null-terminated strings”. Замість того щоб зберігати метадані у масиві, цей метод обрав більш імплікативний спосіб. Після останнього символу рядка у масив поміщається null-символ. Він має числове значення нуля і позначає кінець цього рядка. Цей метод розв’язує проблему обмеженості рядків, оскільки тепер можна виділити довільну кількість пам’яті і вкінці додати один нульовий символ.

Ця імплементація набула поширеності завдяки своєму використанню у C++ та в операційній системі Unix [3]. Свого часу Unix був дуже популярним, а поєднуючи це з тим, що його часто розповсюджували безкоштовно для студентів [4], то не дивно, що саме ця реалізація рядка набула такої популярності.

Тепер щоб визначити поточний розмір рядка варто пройтися по усіх символах від початку і рахувати їх кількість, доки не дійдемо до null’я.

На жаль, це привело до нової проблеми – тепер неможливо без додаткових даних однозначно описати розмір і місткість рядка одночасно. Якщо ми поставимо null-символ безпосередньо після фактичного тексту, то втратимо інформацію про усю виділену пам’ять. Якщо ж він буде в кінці масиву, то ми не зможемо точно сказати, де останній символ. Цю проблему можна вирішити додатковим зберіганням одного зі значень у додаткові змінній.

Таким чином, рядок перекладає частину відповідальності за збереження структури на програміста, проте прибирає обмеження по розміру і надає більшої гнучкості.

Одним із цікавих моментів такої імплементації є неочевидна, але логічна поведінка вищезгаданого способу підрахунку розміру рядка. Поглянемо на таку реалізацію методу мовою C++:

```
int length(const char* str) {
    int i = -1;
    while (str[++i] != '\0');
    return i;
}
```

У ньому ми проходимось по усіх елементах від початку до кінця, доки не зустрінемо null, після чого повертаємо індекс. Початкове значення індексу дорівнює -1, оскільки ми використовуємо преінкремент.
Розглянемо такий рядок:

```
char str[] = "hello\0world";
```

Інтуїтивно здається, що розмір рядка повинен бути 11, проте після виконання методу отримуємо значення 5. Стає зрозуміло, що в процесі роботи алгоритму, коли індекс доходить до 5, то зустрічається null-символ, умова перестає виконуватися і значення повертається.

У загальному випадку це можна знехтувати. Null-символ майже не зустрічається у тексті поза контексту кінця рядка. Проте це може мати наслідки у вигляді пошкоджених або втрачених даних. Тому програмістам варто уважно розробляти застосунки, у яких є можливість отримання тексту з null-символом усередині.

1.3 Використання структур і класів для реалізації рядків

Як бачимо, при роботі із такими “голими” рядками, які повністю контролюються розробником, виникає низка однотипних проблем. Це збереження додаткових даних про розмір та ємність рядка. Також абсолютна

більшість маніпуляцій з рядками, яких потребують програмісти, є однотипними та мають очевидну реалізацію для усіх типів структури рядків.

З огляду на це та на популяризацію об’єктно-орієнтованого програмування, багатьма розробниками було вирішено створити клас рядків. Таким чином, можна буде помістити в одну структуру корисний текст та метадані до нього. Стандартні методи для їх обробки існують як інтерфейс класу, що робить код зрозумілішим і більш модульним. Також це дозволило б використовувати складніші реалізації без негативного впливу на читабельність коду. Паралельно з цим, планування, створення та підтримка будь-яких оптимізацій стає легшою.

Перша і найочевидніша реалізація класу має такий вигляд:

```
class string {
    char* data;
    int size;
    int capacity;
};
```

Це впровадження є легким для сприйняття. Попри це, для малих розмірів рядків дана структура не буде дуже ефективною, оскільки вона завжди буде виділяти мінімум 16 байтів на стеці та додаткову пам’ять у купі, при тому що сам текст займав би менше місця. У реальному середовищі короткі рядки з’являються набагато частіше. Саме про цю проблему і її рішення буде іти мова у даній роботі.

1.4 Імплементації рядків для редакторів тексту

Було показано, як класи чудово вирішують проблему фактичного впровадження структури даних для тексту. Розглянутий клас є загальним та існує для задоволення потреб більшості програмістів. Він не використовує припущень щодо можливих методів його застосування.

Згадаємо одну із головних характеристик тексту – змінність. У певних задачах ця властивість може бути дуже критичною. Наприклад, у програмах редагування тексту. Чи то стандартний вбудований додаток “Нотатки”, чи інтегроване середовище розробки, усі потребують функції миттєвої зміни тексту у будь-якому місці. Можна було б спробувати використати стандартну реалізацію, проте постійні довільні перетворення вимагали б набагато більшої кількості обчислень.

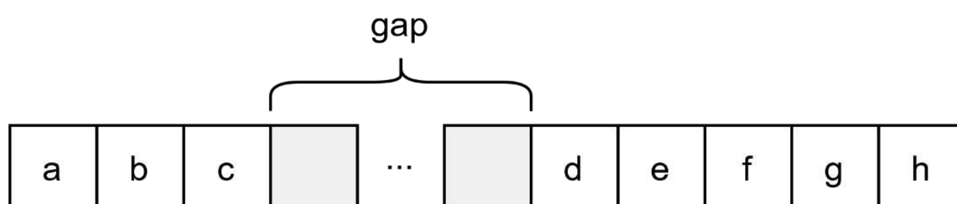
З цієї причини на світ з’явилися відповідні реалізації рядків.

1.4.1 Рядок з проміжком посередині. Gap Buffer

Схема “Gap buffer” є найбільш схожою на розглянуті. Буфер з проміжком так само використовує масив елементів як головний спосіб зберігання даних. Доповненням до структури є поняття “курсора”, яке вказує на місце, де користувач хоче додати або видалити текст.

Під час створення масиву окрім місця для зберігання наявного тексту, виділяється певна кількість додаткової пам’яті, що буде використана для додавання нових символів. На відміну від стандартного рядка, де вільний простір лежить після даних, вкінці масиву, пробіл буде розміщений між частинами тексту, де займає місце умовний курсор. Нехай потрібно створити об’єкт такого класу. Для тексту розміру n, буде виділено масив розміру n+m. Курсор розташований після t-ого символу. Тоді у масиві з першої по t-ту комірку будуть розташовані перші t символів тексту, наступні m будуть порожніми, а потім лежатиме залишок з t +1-ого по n-ий символи.

Для прикладу використаємо текст “abcdefgh”. Нехай курсор розташовується між ‘c’ та ‘d’. Тоді буфер буде мати такий вигляд:



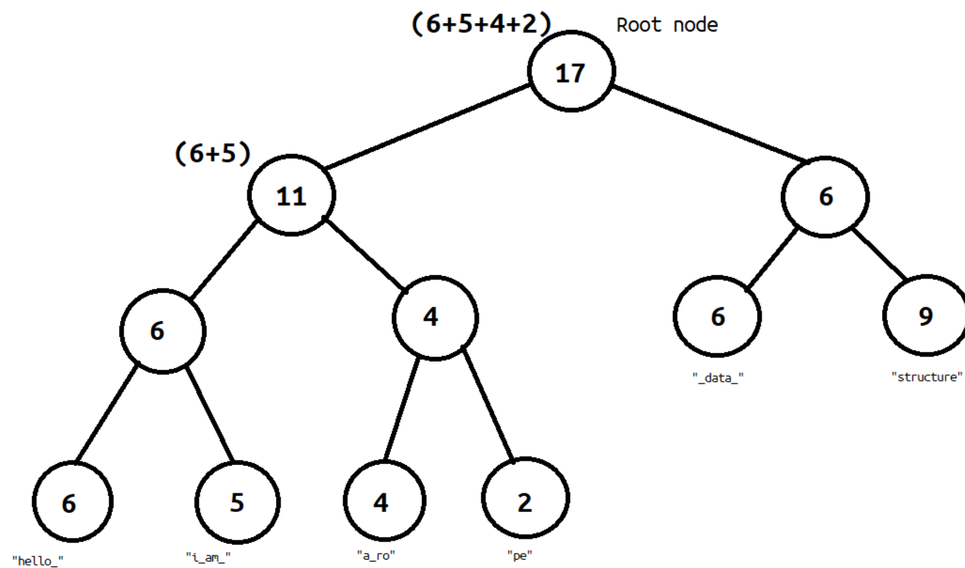
Цією структурою легко маніпулювати. При додаванні тексту символи записуються на початок проміжку. Видалення насправді не потребує маніпуляцій у буфері, воно оброблюється шляхом зміни індексів самого проміжку. Доступ до елементів має все ще константу складність, проте через наявність проміжку, потрібно перевіряти, в якій половині тексту лежить символ.

У момент, коли потрібно змінити положення курсора, дані переміщаються в межах самого буферу, після чого змінюються координати проміжку. Насправді немає потреби робити це при кожній перестановці курсора. Використовуючи принцип лінивого обчислення, положення курсора буде зберігатися окремо і лише при редагуванні тексту застосуються зміни.

При заповненні буфера, доведеться виділяти новий масив та копіювати дані. Під питанням залишається лише розмір нового проміжку. Його можна визначити константою, проте більш елегантним рішенням буде виділення, що залежить від розміру тексту або що спирається на оптимізації використаного алокатора.

1.4.2 Представлення рядка бінарним деревом. Rope

Іншою структурою даних, що так само існує задля швидких змін у тексті, є “мотузка”, або ж “rope”. Проте вона відходить від стандартного методу представлення рядка у вигляді масиву. Замість цього, тут використовується структура збалансованого бінарного дерева [5]. Кожна нода є або розгалуженням, або листком. Тільки у листках зберігається текст. У проміжних нодах зберігається розмір рядка в цій ноді.



У кожному листку зберігається об’єкт спеціального рядка. Для них визначений спеціальний сталий максимальний розмір. Також ці рядки є незмінними, тобто якщо надійде команда про зміну тексту всередині листка, то його доведеться знищувати та створювати новий листок або піддерево. Проте таке обмеження дозволяє застосувати оптимізацію “копіювань при записі”, або ж “Copy-On-Write”. Його ідею буде пояснено пізніше, зараз важливий лише факт його існування.

При створенні рядка такого виду, текст буде розбиватися на окремі малі підрядки з вже зазначеним максимальним розміром. Потім їх помістять у окремі листки. Наприкінці усі листки поєднають в одне велике дерево, що і буде представленням даного рядка.

Така структура дуже сильно виграє у швидкості додавання і видалення тексту. Тепер замість того, щоб модифікувати один великий шматок тексту, ми працюємо з окремими малими відрізками, які можна легко відв’язувати від основної структури.

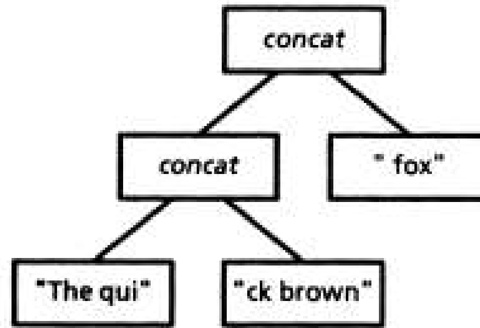
Доступ до символу рядка відбувається за допомогою бінарного пошуку. Коли надходить запит із індексом, перевіряється, чи він належить лівому чи правому піддереву і шукає символ у відповідному. Таким чином, складність буде O(log(n)), що набагато швидше за лінійну складність стандартного рядка.

Конкатенація двох рядків відбувається шляхом створенням нової головної ноди, де її лівим і правим піддеревами стають відповідні початкові рядки. Їхня довжина додається і поміщається у головну ноду. Таким чином, додавання рядків завжди має часову складність O(1).

Видалення тексту відбувається за двома сценаріями. Якщо є потреба видалити текст усередині листка, то через немутабельну природу внутрішніх рядків, доведеться видаляти та вставляти новий листок. У випадку з видаленням тексту, що представлений цілим листком, то, очевидно, просто видаляється увесь листок. Суміщений випадок (коли потрібно видалити текст, що є як частиною листка, так і іншим цілим листком) виконує одразу ці дві дії.

Однією із проблем такої реалізації є те, що після серії змін дерево усередині рядка може перестати бути збалансованим, тобто найбільша різниця між глибинами його піддерев більша одиниці. Тут необхідна можливість перебалансувати дерево.

Загальний алгоритм дуже схожий на сортування злиттям, де масив розділяється надвоє і рекурсивно поєднуються елементи. Спочатку з дерева дістаються усі листки та поміщаються у окремий контейнер (зазвичай, масив із покажчиків на листки, аби уникнути копіювання). Потім контейнер розділяється навпіл і викликається цей самий метод на обидві половини. Коли виклик доходить до двох листків, то з них створюється дерево. Після цього виклики “розгортаються” і дерева поєднуються вже визначеним способом. Таким чином, максимальна різниця глибин точно буде одиницею, оскільки самі піддерева гарантовано матимуть різницю не більше одиниці. Максимальна глибина буде відрізнятися від нуля, якщо загальна кількість листків є непарною і у одному з кінців контейнеру доведеться поєднувати 3 листки, що призведе до дерева такого вигляду:



Із цікавого, якщо перекласти слово “string” напряму, то отримаємо слово “нитка”, а “rope”, як вже було згадано, означає “мотузка”. Це неначе вказує на більшу потужність останнього. Також це перегукується із ідеєю нод, які використовуються в rope, оскільки вони ще називаються “knot”, або ж “вузол”.

1.4.3 Представлення рядка таблицею. Piece table

Третій і останній метод представлення рядків, який буде розглянутий, є “таблиця шматків”. У ній замість того, щоб зберігати безпосереднє представлення рядка, зберігається початковий текст, масив усіх доданих даних та масив посилань на, власне, шматки.

Загалом таку структуру можна представити так:

```
struct chunk {
    std::string* text;
    int offset;
    int length;
};
class piece_table {
    const std::string initial_text;
    std::vector<std::string> added;
    std::vector<chunk> chunks;
};
```

В основі цієї структури лежить доволі цікава ідея. При її створенні, початковий текст (дані з файлу) зчитується та поміщається у initial_text. Цю змінну не можна змінювати, вона доступна тільки для читання. Коли

користувач додає текст (неперервну послідовність символів), вона додається як ще один елемент до масиву added. Головним представленням тексту є масив chunks. В ньому кожен елемент посилається на початковий текст, або на один із доданих із значенням першого символу та їх кількості опісля. Дізнатися поточне представлення тексту можна, послідовно зчитуючи шматки.

При початковому створенні об’єкта, у масиві буде лише один шматок, що посилатиметься на початковий текст. Offset буде рівним нулю, а length буде дорівнювати довжині тексту. Якщо потрібно додати текст посередині, то єдиний шматок видаляється і створюється два нових, де перший і другий посилатимуться на відповідні частини тексту до і після доданого.

Наприклад, для тексту “abcd” початковий стан виглядає так:

Piece Table	
initial	"abcd"
added	[]
chunks	[{initial, 0, 4}]

Після додавання “mid” усередину (“abmidcd”), структура зміниться на:

Piece Table	
initial	"abcd"
added	["mid"]
chunks	[{initial, 0, 2}, {added[0], 0, 3}, {initial, 2, 2}]

Стирання символів може відбутися за 3-ма сценаріями. Видалення тексту, що визначений цілим шматком, означає видалення усього шматка. Якщо

потрібно видалити текст на краях шматка, то змінюються тільки значення зміщення (offset) і довжини (length). У випадку видалення тексту усередині шматка, шматок видаляється і створюється два нових із урахуванням операції (аналогічно до додавання нового шматка).

Доступ до елементів має складність O(n), оскільки доведеться проходитись по усіх шматках, щоб дізнатися, де саме можна знайти символ. Отримати підрядок можна шляхом копіювання відповідних шматків і, якщо необхідний текст починається або закінчується посередині, зміною зміщення і розмірів.

Перевагою такого представлення є те, що, насправді, не обов'язково видаляти текст, який користувач стер. Замість цього у структуру шматка можна додати прапор "існування".

```
struct chunk {
    //rest of code
    boolean exists;
};
```

Якщо цей рядок репрезентує частину загального тексту, то значення буде true, якщо ж він видалений, то false. Таким чином, ми можемо зберігати "історію" змін.

І це дуже вдале доповнення до структури. Досі структури розглядалися лише з точки зору максимальної швидкості змін. Проте у справжніх застосунках це не є єдиною метою. Найпростіша функція редакторів тексту – скасування зміни – потребуватиме додаткового планування. Необхідно створити додаткову структуру, яка б визначала історію змін. Використовуючи попередні імплементації, це доповнення погано інтегрувалося б з уже наявним класом рядків. Воно б створило накладні витрати, з якими важко працювати.

У piece table ця проблема за замовчуванням частково вирішується. З мінімальними змінами, такими як додавання лічильника змін у загальну

структуру та номер зміни у окремий шматок, можна ефективно реалізувати функцію скасування (Ctrl+Z).

Розглянуті класи є гарними прикладами спеціалізації. Загальна імплементація існує для того, щоб точно виконувати усі можливі функції, яких би хотіли розробники. Спеціалізація ж дозволяє оптимізувати роботу структури для певного напрямку. Зазвичай це супроводжується втратою швидкодії в інших моментах. Наприклад, як можна було побачити, спеціалізовані класи часто потребують більше роботи для отримання окремого символу із тексту. Проте у редакторах тексту немає потреби часто отримувати окремі символи, а переваги у роботі із модифікаціями геть переважають.

Водночас спеціалізацію можна "продовжити" і отримати ще цікавіші структури. Наприклад, VS Code – одне із найпопулярніших середовищ розробки у світі. У ньому використовується суміш piece table та rope [6]. Загальне впровадження схоже на таблицю, з початковим текстом і набором додатків. Проте послідовність шматків представлена не масивом, а бінарним деревом, як rope.

```
class Node {
    std::string* text;
    int start;
    int length;
    int left_subtree_length;
    int left_subtree_lfcnt;
    Node* left;
    Node* right;
    Node* parent;
};
class PieceTable {
    std::string initial;
    std::vector<std::string> buffers;
    Node* rootNode;
};
```

Різні тести [5] [7] [8] показують доволі різні результати. Загалом, базові дії (створення файлу, читання, одноразове додавання тексту) відбуваються набагато швидше на gap buffer. З іншого боку, інтенсивне редагування набагато

краще працює з rope та piece table (очевидно, оскільки саме для цих дій вони були оптимізовані). Важливим моментом є накладні витрати по пам’яті. Rope та piece table витрачають її в сотні разів більше, що може бути вирішальним фактором, якщо ціллю є створення мінімально вибагливого редактору тексту. Але, такі, на перший погляд, великі витрати, часто мало впливають на швидкодію завдяки потужностям сучасних процесорів та об’ємам пам’яті.

2 ОПТИМІЗАЦІЇ ПРОСТИХ РЯДКІВ

Очевидно, зміни у загальному підході до репрезентації рядків можуть значно вплинути на ефективність структури. Такі спеціалізації можна використовувати навіть поза сферами, для яких їх було створено. Проте більшість програмістів не бажають змінювати свої інструменти розробки, оскільки велика кількість роботи так само буде відрізнятися.

Через це, стандартом є саме усереднений варіант з використанням стандартного буфера у купі. Така структура буде передбачувано працювати і видавати прийнятні результати для будь-якого сценарію.

До того ж, в рамках цих структур усе одно можна додати певні покращення та оптимізації. Вони не будуть настільки ефективними, як повна зміна репрезентації, проте вони стовідсотково зроблять рядки швидшими у певних випадках, при цьому не сповільнюючи інші аспекти.

2.1 Оптимізація рядків способом лінивого обчислення. Copy-On-Write

Впродовж роботи із рядками програмісти помітили одну проблему, що сильно впливала на швидкодію програм – рядки часто копіювалися, при цьому ніяк не модифікувалися. Це виглядає комічно, з огляду на те, скільки сил та досліджень було вкладено саме в оптимізацію процесів модифікації. Загалом, дуже часто рядки використовувалися майже як примітивні дані – створювалися, багато копіювалися, зберігалися, але не змінювалися. Водночас час на копіювання рядка у купі нікуди не дівався.

Цю проблему було вирішено, представивши нову підструктуру до рядка. Тепер основний інтерфейс був лише обгорткою для вкладеного рядка. Нова структура мала такий вигляд:

```
class inner_str {
    char* data;
    size_t size;
    size_t capacity;
    size_t use_count;
};
class string {
    string_impl*impl;
};
```

Така імплементація отримала назву “Copy-On-Write” (COW), або ж “копіювання при записі”. Така структура перегукується із концепцією лінивого обчислення, оскільки дія виконується лише при її безпосередній потребі, а не під час виклику.

Основною модифікацією є додаткове поле use_count, що відслідковувало, скільки рядків посилаються на ті самі дані у купі. При створенні рядка створювався нова підструктура зі значенням use_count одиниця. При копіюванні, створювалося лише нове посилання на inner_str і інкрементувався лічильник.

Якщо потрібно змінити рядок, то спочатку перевіряється, скільки інших рядків володіє цими ж даними. Якщо той, хто викликав – єдиний власник, то рядок одразу модифікувався. У випадку, що власників є декілька, підструктура копіювалася, а її лічильник набував значення 1 (оскільки на нього посилається тільки рядок, який, власне, і ініціював копіювання). Після цього змінювалася копія. Видалення так само перевіряло, скільки є власників, і якщо їх декілька, то просто знищувалася обгортка, не зачіпаючи даних, а якщо один, то стирався геть увесь рядок.

Така імплементація була дуже корисною і справді значно пришвидшувала програми. Оскільки на момент такої імплементації ще не існувало поняття, як “r-value reference”, то тимчасові рядки завжди створювалися, копіювалися і потім знищувалися.

Наприклад, у цьому коді:

```
std::map<string, int> m;
m.insert(
    std::pair<string, int>("lorem ", 1)
);
```

Спочатку створюється об'єкт рядка із символів "lorem". Потім його значення копіюється у пару (std::pair). Далі сама пара копіюється у мапу. Загалом, ми маємо цілих три копіювання, хоча визначали лише один рядок.

З використанням COW, цей процес значно пришвидшувався, оскільки усі три копіювання сходилися до копіювання покажчиків.

На жаль, така імплементація, незважаючи на свою відносну простоту, зіштовхнулася із великою проблемою. Окрім програмного забезпечення, паралельно розвивалася і апаратна частина. Комп'ютери перейшли від одноядерної архітектури до багатоядерної. Багатопоточність програм стала новою метою для розробників, оскільки тепер можна було розділити завдання на певну кількість підзадач, які можна виконувати паралельно.

Для рядків з COW оптимізацією це стало поганою новиною. Оскільки різні, непов'язані об'єкти володіли одними і тими самими даними, це призводило до стану гонитви, коли паралельний доступ одного ресурсу мав непередбачуваний результат.

Наприклад, якщо один рядок викликав деструктор, а інший конструктор на тих самих даних, то може відбутися таке, що спочатку обидва зроблять перевірку і дійдуть до висновку, що проблем немає і можна виконати свої дії. Другий починає копіювати дані, а перший в цей момент їх видаляє. Таким чином, копіювання буде відбуватися на сміттєвих даних, тобто дані були втрачені.

Цю проблему можна вирішити, змінивши тип поля лічильника:

```
class inner_str {
    //rest of code
    std::atomic<size_t> use_count;
};
```

Тепер він буде атомарним, тобто, якщо два рядки спробують одночасно зробити якусь дію над рядком, що передбачає модифікацію, перевірки відбудуться поступово. Це знизить шанс неправильної роботи з даними. Проте все ще є проблеми. Якраз у конструкторах і деструкторах може бути декілька нестандартних сценаріїв, які, незважаючи на їхню рідкість, потрібно вирішувати. Через це код стане більш заплутаним, але більшість переваг від COW все ще залишиться.

Зрештою, проблеми усе ще були присутні. Проблема швидкості копіювань, яку COW намагалася виправити, з'явилася знову, але тепер з іншого боку. Тепер проблемою є операції з атомарним полем use_count. Більшість функцій (конструктори, деструктори, неконстантні методи) потребували роботи з цим полем. Якщо декілька окремих потоків хотіли якось модифікувати рядок, то спочатку усі вони мали пройти через перевірку лічильника. Це призводило до створення великої черги. Чим більше потоків хотіли працювати з рядком, тим більше ставала черга і тим менш ефективною була робота програми.

Цю проблему вже не можна було вирішити. Або потрібно було залишити усе як є, або повернутися до неоптимізованої, але надійної імплементації.

У стандарті C++11 [9] Copy-On-Write став застарілим та забороненим. Через проблеми з її ефективністю і підтримкою, було вирішено повністю відмовитися від неї. Також роль зіграли правила інвалідації похідних структур. Ітератори з рядків, покажчики на внутрішні масиви могли різко перестати бути валідними, оскільки інший потік змінив дані, що відбувалося зазвичай через дуже тривіальні методи (operator[], c_str).

Сьогодні, COW мало коли використовується у проектах. Зазвичай це legacy проекти. Їх використання виправдане лише у низці завдань, де розробники чітко розуміють, навіщо вона їм.

2.2 Оптимізація коротких рядків за допомогою типу union. Short String Optimization

На сьогоднішній день, усі популярні компілятори (GCC, Clang, MSVC) використовують short string optimization (SSO), або ж “оптимізацію коротких рядків” у своїх імплементаціях стандартної бібліотеки C++.

Ідея цієї оптимізації полягає в тому, що часто рядки займають багато пам’яті, для збереження доволі малих текстів. Стандартна імплементація рядка завжди займатиме мінімум 24 байти пам’яті: 8 байтів для покажчика на дані в купі, 8 байтів на розмір та 8 байтів на місткість. Для малих рядків така схема буде дуже незбалансованою. Якщо, для прикладу, взяти рядок “ipsum”, то його фактичний розмір – 5 байтів. Також розмір і місткість рядка нечасто будуть досягати тих значень, яких їм дозволяє пам’ять (18 квінтильйонів).

Спираючись на таке бачення, було створено метод оптимізації коротких рядків. В його основі лежить використання структури “union”. Вона дозволяє декільком змінним ділити між собою ті самі комірки пам’яті. Ця структура є дуже специфічною, оскільки в основному вона використовується при одному сценарії. Створюється структура, де є поле, що показує, якого типу саме повинно бути інше поле, і власне union, в якому є самі дані.

```
struct values {
    int type;
    union {
        char text;
        short num;
        float fl;
    }
}
```

Тут, при доступі до структури, спочатку перевіряється значення у type, після чого, в залежності від значення, повертається конкретне значення конкретного типу з union.

Саме з такою ціллю і використано union в SSO. Тільки у випадку рядків розрізняють лише два типи: короткий і довгий (або стандартний). Різні вендори

по-різному вирішують, в якому саме вигляді використовувати union, оскільки навіть тут є свої розгалуження та міні-оптимізації. Часто, різниця в імплементації пояснюється вже наявними стандартами або бажанням полегшити конкретні частини коду.

Зазвичай рядки з SSO мають такий вигляд:

```
class string {
    size_t size;
    union {
        struct {
            size_t capacity;
            char* heap;
        } _long // long string
        char[] buff; // short string
    }
}
```

У цій структурі пам’яттю діляться структура, що складається з посилання на текст у купі та ємкості, та масив (буфер) на стеку. Таким чином, буферу надається 16 байтів, які він може використати для зберігання тексту без алокації.

Така імплементація все ще займатиме 24 байти. Проте тепер рядки, що менші за 16 символів (рівно 16 не підходить, оскільки також потрібен один байт для null-символу) можна зберігати на стеці. Це пришвидшує створення, копіювання, доступ до символів і більшість інших методів рядка, на відміну від постійного використання даних у купі.

SSO не ділить дані на шматки, не розподіляє їх між різними об’єктами, не працює з кількостями, а лише трохи пришвидшує алокацію для дуже коротких рядків. Це не призведе до колосальних виграшів у швидкодії та навіть потребує більших зусиль від програмістів для імплементації. Однак воно є повністю модульним, і усі дії з рядками зачіпають тільки той рядок, з якого вони були безпосередньо викликані.

3 РІЗНИЦЯ ІМПЛЕМЕНТАЦІЙ SSO У STL ТА ІНШИХ БІБЛІОТЕКАХ

Як вже було підмічено, імплементація оптимізації коротких рядків дуже залежить від самого вендора компілятора, а точніше бібліотеки.

На сьогодні абсолютна більшість проектів на C++ використовують C++ Standard Library (стандартна бібліотека C++). Це набір шаблонів та функцій до них. Насправді вона базується на старій бібліотеці STL (Standard Template Library), а саме є її стандартизацією [10]. Створена у 1993 році, вона швидко набула популярності завдяки тому, що визначала усі найпотрібніші у розробці класи (рядок, масив, список, черга і стек, словник\мапа і т.п.), при цьому не перенасичувала себе зайвими класами. Через це вона стала стандартом у комерції.

Цікаво те, що, будучи набором шаблонів, стандартна бібліотека за своєю суттю сама є шаблоном. Існують загальні вимоги до того, як повинна організовуватися структура всередині бібліотеки. Сам стандарт має назву ISO/IEC 14882 [11]. Нас цікавитимуть конкретні імплементації в рамках цього стандарту.

Далі будуть розглянуті імплементації від: GCC (GNU Compiler Collection), Clang (набір компіляторів на базі LLVM[1]), MSVC (Microsoft Visual C++), folly (бібліотека від Facebook). Також буде присутня окрема власна імплементація, що поєднує GCC та folly. Незважаючи на те, що folly є власною бібліотекою, її ми все ще розглянемо, оскільки вона є відносно популярною та має багато спільного із стандартною бібліотекою, оскільки її основною ціллю було оптимізувати та організувати код саме для потреб Facebook’у.

Технічно розглядатимуться саме бібліотеки, що йдуть разом з цими компіляторами. Але назви цих бібліотек (libstdc++, libc++) є зазвичай схожими

[1] https://llvm.org/

або мають ту ж назву, що і компілятор. Тому набагато легше буде користуватися саме назвами їх відповідних компіляторів.

Наступні приклади коду будуть спрощеними. Опущені будуть тільки дуже тривіальні додаткові структури, та шаблонізація для різних типів символів, бо це зазвичай існує через можливість використання різних алокаторів та стадії компіляції, який ми в цій роботі не розглядаємо. Частини коду, що напряму впливають на імплементацію, матимуть спрощене представлення типів та назви. Наприклад, замість назви типу size_type буде використано size_t, тому що воно ним і є.

3.1 Оптимізація з використанням загального покажчика. GCC

GNU Compiler Collection є найстаршою колекцією компіляторів, що використовується і дотепер. І хоча GCC є саме набором для різних мов, оскільки більшість з них вже застаріла (Fortran, Objective-C, COBOL та ін.), він розглядається саме з боку компіляції C++. Бібліотека GCC має назву libstdc++[2].

Перейдемо одразу до самого коду:

```
class string {
    char* pointer;
    enum {
        local_capacity = 15 / sizeof(char) // 15 / 1 = 15
    };
    size_t length;
    union {
        char local_buf[local_capacity + 1];
        size_t capacity;
    }
}
```

Видно, як union використовується для поєднання місткості рядка та буфера на стеці. Ідея є логічною, оскільки, якщо ми знаємо, що рядок короткий, то його

[2] https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/basic_string.h

місткість завжди стала, а якщо ж він довгий, то його можна дістати з capacity, бо ми будемо впевненими в тому, що local_buff не має корисної інформації. Окреме поле length працює для обох, оскільки ми не можемо зробити припущень щодо розміру у жодному випадку.

Проте така імплементація має ще одну цікаву рису. Покажчик pointer завжди буде валідним, тобто завжди вказувати на комірку, де зберігається текст. Це досягається дуже простим, але неочевидним прийомом. Якщо рядок ініціалізується коротким (або перетворюється в такий), то після усіх маніпуляцій покажчик набуде значення буфера:

```
pointer = local_buf;
```

Таким чином, одразу вирішується декілька проблем. По-перше, перевірка типу рядка буде виконуватися через рівність указника буферу:

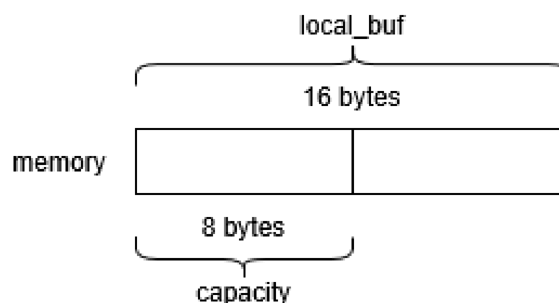
```
bool is_local() const {
    return pointer == local_buff;
}
```

Якщо покажчик і буфер вказують на одну й ту саму пам’ять – це короткий рядок, у іншому випадку – довгий.

Також такі операції, як доступ до окремих елементів рядка, не потребуватимуть перевірки типу; завжди можна одразу звернутися до покажчика:

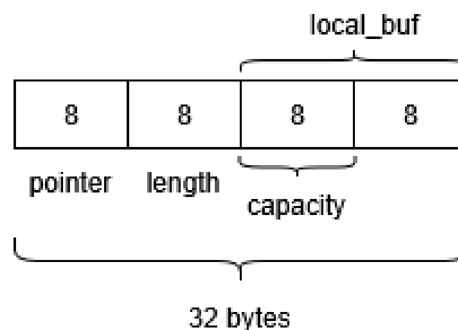
```
char& operator[](size_t __pos) {
    // error handling
    return pointer[__pos];
}
```

Повернімося до union. Ми об’єднуємо місткість і статичний масив. Оскільки місткість має тип size_t – 8 байтів, а масив займає 16 байтів, в результаті отримуємо таку схему:



Таким чином, capacity і local_buf ділять між собою 8 спільних байтів, а інші 8 байтів використовуються тільки останнім. В цьому випадку використання union не є дуже ефективним, оскільки вісім байтів просто "висять". Також з цього виводимо, що максимальний розмір короткого рядка – 15 байтів. Один байт завжди потрібно буде використати для null-символу.

Якщо поглянути на загальну структуру, можна побачити, що вона займає 32 байти:



Це на 8 байтів (33%) більше, ніж стандартна реалізація. Така додаткова витрата може здатися непропорційною. Але, оскільки це пам'ять на стеці, робота з нею буде набагато швидше, а додаткові витрати вийдуть не настільки критичними через великі об'єми оперативної пам'яті на сучасних комп'ютерах.

3.2 Оптимізація виділенням окремих структур. Clang

Clang є одним із найновіших компіляторів на ринку. Базується він на технології LLVM (оригінально розшифровувалася як "Low Level Virtual Machine", проте з часом від цього відмовилися і LLVM тепер є просто назвою). Ця технологія дозволяє розробникам компіляторів легше та швидше розробляти фронт- та бек- частини, застосовуючи ті самі оптимізації для різних мов.

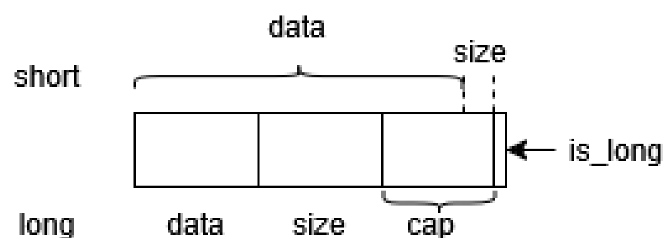
Clang вважається потужнішим за GCC, оскільки він враховував велику кількість проблем попередника. Незважаючи на це, GCC все ще використовується більше, оскільки він вже вкоренився в індустрію.

Ця імплементація рядка у бібліотеці libc++[3] для обох типів рядка використовує дві різні структури:

```
struct __long {
    char* __data_;
    size_t __size_;
    size_t __cap_ : 63;
    size_t __is_long_ : 1;
}; // size = 24

enum {
    __min_cap = (sizeof(__long) - 1) / sizeof(value_type) > 2 ?
(sizeof(__long) - 1) / sizeof(value_type) : 2 // 23
};

struct __short {
    char __data_[__min_cap];
    unsigned char __size_    : 7;
    unsigned char __is_long_ : 1;
};
```

Ця структура значно відрізняється від усіх інших. Замість того щоб поєднувати окремі поля, цей рядок повністю поєднує підструктури. В цьому випадку спільним полем є __is_long_, яке займає тільки 1 біт і вказує на те, чи є рядок довгим.

Для кращого розуміння структури варто поглянути на рисунок:



Clang використовує функціонал union більш щедро, накладаючи один на одного різні структури. Така розподілення робить саме цю імплементацію поки

[3] https://github.com/llvm/llvm-project/blob/main/libcxx/include/string

що найефективнішою з усіх – вона використовує мінімальну кількість байтів (24) і при цьому має найбільший розмір короткого рядка з усіх – 22 символи.

Але одразу видно певну проблему – місткість рядка (cap) має не повні 8 байтів (64 біти), а 63 біти, тобто максимальна кількість символів у рядка менша вдвічі. Звісно, це не велика проблема, бо 63 біти дозволяють місткості набувати максимального значення у 9 квінтильйонів. Очевидно, такої кількості символів не буде у звичайному використанні. Якщо ж все-таки доведеться користуватися неймовірно великими рядками, то розробники, очевидно, не пропустять цей момент і приділять йому окрему увагу.

Також мінусом такої реалізації є те, що абсолютно усі операції над рядком завжди спочатку вимагатимуть перевірки типу рядка. Поле is_long – єдина спільна частина для обох, яка якраз і виконує таку функцію, тому уся робота повинна буде розділитися на два типи.

Але цим ми платимо за настільки ефективну імплементацію – в обмін на трохи більше роботи в рантаймі, ми використовуємо менше пам’яті на рядок і можемо мати більші короткі рядки. Тобто, у випадку, коли нам потрібно працювати з величезною кількість маленьких рядків, Clang може бути одним із перших кандидатів при виборі компілятора.

3.3 Стандартна реалізація SSO. MSVC

MSVC є, напевне, лідером за використанням серед розробників, які навіть не знають, що таке компілятор. Через те що програмістам з операційною системою Windows часто рекомендують Visual Studio як IDE для розробки на C++ і MSVC за замовчуванням використовується в ньому, то, очевидно, цей компілятор часто використовується без точно розуміння в чому різниця між ним та іншими.

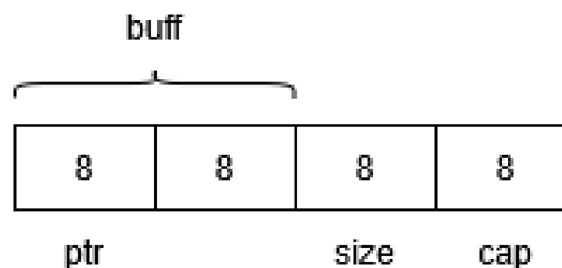
Сам компілятор є пропрієтарним, тобто він працює лише у сукупності з конкретною ОС – Windows. Це дозволило розробникам відійти від проблеми

оптимізації для різних платформ і сфокусуватися тільки на власній, тому компілятор часто використовує оптимізації, що притаманні тільки Windows.

Власне, код[4]:

```
union {
    char buff [16];
    char* ptr;
};
size_t size;
size_t cap;
```

Така імплементація дуже схожа на ту, яка ми бачили в GCC. Тільки на цей раз, поєднується не масив і місткість, а масив і покажчик на купу.



Тобто використання пам'яті є таким же, як і в GCC – 32 байти на структуру загалом та 16 байтів (15 символів) на короткий рядок.

Єдине, на що варто звернути увагу – це на те, як визначається чи є рядок коротким чи довгим. MSVC перевіряє, чи теперішня місткість більше за місткість малого рядка:

```
bool _Large_mode_engaged() const {
    return cap > _Small_string_capacity;
}
```

Очевидно, такий метод матиме вимогу до реалізації рядка: якщо місткість рядка дорівнює (або якось стала менше) місткості короткого, то це точно повинно означати, що рядок є коротким. Уявимо такий приклад: ми маємо довгий рядок розміру 20. Очевидно, він довгий і умова буде виконуватися.

---

[4] https://github.com/microsoft/STL/blob/main/stl/inc/xstring

Нехай ми видалимо в ньому символи, щоб його розмір став 13. Місткість не повинно була змінитися, умова все ще виконується. А тепер використаємо метод shrink_to_fit(), що зменшує місткість рядка до такої, що точно б вміщала розмір. І тут маємо проблему: якщо імплементація не враховує те, що рядок може змінитися з довгого на короткий (тобто не переносить дані з покажчика у масив), то отримаємо суперечність: метод показуватиме, що рядок короткий, хоча він насправді буде довгим. Звісно, таку проблему було враховано і рядок в MSVC працює правильно. Проте це непоганий приклад того, як така, на перший погляд, мала зміна у структурі призвела до геть іншої роботи з рядком та тим, наскільки важливим стала імплементація такого непопулярного методу.

3.4 Оптимізація перетворенням останнього байту. Folly

Найгеніальнішою імплементацією серед усіх є, напевне, рядок з бібліотеки folly. Цю бібліотеку розробила компанія Facebook. Очевидно, настільки велика, глобальна корпорацію, що оперує гігабайтами даних на секунду, хотіла б максимально оптимізувати свій код. Звісно, їхні оптимізації не були неймовірно значущими, проте їх було багато. У 2016 році відбувся збір “CppCon”. На ньому виступав Ніколас Ормрод[5] – один із розробників. У своєму виступі він якраз оглядав реалізацію рядка у їхній бібліотеці. “Ці малі покращення працюють по всьому Facebook’у… Ми маємо велику кількість таких малих покращень, що з часом накопичуються і виливаються у один великий виграш у швидкодії.” – так він виправдав існування цих покращень.

Реалізація[6] рядка є одним із таких. Її автором є Андрей Александреску. Вона цікава тим, що короткий рядок не потребує жодного додаткового поля, окрім власне масиву тексту:

```
struct MediumLarge {
    char* data_;
    size_t size_;
```

[5] https://www.youtube.com/watch?v=kPR8h4-qZdk&t=880s
[6] https://github.com/facebook/folly/blob/main/folly/FBString.h

```
    size_t capacity_;
};

union {
    char small_[sizeof(MediumLarge) / sizeof(Char)];
    MediumLarge ml_;
};
```

Бачимо, що через union поєднується структура простого рядка (MediumLarge) і масиву для короткого рядка. По-перше, таким чином рядок займатиме 24 байти і зможе мати максимальну довжину 23 символи. Очевидно, з'являється питання щодо того, а як зберігати розмір короткого рядка. Тут використовується неймовірно цікава техніка – зберігання залишку місткості рядка в останньому байті масиву. Тобто, нехай ми матимемо текст "lorem" у рядка. Тоді, у останній байт поміститься число 18 (23 - 5):

l	o	r	e	m	\0																		18

При додаванні тексту так само буде і зменшуватися останній байт:

l	o	r	e	m		i	p	s	u	m	\0												12

І найцікавіший момент: при наближенні рядка до ліміту останній байт буде зменшуватися. Таким чином, коли рядок набуде максимальної довжини, останній байт буде дорівнювати нулю (відсутній залишок місткості). Проте значення 0 також є значенням null-символу, таким чином ми автоматично маємо його у рядку:

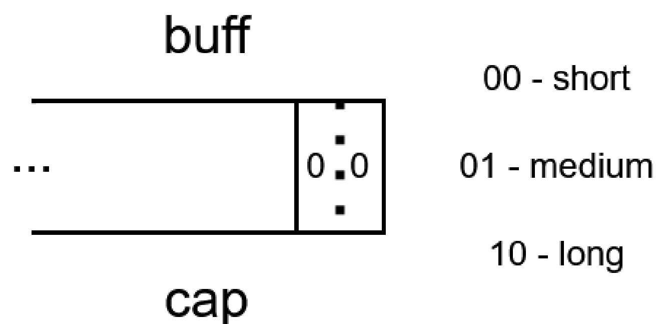
l	o	r	e	m		i	p	s	u	m		d	o	l	o	r		s	i	t		\0	1

l	o	r	e	m		i	p	s	u	m		d	o	l	o	r		s	i	t		a	0

= \0

Така структура дозволяє нам повністю відкинути будь-які додаткові поля у короткому рядка. Якщо нам потрібно буде дізнатися розмір такого рядка, то від місткості рядка ми віднімемо останній байт. Для будь-яких дій із рядком, спочатку потрібно буде перевірити тип самого рядка.

Залишилося тільки зрозуміти, як розрізняти рядки різного типу. Цим цей рядок схожий на Clang. Останні 2 біти масиву (або 2 біти місткості) будуть зарезервовані для визначення типу. Через це ми матимемо два наслідки. Місткість буде в 4 рази меншою, що все ще не є проблемою для звичайних застосунків, бо максимальна місткість все ще буде достатньо великою. Друге - це те, що оскільки останній байт у малому рядка повинен дорівнювати нулю при максимальному розмірі, то і останні 2 біти так само повинні бути нульовими. І звісно ж тепер операція як отримання розміру короткого рядка або місткості довгого передбачуватимуть побітові операції. Вони не сильно вдарять по швидкодії, але явно знизять читабельність коду.



Ця імплементація використовує два біти замість одного, оскільки, окрім малого та звичайного, в ньому також присутній тип великого рядка. Цей великий рядок використовує принцип COW, який ми уже розглядали. При збереженні рядка в купі виділяється на один байт більше, ніж потрібно, використовуючи один зайвий байт для лічильника посилань.

Рядок буде ініціалізовано як довгий, коли його довжина перевищує 255 символи. А оскільки рядків такого розміру доволі мало, то COW буде використовуватися не дуже часто, уникаючи того частої появи вже згаданих проблем.

3.5 Поєднання імплементацій GCC та Folly. Власна

Оглянувши усі найвідоміші приклади, ми бачимо, наскільки різними бувають реалізації однієї і тієї самої оптимізації. Оскільки union дозволяє поєднувати будь-які дані, а поле місткості можна опустити для короткого, виникла така різноманітність.

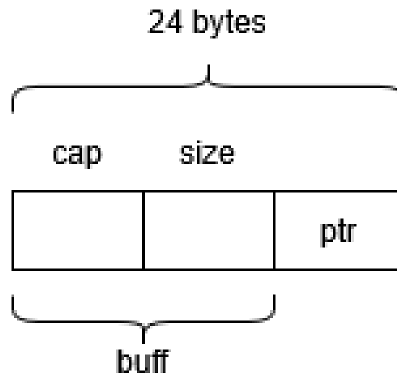
Тому в рамках роботи також була створена власна імплементація рядка, що поєднує ідеї з двох інших: GCC та Folly. А саме, окремий покажчик, що завжди показуватиме на правильні дані та збереження залишку місткості в останньому байті масиву.

Оглянемо сам код:

```
class short_string {
    char* _pointer;
    union {
        struct {
           std::size_t _capacity;
            std::size_t _size;
        } _long;
        char _buff[16];
    };
}
```

Ця імплементація має декілька переваг над іншими. По-перше, у структурі union ми не “втрачаємо” байти, як в GCC, оскільки тут використовується буфер на 16 байтів та структура з двох полів типу size_t, що в сумі також дорівнює 16 байтів. Сам клас займає 24 байти, як і оригінальний, тобто ми не витрачаємо більше пам’яті. Але розмір малого рядка тепер знову 15 символів. Така структура “витискає” максимум із union, при цьому не робить ідею занадто складною; не потрібно використовувати побітові операції.

В цій структурі використовуються реалізації обох бібліотек. При перевірці типу рядка будуть порівнюватися показник і локальний буфер, як в GCC. Коли ж потрібно буде дізнатися розмір короткого рядка, то буде використовуватися останній байт буферу, як в Folly.



Також варто відзначити копіювальний конструктор у такій імплементації, а саме копіювання у випадку короткого рядка

```
short_string(const short_string& str) {
    if (str.is_long()) {
        // long copy
    }
    else {
        _long._size = str._long._size;
        _long._capacity = str._long._capacity;
        _pointer = _buff;
    }
}
```

В такому випадку ми копіюватимемо дані з масиву, що знаходиться на стеці. Але оскільки ці самі дані збережені у size та capacity у структурі довгого рядка, то замість того, щоб копіювати через цикл усі елементи, ми можемо просто скопіювати ці поля, а потім призначити покажчику значення буфера.

4 ПОРІВНЯННЯ ІМПЛЕМЕНТАЦІЙ SSO. ТЕСТИ

Розглянувши усі імплементації, ми побачили наскільки різноманітними вони можуть бути. Малі зміни призводять до того, що варто повністю переосмислювати структуру та її внутрішню роботу.

Хотілося б порівняти їх у дії, який вплив вони мають на швидкодію. Тому був обраний список завдань, в яких вони будуть порівнюватися.

В тестах будуть брати участь усі вищезгадані бібліотеки з їх відповідними компіляторами. Звісно, можна сказати, що використання різних бібліотек з різними компіляторами призведе до різних результатів, оскільки кожен компілятор має свої оптимізації, що можуть вплинути на результати. Проте ці бібліотеки були створені якраз для використання зі своїми компіляторами, а їх використання в інших зазвичай призводить до великої кількості проблем і загалом не є рекомендованим. Таким чином, ми тестуватимемо бібліотеки у своєму "природному" середовищі.

Бібліотека Folly буде тестуватися з MSVC. Також буде використана старша версія GCC 4.8.4, де все ще використовувався COW. Власна імплементація рядка буде тестуватися на GCC та MSVC, щоб порівняти сам код та вплив компіляторів на швидкодію. При компіляції коду на GCC 13.3 та Clang будуть використовуватися прапорці "-O2 -DNDEBUG", оскільки вони наблизять компіляцію до рівня "Release" в MSVC.

Апаратне та програмне забезпечення:

- CPU: AMD Ryzen 5 5500
- RAM: 32GB, 3200MHz, CL16
- OS: Windows 10

Для тестування старої і нової версій GCC буде використаний WSL2 (Windows Subsystem for Linux), оскільки тільки з цим можна було повноцінно отримати ці версії компіляторів без повної зміни операційної системи. Тести [12] показують, що в загальному випадку вплив такої віртуалізації є

мінімальним. Також, оскільки GCC створений саме для Linux, а на Windows, то він потребував би додаткового шару сумісності (зазвичай, MinGW).

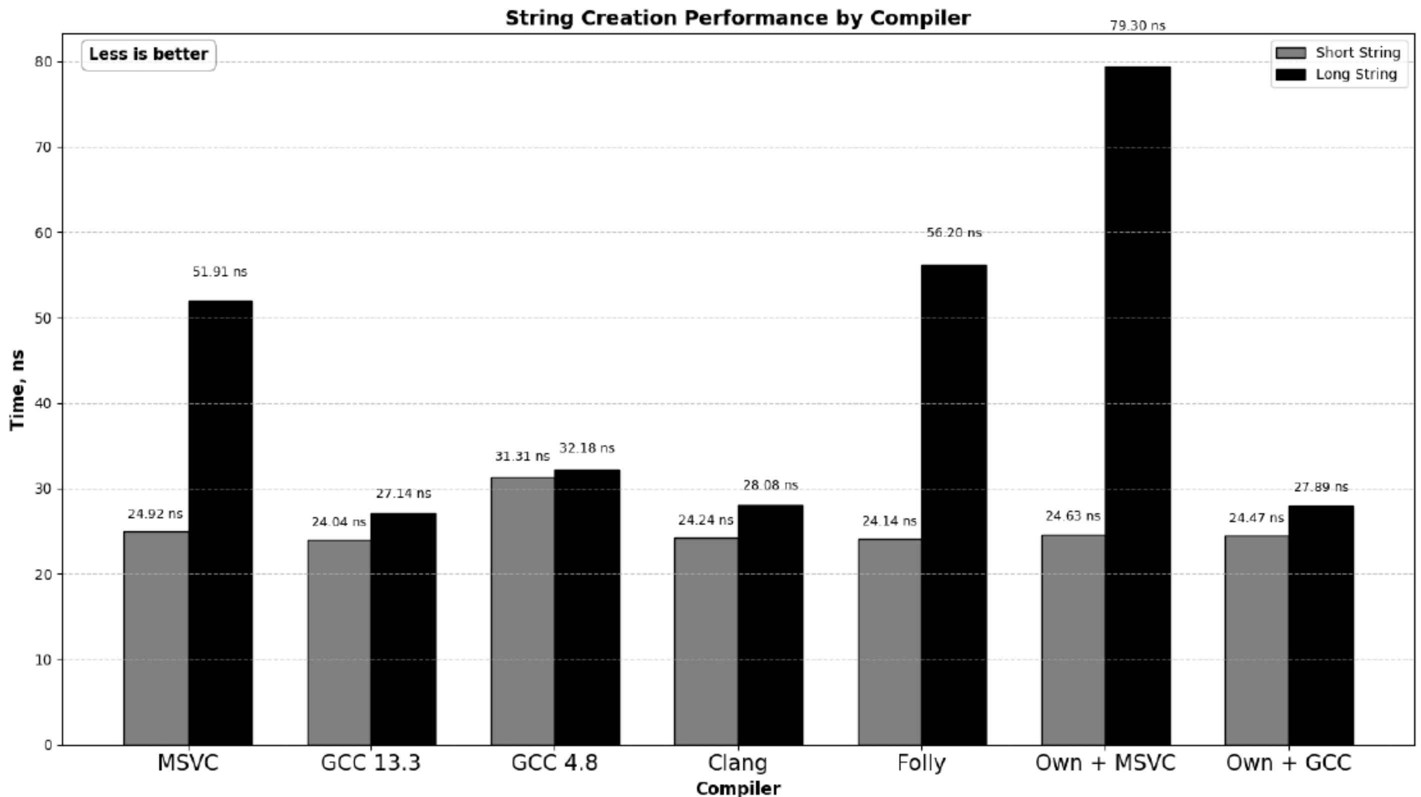
4.1 Тестування створення рядка різної довжини

Першим та найпростішим тестом буде створення рядка. Код тесту має такий вигляд:

```
void test_create(unsigned long long amount, const  char* testing) {
    high_resolution_clock::time_point start, end;

    long long dur = 0;

    for (unsigned long long i = 0; i < amount; ++i) {
        start = high_resolution_clock::now();
        std::string copy{ testing };
        end = high_resolution_clock::now();
        dur += (end - start).count();
    }

    std::cout << "string copy: " << dur << " ns" << std::endl;
    std::cout << "string copy average: " << 1.0 * dur / amount << " ns" <<
std::endl;
}
```

Використовуючи стандартну бібліотеку C++ “chrono”, ми зможемо з точністю до сотих наносекунди порахувати кількість часу, яка знадобилась для виконання коду. Визначення початку та кінця підрахунку процедури зроблено таким чином через “dead code elimination”, оскільки компілятор би вирішив, що код з циклу ніде не використовуються, то його можна відкинути. А додавши підрахунок часу в цикл, ми уникаємо цієї оптимізації.

Тестування відбуватиметься на рядках “sso” та “a very long string to surpass sso”, які повинні відображати короткі та довгі рядки відповідно.



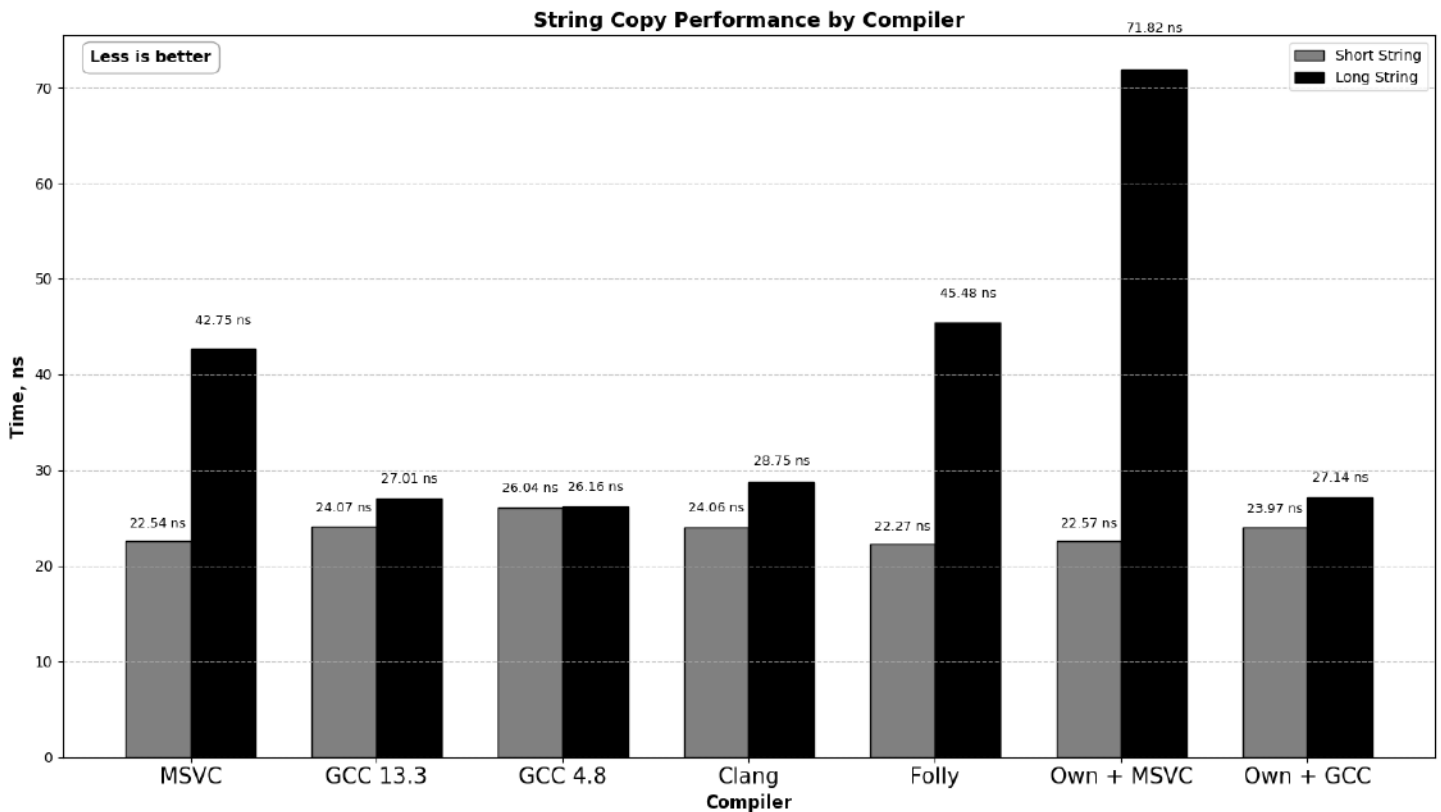
Цей результат є доволі цікавим. По-перше, в усіх тестах, окрім GCC 4.8, спостерігаємо значне збільшення часу виконання при переході від коротких до довгих рядків. Це не можна скинути на більший розмір тексту, оскільки у GCC 4.8 якраз відсутня SSO, через що різниця у двох тестах відображає вплив більшого розміру рядка, а він є доволі мізерним. По-друге, усі імплементації з SSO показують майже ідентичні результати для коротких рядків. Цікаво, що GCC 4.8 виконується лише трохи повільніше за GCC 13.3 та Clang. Таку поведінку ми також бачитимемо і в наступних тестах, де, на перший погляд, гірша імплементація матиме непогані результати. Усі тести, що проводилися з MSVC компілятором, показують сильне відставання на довгих рядках, хоча на коротких результати на рівні з іншими. Власна імплементація з MSVC програє на довгих рядках в середньому на 40 нс, при цьому з GCC показує результати на рівні з GCC 13.3 та Clang. Таке сповільнення MSVC буде з’являтися ще декілька разів.

4.2 Тестування копіювання рядка різної довжини

Цей тест буде використовувати ті самі рядки, але на цей раз для копіювання.

```
std::string copy = testing;
```

Наперед можна передбачити, що GCC 4.8 буде працювати краще всіх, оскільки саме він повинен оптимізувати цю операцію.



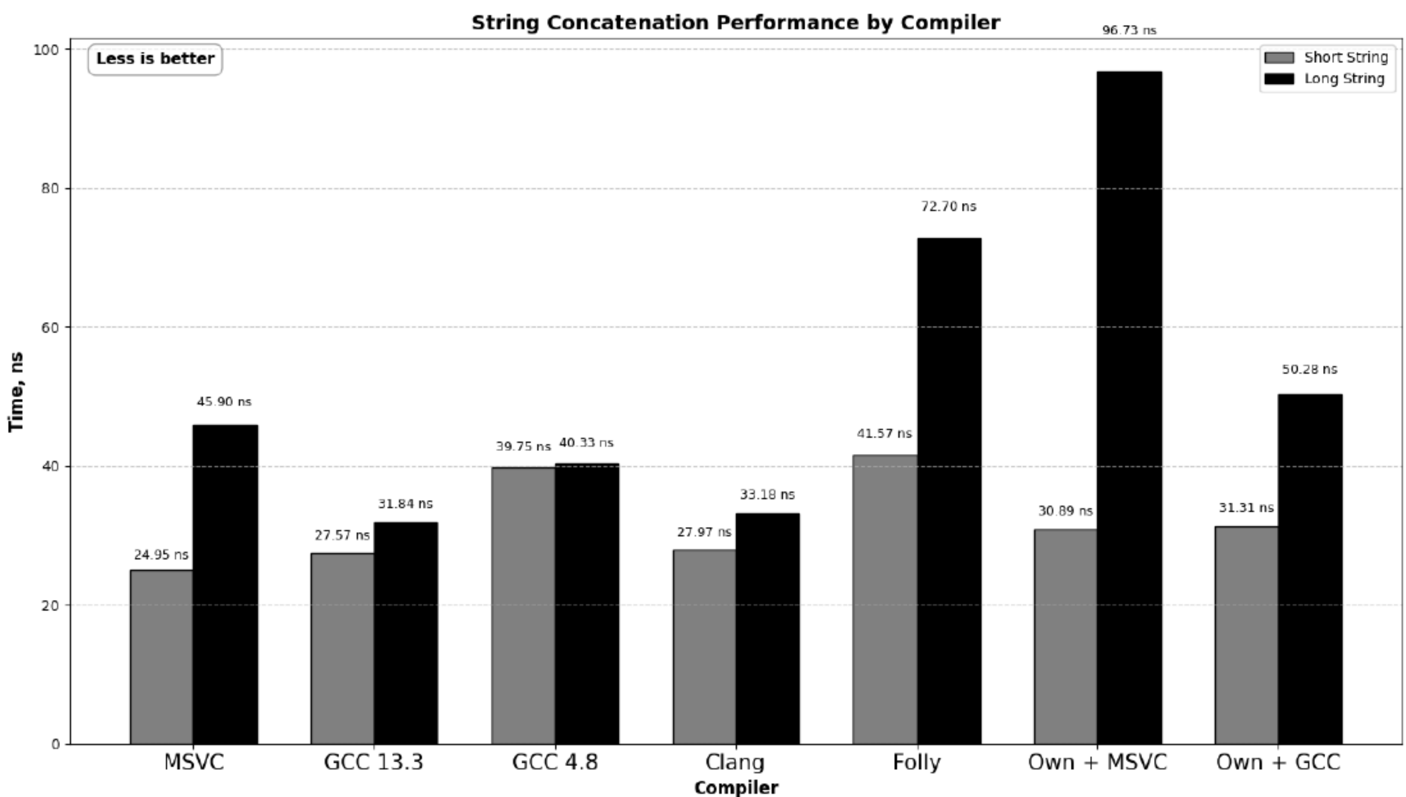
Справді, GCC 4.8 має найкращі покажчики при копіюванні великих рядків. Проте такий самий час займає і копіювання малих (бо копіюється лише покажчик), при цьому компілятори з SSO показують в цьому випадку кращий час. Усі реалізації з SSO так само мають схожі результати на коротких рядках з різницею у 1-2 нс. Власна імплементація так само має непоганий час, але сильно сповільнюється з використанням MSVC, як і в попередньому прикладі.

Тестування конкатенації рядків різної довжини

Конкатенація також є доволі важливою операцією, з якою класам рядків може бути важко. Тепер код матиме вигляд:

```
std::string copy = testing + testing;
```

Додавання коротких все ще дасть нам короткий рядок довжиною 6 символів, а довгі так само створять один довгий.



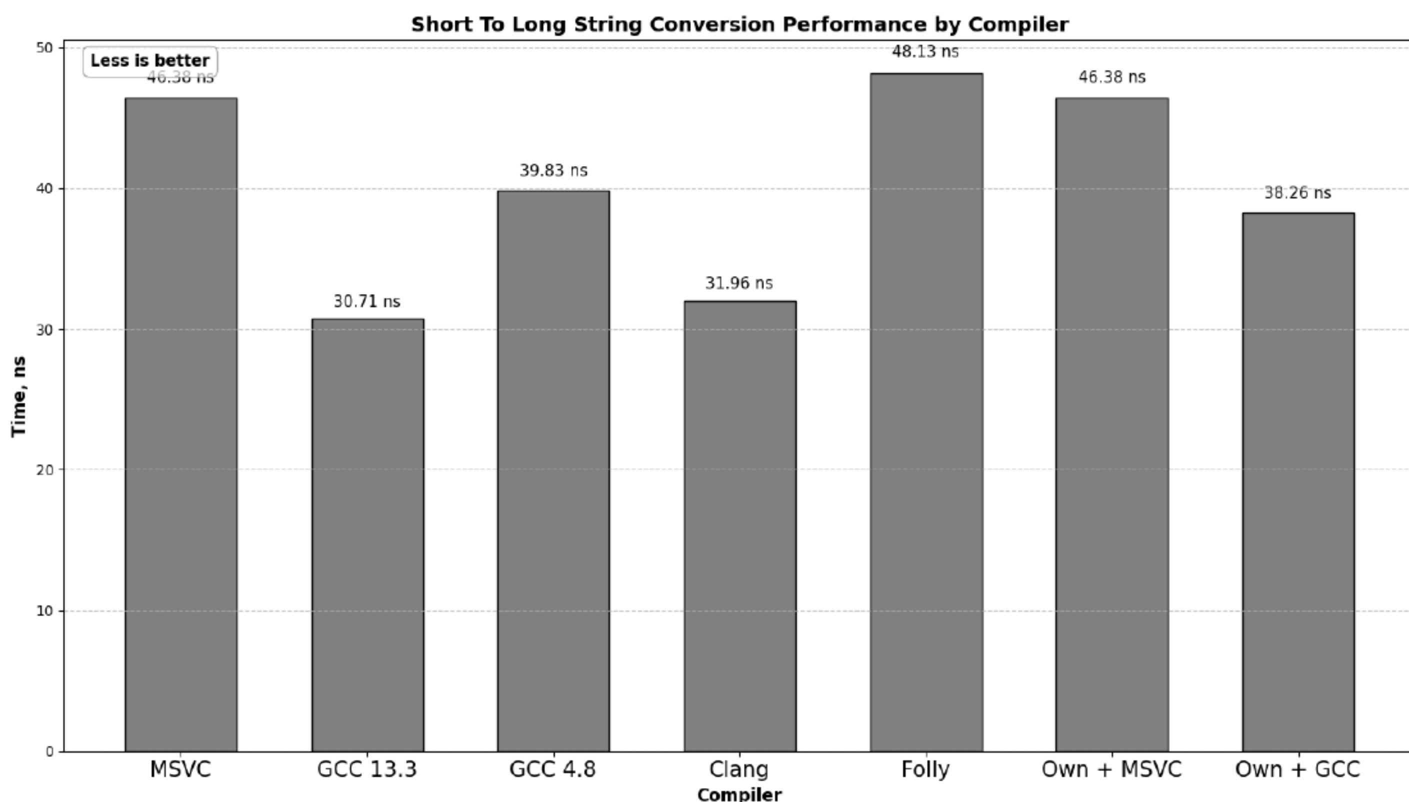
Результати очікувані: GCC 13.3 та Clang мають найкращий час, GCC 4.8 показує трохи гірші результати, реалізації з MSVC сильно сповільнюють довгі рядки. Варто окремо відмітити те, що Folly на диво погано показує себе в усіх варіантах, будучи найповільнішим з усіх. Власна реалізація показує себе посередньо на довгих рядках.

4.3 Тестування перетворення рядка з короткого у довгий

Однією із особливостей SSO є операція перетворення рядка з короткого у довгий. Це теж займає певний час, і вона може викликатися непомітно при роботі з рядками.

```
testing.resize(40);
```

Опираючись на попередні тести, можна передбачити, що власна імплементація та Folly матимуть гірший час. GCC 4.8 буде трохи повільнішою за GCC 13.3 оскільки у обох випадках створюватиметься масив у купі з подальшим копіюванням.



Загалом очікування підтвердились. GCC та Clang виявилися найшвидшим, MSVC на рівні з Folly та власною імплементацією. Аналогічно до попередніх тестів, власний рядок з GCC 13.3 працює швидше.

4.4 Тестування рядка алгоритмом Кнута-Морріса-Пратта

Розглянуті тести можна назвати “штучними”. Вони перевіряють лише окремі аспекти імплементацій, виділяючи певні прогалини. Проте так рядки не використовуються, варто протестувати їх у реальних ситуаціях.

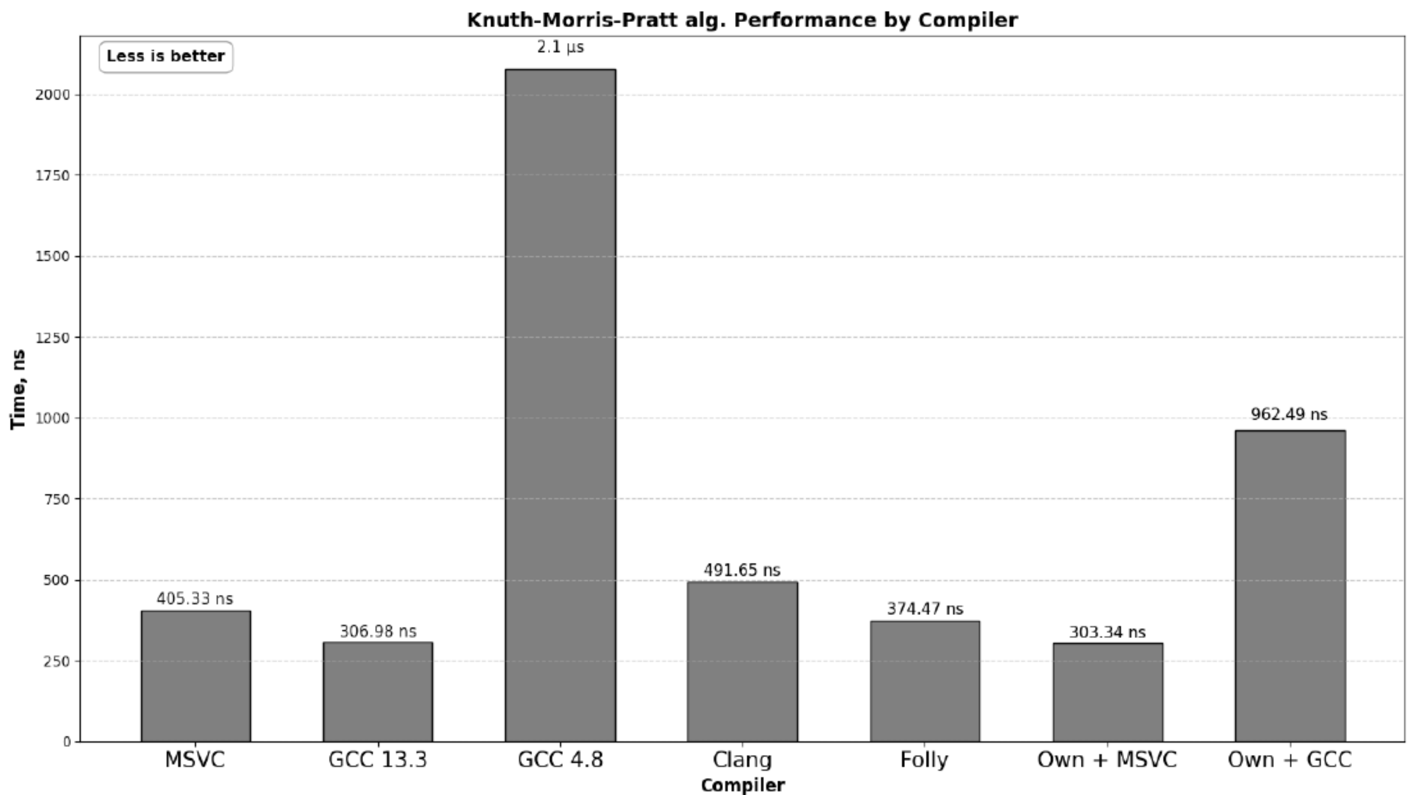
Тому буде використано алгоритм Кнута-Морріса-Пратта. Він використовується для швидкого пошуку підрядка у заданому рядку. У Linux він використовується у команді “grep” [13], що і виконує функцію пошуку рядка у файлах. Його перевагою є те, що перед початком роботи, він створює додатковий масив, в який вкладає певні числа. Ці числа вказують, на довжину префіксу шуканого рядка, яку можна пропустити, бо вона вдруге зустрілася у шуканому. Для кращого розуміння розглянемо рядок “ABCABCDA” та його додатковий масив:

A	B	C	A	B	C	D	A

0	0	0	1	2	3	0	1

Тепер можна краще зрозуміти: для кожного символу рядка число у додатковому масиві вказує на те, скільки символів ми можемо пропустити у шуканому. Якщо ми побачимо, що перші 4 символи з рядком, в якому шукають, рівні з шуканим, а 5-ий буде не рівний, то алгоритм погляне у 4-ий елемент додаткового масиву та виявить, що 4-ий символ є префіксом шуканого і його можна пропустити і перевіряти вже з другого.

Для цього тесту був обраний повний текст “Гамлета” англійською, і в ньому шукалося слово “retrieve”.



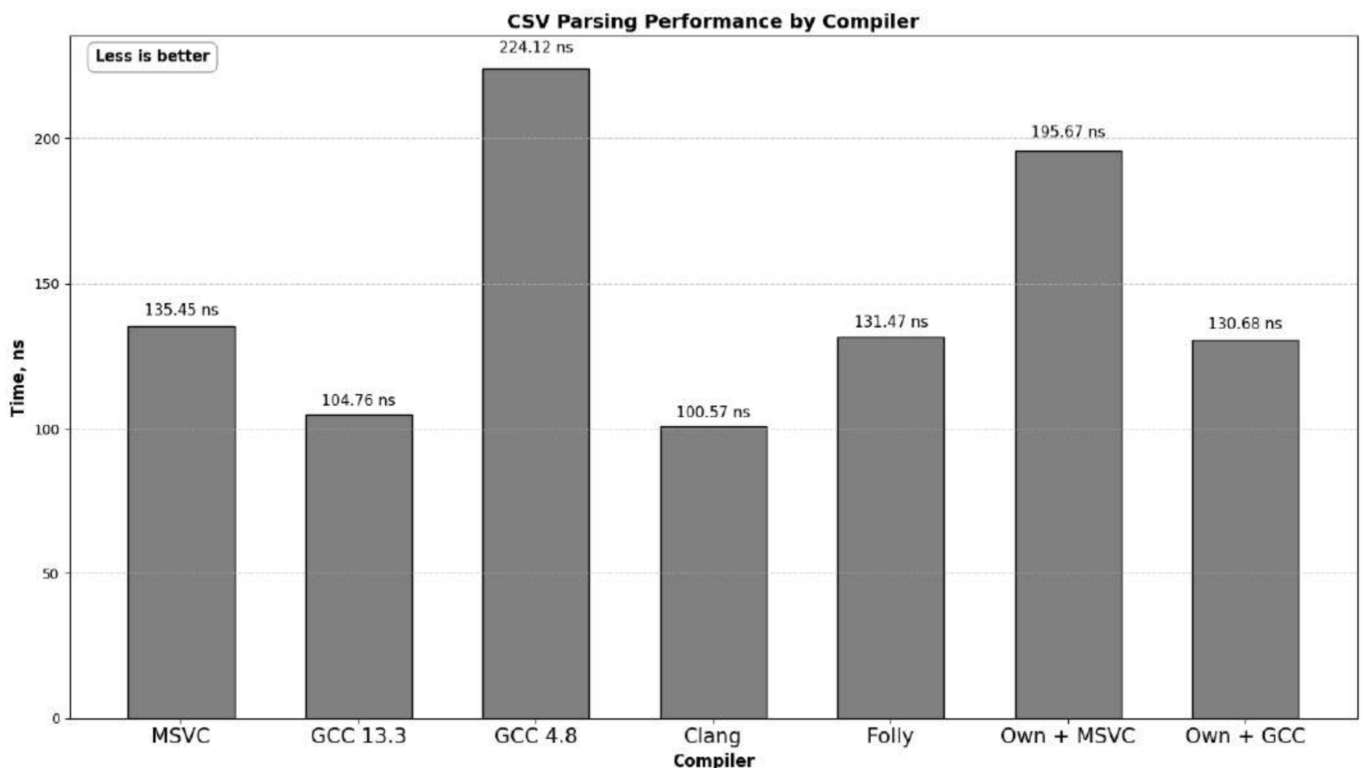
Очікуваною є поведінка GCC 4.8. Для нього потрібно увесь час звертатися до рядка в купі. Власна імплементація приємно дивує та має найбільшу швидкість, будучи трохи швидшим за GCC 13.3. З іншого боку, якщо порівнювати власну імплементацію з MSVC та з GCC 13.3, то другий набагато повільніший. Тут демонструється поведінка обернена до попередніх тестів, тепер новий GCC чомусь сильно сповільнює код. Скоріше за все, це пов’язано з певними прапорцями, які не були увімкнуті.

4.5 Тестування рядка парсингом CSV

Ще одним із способів використання рядків є читання CSV (comma separated value / значення, розділені комою) файлів. Вони зазвичай містять велику кількість інформації, що комами розділена на відносно короткі рядки.

Для тесту було використано CSV-файл з 100000 рядками, в кожному з яких є по 7 рядків довжиною від 3 до 40 символів, що складаються тільки з букв англійського алфавіту.

Парсер читатиме файл один рядок за раз. Потім він буде проглядати рядок посимвольно і, зустрівши кому, додаватиме зчитане слово до масиву



В цьому випадку результат є найбільш очікуваним. GCC та Clang мають приблизно однакові результати. GCC 4.8 є дуже повільним. Folly, MSVC та власна імплементація з GCC знаходяться на одному рівні. MSVC продовжує тенденцію та сповільнює власний рядок.

4.6 Підсумки результатів

SSO справді позитивно впливає на швидкодію. У всіх завданнях SSO надавав на 10-50% кращі результати. Звісно, це залежало від компілятора та бібліотеки, але це дає змогу впевнитися у беззаперечній перевазі SSO.

Проте так само було помічено, на диво, непогану швидкість COW імплементації. У більшості "штучних" тестів вона показала себе краще у випадку довгих рядків та поступалася лише до 10 нс з короткими. Можна зробити висновок, що "прямолінійна" природа COW дозволяє йому дуже швидко виконувати прості операції, тоді як SSO зазвичай вимагає великої кількості перевірок для однієї.

Власна імплементація показала себе дуже гарно. В більшості тестів ця імплементація мала такий самий або навіть кращий результат. Лише у випадках роботи з довгими рядками з MSVC власна реалізація сповільнювалась, але така поведінка була присутня у всіх результатах з MSVC. Можна зробити висновок, що проблема полягає в роботі з пам'яттю у купі. Різні компілятори по-своєму оптимізують доступ до неї, тим самим видаючи настільки несхожі результати.

Загалом, оглядаючи результати тестів, можна дійти певного висновку. SSO є корисним та повністю виправдовує своє існування. З огляду на власну імплементацію та Folly можливо варто було б створити іншу реалізацію рядка, що імплементувала б саме їхні ідеї, але при цьому ще краще співпрацювала з компілятором для роботи з довгими рядками. Ця ідея справді може бути корисною, оскільки власна імплементація в окремих завданнях обганяє усі інші. Проте це також можна назвати "вигадуванням велосипеду". Імплементації рядків у межах компіляторів розробляються не один рік, і перші дуже тісно співпрацюють з другими. Розробляти власні версії рядків потрібно лише якщо існує безпосередня потреба у оптимізації конкретних задач, наприклад оптимізація використання пам'яті при читанні великої кількості малих рядків.

ВИСНОВОК

У роботі було розглянуто рядки та їхні імплементації у розповсюджених компіляторах і бібліотеках. Було приділено особливу увагу реалізаціям SSO.

Додатково була запропонована та створена власна імплементація оптимізації коротких рядків, що є більш конкурентно спроможною при використанні коротких рядків. Найголовніше, було проведено ряд тестів задля визначення впливу оптимізацій на швидкодію у окремих ізольованих завданнях та більш загальних задачах.

Проведений обчислювальний експеримент продемонстрував ефективність SSO в усіх випадках. Середній приріст швидкодії лежить в межах 10-50%, при цьому варіативність самої імплементації демонструє незначний вплив на продуктивність програм. Для містких завдань SSO пришвидшує виконання на 33%, порівняно з COW.

Власна імплементація мала аналогічний до наявних прикладів приріст у швидкодії, іноді навіть обганяючи популярні рішення. Зокрема власна реалізація перемагає у створенні, копіюванні, додаванні та пошуку підрядка. Тому вона рекомендована при роботі з короткими рядками, як ефективніша для виконання задач такого типу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] R. Clark, «The UCSD Pascal handbook : a reference and guidebook for programmers,» 1982.

[2] ISO/IEC, «Extended Pascal,» International Organization for Standardization, 1990.

[3] D. Ritchie та K. Thompson, «The UNIX Time-Sharing System,» 1974.

[4] M. McIlroy, E. Pinson та B. Tague, «UNIX Time-Sharing System: Forward,» *Bell System Technical Journal,* т. 57, № 6, p. 6, 1978.

[5] H.-J. Boehm, R. Atkinson та M. Plass, «Ropes: an Alternative to Strings,» John Wiley & Sons, Ltd., 1995.

[6] P. Lyu, «Text Buffer Reimplementation,» 2018.

[7] C. Crowley, «Data Structures for Text Sequences,» 1998.

[8] Core Dumped, «Text showdown: Gap Buffers vs Ropes,» 2023.

[9] A. Meredith, H. Boehm, L. Crowl, P. Dimov та D. Krügler, «Concurrency Modifications to Basic String,» ISO/IEC, 2008.

[10] «The C++ Standard Template Library. Student Manual».

[11] « ISO/IEC 14882:2024,» ISO/IEC, 2024.

[12] M. Larabel, «Windows 11 WSL2 Performance vs. Ubuntu Linux With The AMD Ryzen 7 7800X3D,» 2023.

[13] M. Haertel, «why GNU grep is fast,» 2010.

[14] R. Chen, «An informal comparison of the three major implementations of std::string,» 2024.