СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Котелянець В. В. Інформаційна технологія моніторингу навколишнього середовища на базі концепції інтернету речей. МОН України, ЧДТУ. Черкаси, 2019.
2. Використання дисперсії Аллана для ідентифікації нормальної роботи сенсорних вузлів / В. Г. Крижановський, В. Ф. Комаров, С. П. Сергієнко, Л. В. Загоруйко. *Вісник Вінницького політехнічного інституту.* 2021. № 3.
3. Non-Stationary Noise Analysis of Magnetic Sensors using Allan Variance / K. Draganová, V. Moucha, T. Volcko, K. Semrád. *Acta Physica Polonica.* 2017. Vol. 131(4). P. 1126–1128.
4. Implemented statistics functions. URL: https://allantools.readthedocs.io/en/latest/functions.html
5. Python. URL: https://www.python.org/
6. Котелянець В. В. Інформаційна технологія моніторингу навколишнього середовища на базі концепції інтернету речей. МОН України, ЧДТУ. Черкаси, 2019.
7. Ємець К. Методи прогнозування часових рядів з вираженою сезонністю на основі трансформерів. *Herald of Khmelnytskyi National University. Technical Sciences.* 2024. Vol. 333(2). P. 131–134.
8. Marusenkova T. Analysis of the influence of sample rates on the allan variance. *Вісник Національного університету «Львівська політехніка». Інформаційні системи та мережі.* 2019. Вип. 5. С. 53–61.
9. Жураковський Б. Ю., Зенів І. О. Технології інтернету речей: навчальний посібник. Київ, 2021. 271 с.

УДК 004.437

ОПТИМІЗАЦІЇ РОБОТИ STD::LIST ШЛЯХОМ ВИБОРУ НАЙБІЛЬШ ЕФЕКТИВНОЇ СТРАТЕГІЇ ВИДІЛЕННЯ ПАМ'ЯТІ

М. В. Шевцов, О. С. Вєтров

Анотація. У роботі досліджується ефективність контейнера std::list з використанням різних алгоритмів виділення пам'яті. Проводиться порівняння між стандартним (std::allocator) та користувацьким аллокатором, що базується на принципах роботи stack-based allocator. Була детально розглянута теорія з цієї теми та проведене дослідження. Результати викладено у таблиці та візуалізовано у вигляді діаграми.

Ключові слова: C++, std::list, аллокатор пам'яті, StackAllocator, ефективність, оптимізація, зв'язний список.

Сучасний світ програмування визначається постійним прагненням до оптимізації та підвищення ефективності програмного забезпечення. Однією з ключових областей цього пошуку є вибір та оптимізація структур даних, які використовуються для зберігання і обробки інформації. Мова програмування C++ визначається своєю потужністю та широкими можливостями, що вона надає користувачу для роботи з пам'яттю. Водночас важливою залишається вимога до програміста контролювати всі процеси виділення пам'яті (аллокації), оскільки непродумана стратегія безпосередньої роботи з пам'яттю в C++ може привести до некоректного функціонування програмного додатка та всієї операційної системи загалом.

Аллокація – це процес виділення блоків пам'яті потрібного розміру.

Для роботи з масивами інформації має виділятися пам'ять для даних, що оброблюються комп'ютером. Для виділення пам'яті під масиви змінних використовуються відповідні оператори, функції тощо. Зокрема, у мові програмування C++ виділяють способи виділення пам'яті – статичний та динамічний.

Статичне (фіксоване) виділення пам'яті: пам'ять виділяється тільки один раз під час компіляції. Розмір виділеної пам'яті є фіксованим і незмінним до кінця виконання програми.

Динамічне виділення пам'яті: використовується комбінація операторів new і delete. Оператор new виділяє пам'ять для змінної (масиву) у спеціальній ділянці пам'яті, яка називається «купа» (heap). Оператор delete звільняє виділену пам'ять. Кожному оператору new має відповідати свій оператор delete.

Динамічно виділена пам'ять не має області видимості, тобто вона залишається виділеною доти, доки її не буде явно звільнено або доки ваша програма не завершить своє виконання (і операційна система очистить усі буфери пам'яті самостійно). Однак покажчики, що використовуються для зберігання динамічно виділених адрес пам'яті, дотримуються правил області видимості звичайних змінних.

У програмах, реалізованих на C++, доступні два види пам'яті: стек і купа (heap). Керування стеком відбувається автоматично. Під час виходу змінної з області видимості відповідна їй у

стеку пам’ять звільняється. Цей механізм дає змогу розробнику не стежити за видаленням автоматичних змінних. Стек працює дуже швидко, але має обмежений розмір, який зазвичай не перевищує кількох мегабайт. Другий тип пам’яті – купа – влаштований дещо інакше. У купі об’єкти можна зберігати в довільному місці, створювати і видаляти їх у довільному порядку, а розмір купи зазвичай значно перевершує розмір стека. Водночас робота з купою відбувається значно повільніше, ніж зі стеком. До того ж об’єкти з купи не видаляються автоматично.

Розглянемо докладніше найпростіший приклад – це виділення пам’яті під масив на кучі, яке відбувається за допомогою оператора new[]. З розвитком мови на зміну класичному масиву прийшли std::vector та std::string, які інкапсулюють процес роботи з пам’яттю і самостійно виділяють та звільнюють ресурси. Ці операції виконуються за допомогою так званого аллокатора, який є одним із шаблонних параметрів усіх стандартних структур даних (крім std::array) в C++. Бібліотека описує стандартний набір вимог до аллокаторів, що є об’єктами класу, які інкапсулюють інформацію щодо моделі виділення [1]. Ця інформація включає знання типів вказівників, типу їх різниці, типу розміру об’єктів у цій моделі виділення, примітивів виділення та звільнення пам’яті для неї тощо [1]. Стандартом зазначено, що створення та знищення об’єктів виконується окремо від виділення та звільнення пам’яті. Контейнер не взаємодіє з аллокатором напряму, а робить це за допомогою особливого прошарку під назвою std::allocator_traits. Шаблон класу allocator_traits надає єдиний інтерфейс для всіх типів аллокаторів. Але важливо зазначити, що аллокатор не може бути типом, який не є класом, навіть якщо std::allocator_traits забезпечує весь необхідний інтерфейс [1]. Отже, завжди можливо створити похідний клас від аллокатора.

Отже, послідовність дій контейнеру під час роботи безпосередньо з пам’яттю така:

Cointainer -> allocator_traits -> Allocator -> operator new/ operator delete -> malloc/free -> OS

Розглянемо цю послідовність на основі контейнера std::list: std::list (це контейнер, який працює відповідно до структури даних «двозв’язковий список» і, як вже було зазначено, має шаблонний параметр аллокатор, який, як і у всіх стандартних структур даних, дорівнює std::allocator). Методи allocate та deallocate викликають оператори new та delete відповідно, що приводить до виділення на пам’яті на кучі, а методи construct та destroy викликають placement new та деструктори відповідно. Під час чергової спроби роботи з пам’яттю контейнер звернеться до std::allocator_traits, який містить у собі набір членів, одна частина з яких є аліасами, а інша – методами. Крім уже зазначених членів, які виконують зрозумілу роботу, в цьому шаблонному класі є ще допоміжна структура rebind. Вона є надзвичайно корисною у випадках структур даних, шаблонний параметр яких параметризований нестандартним типом, наприклад, у випадку std::list це Node<T>. Далі, оскільки клас std::allocator_traits є просто «директором» усіх аллокаторів, у стандартному сценарії робота передається методам самого об’єкту аллокатора, які вже звертаються до різних перевантажених версій операторів new/delete. Ці оператори, звичайно, викликають низькорівневі функції з мови C під назвою malloc (під час виділення) та free (під час звільнення), які запитують пам’ять в операційної системи.

Як уже зазначалося раніше, std::allocator_traits надає єдиний інтерфейс для всіх типів аллокаторів, а отже, можна зробити висновок, що std::allocator – це не єдина можлива стратегія виділення пам’яті для контейнера. І це правда! Користувач у змозі реалізувати свій власний алгоритм виділення пам’яті, але найголовніше, чого він має дотримуватись – це вимог стандарту C++ до аллокаторів, прописаних у параграфі 16.5.3.5 останньої версії стандарту [2]. Їх дотримання приведе до того, що std::allocator_traits зможе працювати з вашим аллокатором ідентично його роботі зі стандартним. Звернення до операційної системи з запитом пам’яті є доволі ресурсомісткою операцією, і постійний контроль за пам’яттю накладає великі витрати на загальну швидкість програми. Стандартний аллокатор працює саме за таким сценарієм, що не є ефективним. Реалізацій цієї ідеї може бути декілька, наприклад, PoolAllocator або StackAllocator [3]. Зупинимося на останній, особливість якої в тім, що блок пам’яті виділяється на стеку, як масив елементів типу char, які займають один байт. Цей підхід використовує принцип Last In, First Out (LIFO) – останній блок пам’яті, який був виділений, буде першим, що буде

звільнено. Загальна структура моделі пам'яті буде виглядати так (рис. 2). Header тут – це заголовок наступного блоку, який містить інформацію про кількість аллокованих байт та байт, які були виділені заради правильного вирівнювання. Час виконання операції виділення пам'яті для структури на рис. 2 становить $O(n \cdot m)$, де n – це розмір заголовка, а m – кількість викликів функції allocate.

Розглянемо детальніше методи аллокатора: allocate резервує блок фіксованого розміру на вершині стеку (рис. 3). Час виконання – $O(1)$. Під час виклику deallocate блок звільнюється на вершині стеку. Час виконання – $O(1)$.

Block pointer завжди вказує на початок щойно виділеної пам'яті, offset – на кінець всього виділеного блоку пам'яті, в тому числі щойно виділеного. Виділення та вивільнення пам'яті в StackAllocator у випадку, якщо ємність блоку ще не закінчилась, відбувається швидше, оскільки вам просто потрібно змінити покажчик на вершину стеку. Пам'ять, виділена в такий спосіб, резервується максимально компактно та має відсутність фрагментації, оскільки блоки пам'яті розділяють тільки заголовки та відступи вирівнювання. Головний недолік StackAllocator – це обмеженість розміром стеку, який у нашому випадку становить 8 мегабайт. Представлена реалізація передбачає виділення додаткової пам'яті на кучі у разі переповнення виділеного стекового блоку пам'яті, що захищає від аварійного завершення програми.

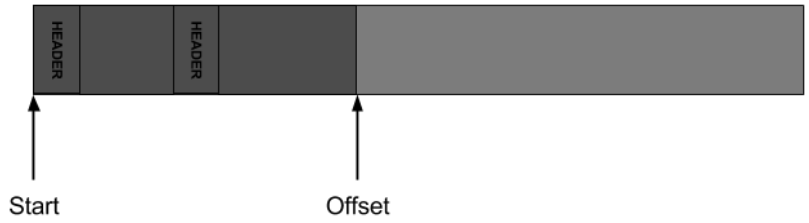

Рис. 1. Структура розміщення аллокованих даних на стеку

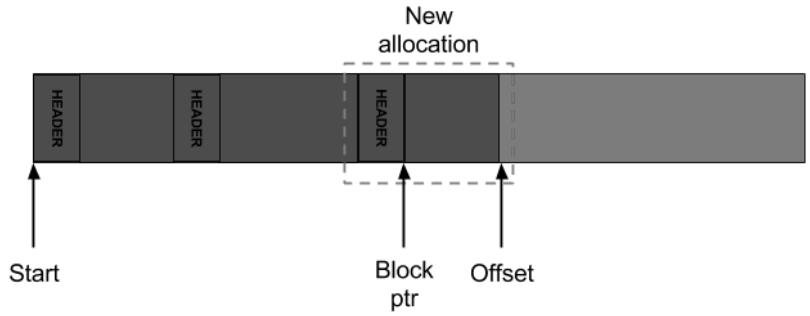

Рис. 2. Дія, яка виконується під час виділення пам'яті

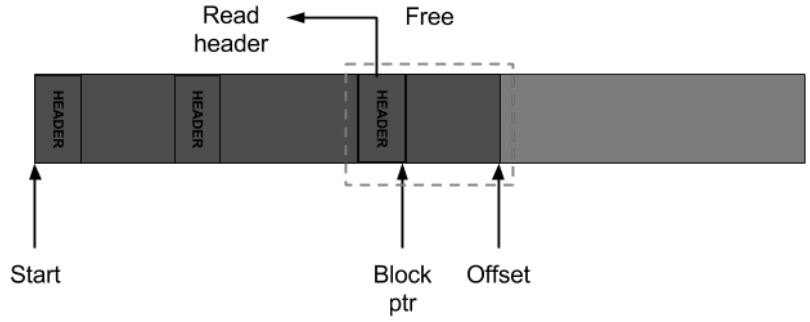

Рис. 3. Дія, яка виконується під час звільнення пам'яті

Оскільки пропонована робота стосується оптимізації стратегії виділення пам'яті, то дослідження націлене на аллокацію та деаллокацію ресурсів. Реалізований власний клас StackAllocator з дотриманням вимог стандарту. Експеримент полягає у створенні зв'язних списків на різних аллокаторах та подальшому виконанні циклу операцій, націлених на роботу з пам'яттю. Розмір так званої арени для StackAllocator ми обиратимемо максимально можливий для коректної роботи, а саме 7.5 мегабайт.

У роботі розглянуті переваги та недоліки використання динамічного та статичного способів виділення пам’яті. Динамічне виділення пам’яті, порівняно зі статичним виділенням пам’яті, дає такі переваги: а) пам’ять виділяється в міру необхідності програмним шляхом; б) немає зайвих витрат невикористаної пам’яті. Виділяється стільки пам’яті, скільки потрібно і якщо потрібно; в) можна виділяти пам’ять для масивів інформації, розмір яких свідомо невідомий. Визначення розміру масиву формується в процесі виконання програми; г) зручно здійснювати перерозподіл пам’яті, інакше кажучи, зручно виділяти новий фрагмент для одного і того ж масиву, якщо потрібно виділити додаткову пам’ять або звільнити непотрібну; д) за статичного способу виділення пам’яті важко перерозподіляти пам’ять для змінної-масиву, оскільки вона вже виділена фіксовано. У разі динамічного способу виділення це робиться просто і зручно.

Водночас перевагами статичного способу виділення пам’яті є: а) статичне (фіксоване) виділення пам’яті краще використовувати, коли розмір масиву інформації заздалегідь відомий і є незмінним протягом виконання всієї програми; б) статичне виділення пам’яті не потребує додаткових операцій звільнення за допомогою оператора delete. Звідси випливає зменшення помилок програмування. Кожному оператору new має відповідати свій оператор delete; в) природність (натуральність) подання програмного коду, який оперує статичними масивами.

Залежно від поставленого завдання програміст повинен уміти правильно визначити, який спосіб виділення пам’яті підходить для тієї чи іншої змінної (масиву).

Отже, в цій роботі було проведено експериментальне дослідження ефективності контейнера std::list з використанням різних алгоритмів виділення пам’яті. Зокрема, порівняння було проведено між стандартним (std::allocator) та користувацьким аллокатором, що базується на принципах роботи StackAllocator. Результати дослідження показали, що з кількістю елементів, які вміщуються у виділений на стеку блок пам’яті, час виконання операцій зв’язного списку на StackAllocator зменшується в середньому в 2,5 рази. Зі збільшенням кількості аллокованих елементів час виконання обох алгоритмів виділення пам’яті стає майже рівним з невеликою перевагою в сторону StackAllocator. Це свідчить про те, що наша оптимізована стратегія виділення пам’яті може забезпечити покращену ефективність контейнера у випадках, коли сумарна пам’ять, зайнята елементами списку, менше розміру виділеного на стеку блоку пам’яті.

Abstract. The paper investigates the efficiency of the std::list container using different memory allocation algorithms. A comparison is made between the standard (std::allocator) and a custom allocator based on stack-based allocator principles. The theory on this topic is thoroughly examined, and research is conducted. The results are presented in tabular form and visualized in the form of a diagram.

Keywords: C++, std::list, memory allocator, StackAllocator, efficiency, optimization, linked list.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. C++23 standard. ISO/IEC JTC1 SC22 WG21 N 4860. 2023. 1841 p.

2. Meyers S. Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley. 2001. 226 p.

3. Langer A., Kreft K. Allocator Types. C++ Report, 1998. URL: http://www.angelikalanger.com/Articles/C++Report/Allocators/Allocators.html

УДК 711.4

ІННОВАЦІЙНІ ПІДХОДИ ДО РОЗВИТКУ КУЛЬТУРНОГО СЕКТОРУ МІСТА ВІННИЦІ

М. В. Шовдра, О. М. Анісімова

Анотація. У статті розглянуто сучасні інноваційні підходи для розвитку культурного міста Вінниці. Акцентується на необхідності адаптації культурної сфери до викликів сучасного світу, поєднання збереження традицій зі впровадженням інноваційних технологій та форматів. Розкрито роль культури в розвитку міста, її вплив на формування ідентичності і привабливості для туристів та інвесторів. На основі аналізу сучасних підходів до розвитку культурного сектору розроблено пропозиції щодо впровадження ефективних інструментів модернізації культурного сектору у місті Вінниці.

Ключові слова: інноваційні підходи, розвиток культури, культурний сектор міста, культура.