Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

# Створення засобів ефективного управління пам'яттю

**Текстова частина до курсової роботи**
**за спеціальністю „ Інженерія програмного забезпечення”**

Керівник курсової роботи
доц. Бублик В. В.

_____
*(підпис)*
“____” _____ 2021 р.

Виконав студент Іванюк Н. О.

“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри мультимедійних систем,
доц.
_____ В. В. Бублик
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Іванюку Назару факультету Інформатики 3-го курсу
ТЕМА Створення засобів ефективного управління пам'яттю
Вихідні дані: Застосунок для здійснення автоматичного управління пам’яттю
Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1 Опис існуючих методів
2 Опис патерну, запропонованого Зінковичем Андрієм
3 Опис власної реалізації патерну
4 Виняткові ситуації
5 Порівняння швидкодії різних способів створення та видалення об’єктів
Висновки
Список літератури
Додатки (за необхідністю)

Дата видачі „___” _____ 2021 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

**Тема:**

Створення засобів ефективного управління пам'яттю

## Календарний план виконання роботи:

| № п/п | Назва етапу курсової роботи | Термін виконання етапу | Примітка |
|---|---|---|---|
| 1. | Огляд технічної літератури для аналізу теми роботи | 01.11.2020 | |
| 2. | Аналіз та ознайомлення з літературою | 9.01.2021 | |
| 3. | Аналіз першоджерела | 10.01.2021 | |
| 4. | Розробка алгоритму | 7.02.2021 | |
| 5. | Аналіз отриманих результатів, написання текстової частини | 22.04.2021 | |
| 6. | Корегування і остаточне оформлення роботи | 15.05.2021 | |
| 7. | Створення і оформлення презентації | 16.05.2021 | |

Студент _____

Керівник _____

“_____”_____

# Зміст

# Вступ

Робота з пам’яттю є особливістю кожної мови програмування. Мова C++ орієнтована на те, що програміст повинен керувати створенням та видаленням об’єктів, переважно, вручну. Це призводить до таких проблем як витоки пам’яті[1], завислі вказівники[2], звернення до ще не створеної, або вже звільненої ділянки пам’яті. Першим інструментом, направленим на вирішення цих проблем стали розумні указники - Smart Pointers. Проте, розв’язання ситуація взаємо зациклених вказівників все ще є досить актуальною, та до кінця не вирішеною проблемою. Під проблемою взаємо зациклених вказівників мається на увазі, що існують об’єкти, до яких неможливо доступитися, адже зв’язок до них був розірваний, проте циклічні посилання між ними перешкоджають звільненню цих об’єктів. Дану проблему частково вирішують сильні та слабкі посилання(див. розділ ‘Опис існуючих методів’). Однак при ускладненні програми побудувати правильну архітектуру стає досить складно, навіть, для кваліфікованих програмістів. Також складнощі виникають при створенні великої кількості об’єктів, які хаотично переплітаються та роз’єднуються під час виконання програми, наприклад, при симуляції рідин, де сама речовина представляється як безліч малих об’єктів, які під впливом один одного та зовнішніх чинників з’єднуються та роз’єднують між собою. В такій ситуації автоматично розставити сильні та слабкі залежності стає досить проблематично. Моя курсова робота має за мету опис патерну контролю за виділенням та звільненням пам’яті, та практичне створення централізованого блоку управління пам’яттю, який би автоматично визначав доступність об’єктів та брав на себе їх видалення при виникненні ситуації, коли об’єкти стають ізольованими від основної програми. За основу я взяв метод, запропонований Зінковичем Андрієм в одній з його статей на інтернет-ресурсі habr.com[3]. Запропонований метод вводить

означення ‘опорних вказівників’, та ‘внутрішніх’(див. розділ ‘Опис патерну, запропонованого Зінковичем Андрієм’). Таким чином, фактором досяжності об’єкту є те, чи можна до його ділянки пам’яті доступитися шляхом посилань від довільного опорного вказівника. Наявний код із статті, який описує цей патерн наведений лише для теоретичної демонстрації роботи, не є приведеним до канонічного вигляду та оптимізованим. Тому надалі в роботі будуть запропоновані методи оптимізації виконання програми, та покращення користувацького інтерфейсу для роботи з патерном.

# Опис існуючих методів

### *Стандартна робота з пам'яттю, використовуючи оператори new та delete*

Першим методом управління пам'яттю, який вивчають програмісти-початківці при вивчені мови C++ є стандартні операції для роботи з створенням та звільненням об'єктів new та delete[4], відповідно. Даний спосіб управління над пам'яттю перекладає всю відповідальність на плечі програміста. Таким чином, якщо десь був створений об'єкт через виклик оператора new, надалі в програмі повинен міститися виклик оператора delete для даного об'єкту, який би повертав раніше зайняту пам'ять назад у програму. Користуючись цим методом, програміст найймовірніше може допустити помилки, такі як: витоки пам'яті, розіменування нульових указників, доступ до вже видаленої пам'яті, або видалення вже звільненого блоку пам'яті. Таким чином, у високорівневих проектах переважно відмовляються від використання стандартних операторів для роботи з пам'яттю, на користь Smart Pointers[5]. Роботу з стандартними операторами можна зустріти переважно в проектах початківців, низьковірневих застосунках, або ж в тих частинах діючих прототипів програмного забезпечення, де повинна досягатись максимальна швидкодія.

### *Smart Pointers*

Наразі, у мові C++ основним автоматизованим методом роботи з пам'яттю є розумні указники - Smart Pointers. Це дуже потужний, інтегрований з стандартною бібліотекою шаблонів[6] інструмент для автоматичного відслідковування та звільнення пам'яті й видалення об'єктів, та забезпечення надійності виняткових ситуацій. Існують різні

типи розумних указників, робота яких відрізняється, проте детально буде розглядатися лише їх різновид std::shared_ptr[7]. Даний підвид допускає можливість посилання на об’єкт декількох вказівників, підраховуючи кількість посилань на цей об’єкт. При значенні лічильника посилань рівному нулеві, віддається сигнал, що даний об’єкт можливо видалити, а виділену під нього пам’ять звільнити й віддати назад програмі. Проте одразу ж виникає наступна проблема, а саме: розв’язування ситуацій циклічної залежності.

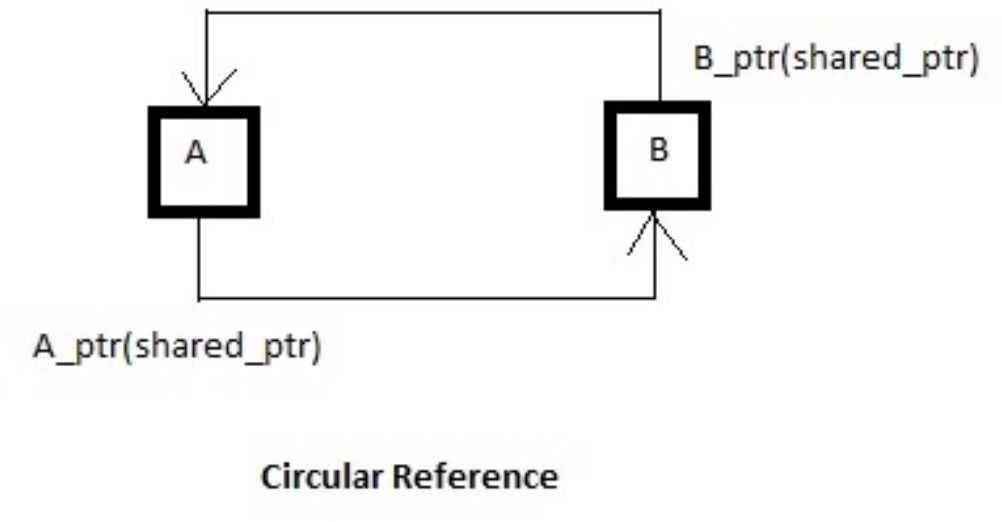


*Рисунок 1 - приклад циклічної залежності указників [https://www.geeksforgeeks.org/auto_ptr-unique_ptr-shared_ptr-weak_ptr-2/]*

Як видно з рисунку 1, лічильник завжди буде рівним 1, та об’єкти ніколи не зможуть видалитися. Для вирішення цієї задачі було введено поняття слабких посилань, тобто тих, які мають доступу до об’єкту, проте не збільшують лічильник кількості посилань на нього, як проілюстровано в рисунку 2.

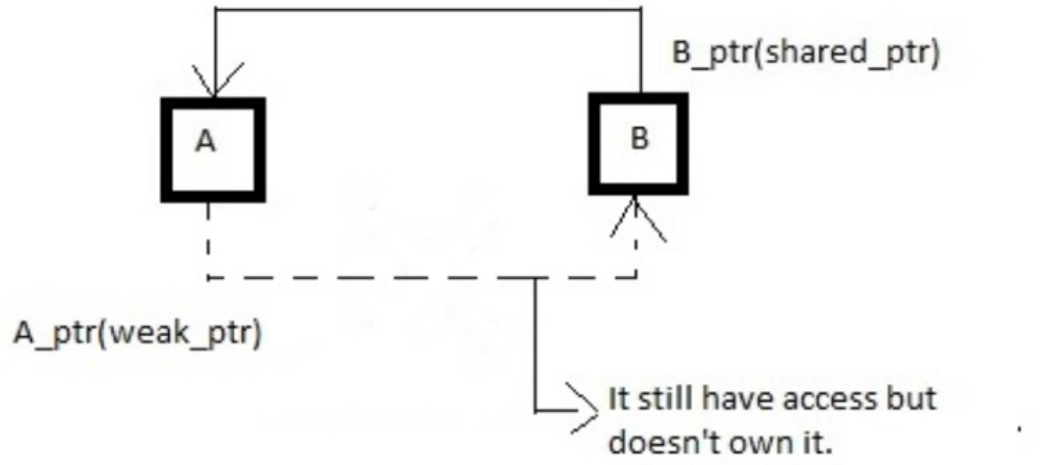


*Рисунок 2 - приклад розв'язання проблеми циклічної залежності указників [https://www.geeksforgeeks.org/auto_ptr-unique_ptr-shared_ptr-weak_ptr-2/]*

Таким чином при втраті зв’язку з об’єктом B, лічильник вказівників на нього буде рівний нулеві, після чого звільниться пам’ять, виділена під цей об’єкт, в результаті чого, пропаде зв’язок між A та B, й об’єкт A також видалиться слідом. В даному прикладі наведена ситуація є досить тривіальною, та легко вирішується методом сильних та слабких посилань. Проте, якщо розглянути варіант, де об’єктів є значно більше, а зв’язки між ними більш хаотичні, розробити правильну архітектуру сильних та слабких посилань стає значно складніше.

### *Boehm-Demers-Weiser Garbage Collector*

Наразі існує створений Гансом Боемом, та його колегами, збирач сміття, або англійською: Garbage Collector, який замінює виклики функції malloc()[8], realloc ()[9] на GC_MALLOC () та GC_REALLOC () відповідно, та відмову від виклику функції free()[10]. Даний збирач працює на мові C, а, отже, його використання можливе й в контексті мови C++, і, також, даний застосунок може використовуватися в режимі контролю та знаходження витоків пам’яті. Проте спосіб його роботи

наразі є застарілим, й не відповідає сучасним вимогам мови C++, де виділення об’єктів переважно відбувається через оператор new, а видалення через оператор delete, тобто з викликом деструктора. Таким чином, Garbage Collector, створений Гансом Боемом, не може замінити наявні реалізації Smart Pointers, та покращити їх, відповідно до описаних вимог.

Отже, розглянувши основні методи для роботи з пам’яттю в мові C++, ми спостерігаємо нестачу повністю автоматизованого способу контролю за створенням та звільненням об’єктів, який би трасував ситуації незвільнених ділянок пам’яті, взаємо зациклених посилань та був би при тому оптимальним та не затратним в часі виконання.

# Опис патерну, запропонованого Зінковичем Андрієм

### *Концепція*

Метод контролю над створенням об’єктів, запропонований у статті Зінковича Андрія вводить поняття опорних указників, тобто тих, які знаходяться в області пам’яті, до якої у користувача є прямий доступ, тобто коли об’єкт створюється в ділянці пам’яті під назвою стек[11], або ж в структурі даних, та внутрішніх указників, тобто тих, які знаходяться в тілі об’єктів. Таким чином, досяжність об’єкту визначається не лічильником посилань на нього, а існуванням шляху посилань від довільного опорного вказівника до даного об’єкту, шляхом дерева посилань.

### *Теоретична реалізація*

Перш за все, в своїй реалізації патерн вимагає при виділенні пам’яті для об’єкту створювати метаінформацію про цей об’єкт та відомість в реєстрі об’єктів про те, що відбулось виділення пам’яті. Таким чином, створюється централізований блок для управління об’єктами, де міститься інформація про всі об’єкти та посилання на них. Також вводиться два класи указників: RootPtr(див. Додаток А) та InnerPtr(див. Додаток А), які є реалізаціями опорних указників та внутрішніх відповідно. Дані класи наслідують абстрактний клас APtr(див. Додаток А), в якому зберігаються посилання на об’єкт, за яким закріплений указник, та посиланням на метаінформацію про нього. Клас RootPtr також наслідує клас InnerPtr та всі операції батьківського класу, за винятком того, що в класі RootPtr довизначена функція IsRoot()(див. Додаток А), яка повертає об’єкт логічного типу, таким чином, що при виклику цієї функції, об’єкт класу InnerPtr поверне значення false, тобто, що він не є опорним указником, а

об’єкт типу RootPtr поверне true, інформуючи, що даний указник є опорним. Створення об’єктів супроводжується перевизначеним оператором new(див. Додаток Б), який і бере на себе відповідальність за створення метаінформації про об’єкт, додає її в реєстр об’єктів та виділяє пам’ять для цього об’єкту. Також вводиться поняття збору сміття. Тобто процесу під час якого блок управління об’єктами визначає досяжність кожного об’єкту, та звільнює недосяжні. Отже, використання даного патерну виключає можливість ручного видалення об’єктів, на які посилаються указники, нащадки класу APtr, проте дозволяє додатково використання Smart Pointers, або ручного управління пам’яттю використовуючи стандартні оператори new та delete. Процес збору сміття введений раніше відбувається наступним чином:

Перш за все вводиться поняття мітки, певного значення, яке повідомляє чи до цього об’єкту можливо доступитися від одного з опорних указників.

1. Перший етап: блок управління пам’яттю змінює значення поточної мітки на інше(наприклад змінюючи логічне значення на протилежне).
2. Другий етап: блок управління пам’яттю в циклі проходиться по всіх указниках, та перевіряє, чи даний указник є опорним.
    - *Якщо указник опорний*: значення мітки в інформації про об’єкт, на який він посилається змінюється на поточне. Алгоритм рекурсивно шукає внутрішні указники об’єкту, шляхом аналізу адрес та адресної арифметики, міняє їх мітку на поточну, й знову рекурсивно викликає процес пошуку внутрішніх указників для даних вкладених об’єктів, і так далі.
    - *Якщо указник внутрішній*: алгоритм його пропускає та іде далі.

3. Третій етап: блок управління пам’яттю проходиться по всіх об’єктах, та порівнює їхню мітку в реєстрі записів про цей об’єкт з чинною
    - *Якщо мітки співпадають*: алгоритм іде далі, залишаючи цей об’єкт.
    - *Якщо мітки розходяться*: це означає, що до об’єкту неможливо доступитися від опорних указників, тому об’єкт визначається як недосяжний від основної програми й підлягає видаленню.
4. Четвертий етап: видалення всіх недосяжних об’єктів.

***Схематичне зображення алгоритму роботи патерну***

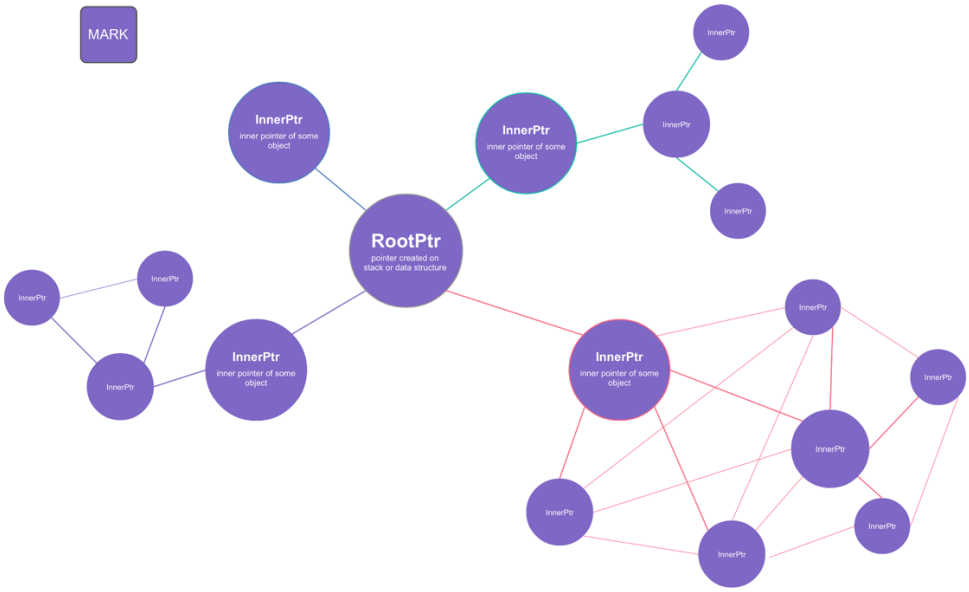


*Рисунок 3 - початковий стан програми. MARK - значення мітки, визначається кольором області.Кола - вказівники, лінії - зв'язки між ними. Напрямок посилань був упущений для спрощення рисунку [self]*

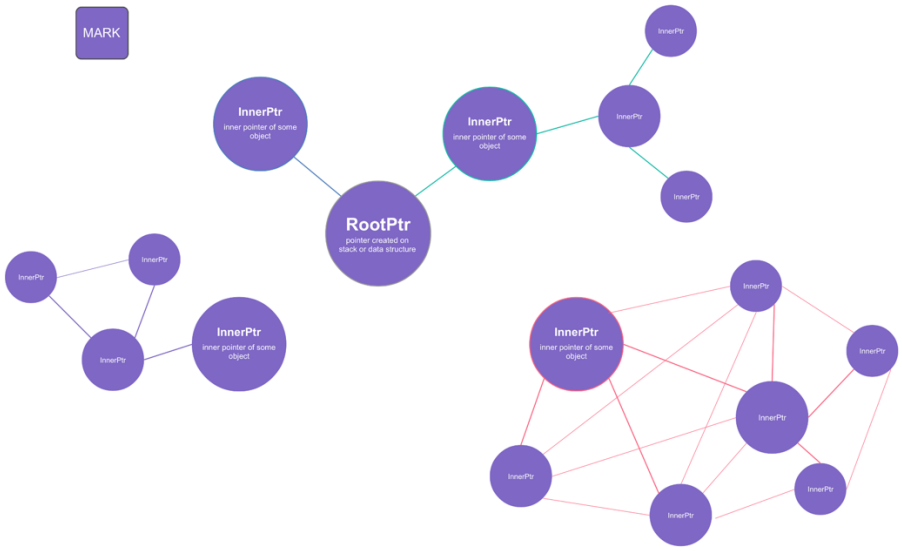


*Рисунок 4 - розрив зв'язку між об'єктом, який зберігається в RootPtr та його внутрішніми указниками InnerPtr [self]*

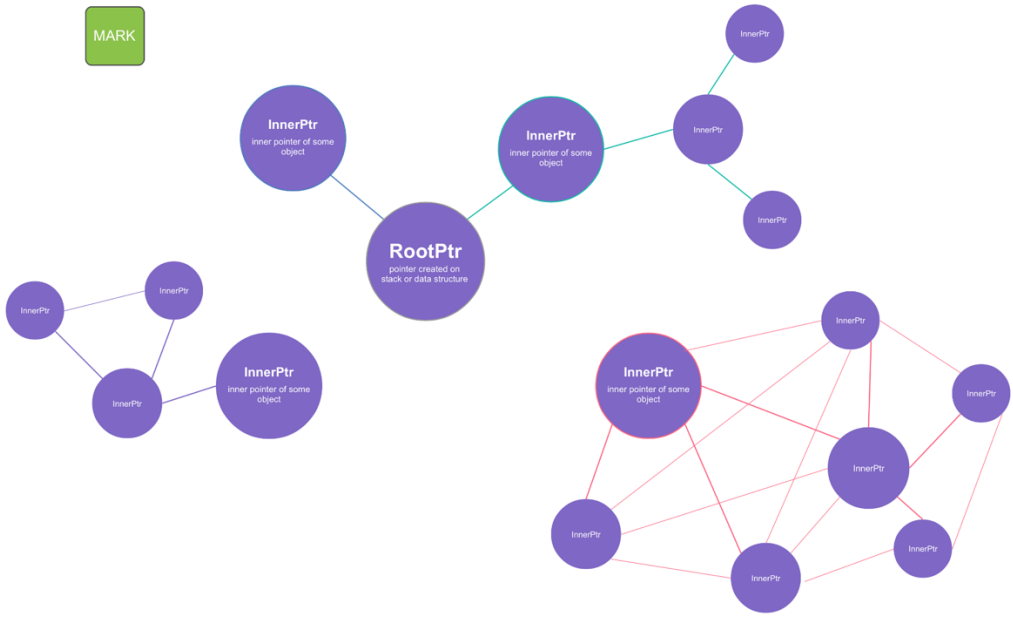


*Рисунок 5 - початок збору сміття. Зміна значення поточної мітки [self]*

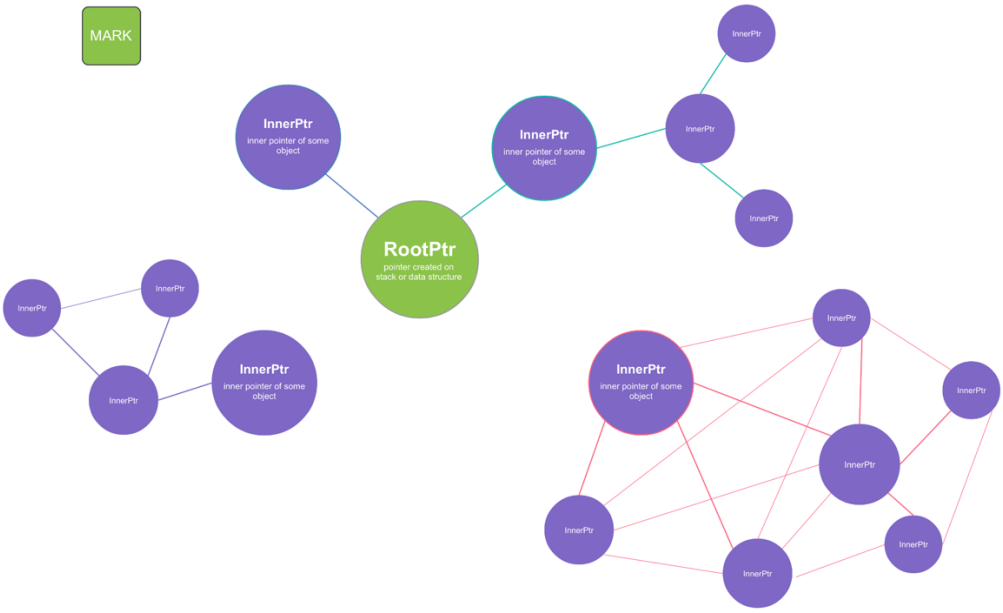


*Рисунок 6 - оновлення мітки всіх опорних указників [self]*

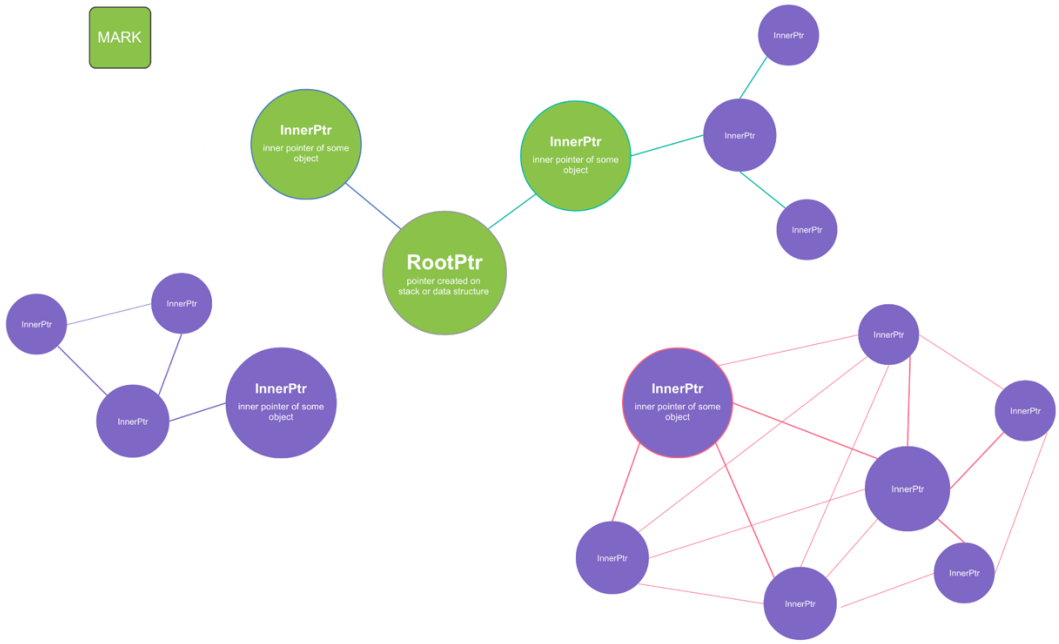


*Рисунок 7 - зміна значення мітки всіх об'єктів, досяжних від опорних указників [self]*

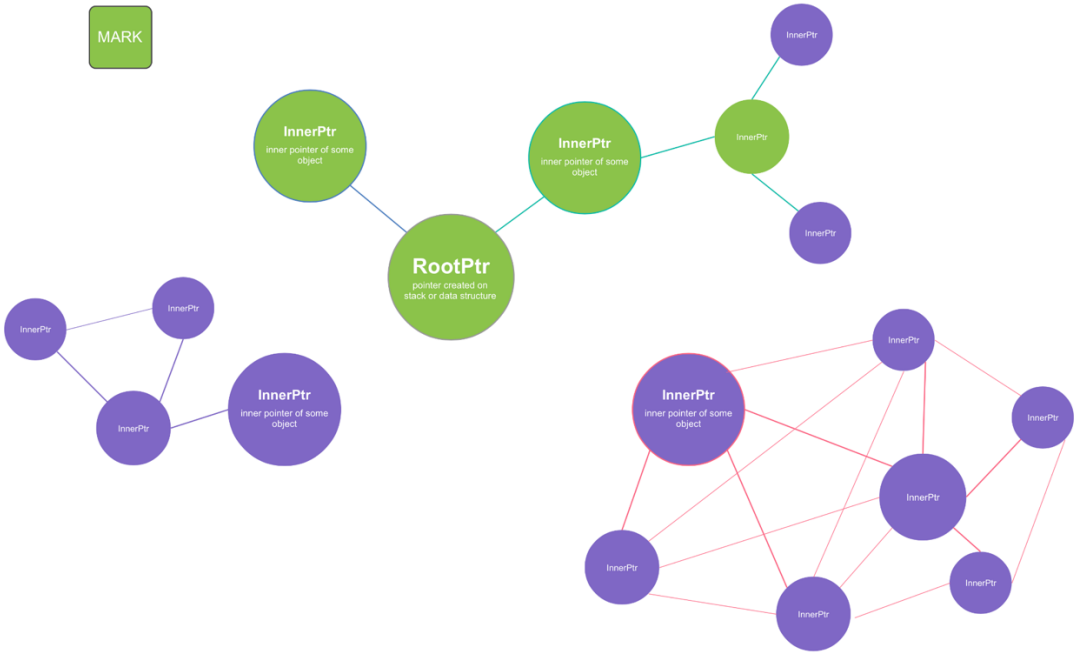


*Рисунок 8 - продовження процесу маркування об'єктів [self]*

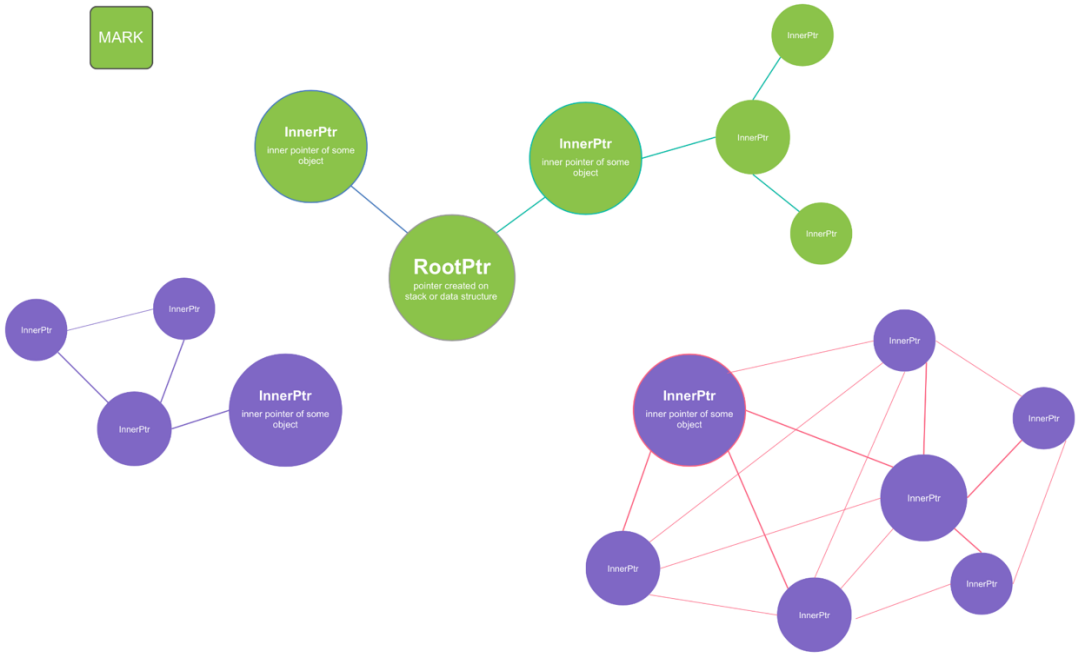


*Рисунок 9 - завершення процесу маркування об'єктів. Всі досяжні від опорних указників об'єкти позначені зеленим кольором [self]*

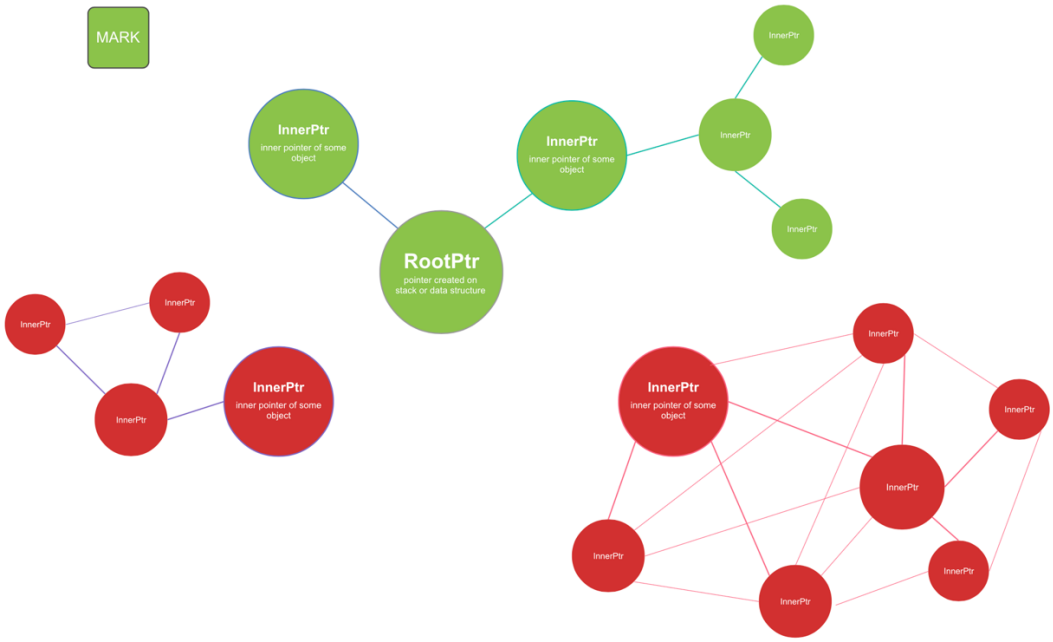


*Рисунок 10 - знаходження всіх недосяжних об'єктів, шляхом співставлення їх мітки та поточної. Об'єкти, зафарбовані червоним - підлягають видаленню [self]*

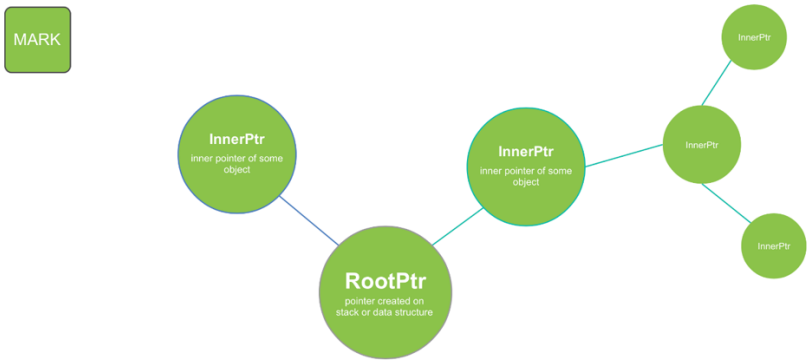


*Рисунок 11 - кінець роботи алгоритму, в пам'яті зберігаються лише досяжні об'єкти [self]*

### *Недоліки наявної реалізації патерну*

Основним недоліком реалізації, запропонованої в статті є погана оптимізація виконуваної програми, та її невідповідність принципам SOLID[12] і правилам Деметри[13]. Таким чином, запропонована мною реалізація наведеного патерну має на меті вирішити описані проблеми та покращити наведену реалізацію.

# Опис власної реалізації патерну

### *Вступ*

Запропонована мною реалізація патерну керування динамічним виділенням пам'яті, методом маркування об’єктів для перевірки досяжності об’єкту від опорних вказівників - взяла за основу деякі алгоритми та архітектурні рішення від методу, описаного в статті Зінковича Андрія, а також, тривіальні реалізації указників, та їх операторів. Проте, у моїй реалізації було впроваджено інакший спосіб створення та зберігання об’єктів в пам’яті, а саме: використання пулу статичної розмірності для послідовного запису та зберігання об’єктів у пам’яті. Таким чином, дана реалізація прискорює виділення пам’яті для об’єктів, шляхом повернення посилання на наступне вільне місце в пулі, на відміну від стандартного методу створення об’єктів, який знаходить в оперативній пам’яті вільну ділянку відповідного розміру.

### *Опис архітектури*

Основою роботи даної реалізації патерну управління пам’яттю є клас GarbageCollector(див. Додаток В), який і проводить диспетчеризацію виділення та звільнення пам’яті, збір сміття та контроль за життям об’єктів. В даному класі зберігаються:

- вектори з указниками на метаінформацію про кожен об’єкт, підконтрольний даному класу, тобто створений через перезавантажений оператор new
- Адреси всіх указників, які наслідують клас APtr
- Об’єкт типу Pool, який містить в собі зарезервований пул пам’яті (див Додаток Г)
- Розмір зарезервованої пам’яті

- Значення поточної мітки
- Кількість зайнятої об’єктами пам’яті

В даному класі реалізовані методи для контролю над розмірністю пулу, додаванням та видаленням указників, додавання нового указнику на метаінформацію про об’єкт до внутрішнього реєстру об’єктів, метод виділення адреси та резервування місця в пулі. Особливістю даного класу є приватний конструктор, який приймає параметром розмір початково виділеної ділянки пам’яті для пулу, та статичний об’єкт цього ж типу, для забезпечення примітивної реалізації патерну Singleton[14].
Наступним важливим елементом структури збирача сміття є безпосередньо класи розумних вказівників. Перш за все це створений абстрактний клас вказівника APtr, який містять в собі посилання на метаінформацію про об’єкт, який він в собі зберігає, та допоміжні функції для роботи з об’єктом. Шаблонний клас внутрішніх(InnerPtr) указників є нащадком класу APtr з додатково реалізованими тривіальними операторами указників, такі як зірочка(*)[15], стрілка(->)[15] та присвоєння(=), а клас опорних(RootPtr) вказівників є нащадком InnerPtr з довизначеною функцію isRoot(), яка у випадку опорного вказівника повертає значення true, сигналізуючи про те, що даний указник є опорним. У випадку виклику функції isRoot() в контексті внутрішнього указника, буде повернуто значення false, що повідомляє, що даний указник є внутрішнім. Об’єкти класу опорного та внутрішнього вказівника можливо створити кількома різними способами, передаючи в конструктор об’єкту даного класу інший об’єкт цього класу(копіювальний конструктор[16]), указник на об’єкт, посилання на який потрібно зберігати в цьому класі, або ж використовуючи порожній конструктор без параметрів. Об’єкт батьківського класу APtr створити неможливо, адже в ньому видалений деструктор[17], що повідомляє компілятору те, що даний клас є повністю абстрактним та не допускає створення екземпляру об’єкту даного типу.

Наступний елемент ієрархії – клас ObjectMetadata, який відповідальний за збереження посилання на об’єкт та метаінформації про цей об’єкт. Його ключовими полями є безпосередньо указник довільного типу на об’єкт, вектор в якому містяться розумні указники, які посилаються на цей об’єкт, розмір об’єкту в байтах, значення мітки для об’єкту та функтор destroy_, який містить в собі стратегію видалення об’єкту(див. Додаток Ґ). При видаленні екземпляру ObjectMetadata, наприклад, при стиранні запису про об’єкт із загального реєстру об’єктів, в його деструкторі викликається функтор destroy_, параметром якому передається посилання об’єкт, інформацію про який зберігає клас ObjectMetadata для явного виклику деструктора цього об’єкту.
Клас Pool містить в собі статичну ділянку пам’яті, в якій об’єкти зберігаються один за одним. Даний клас складається з внутрішньої структури PoolMem(див. Додаток Г), яка складається з указника на виділену ділянку пам’яті, поля, в якому зберігається розмірність даної ділянки та лічильника, який зберігає кількість зайнятої пам’яті в цій ділянці починаючи від початку. Реалізована дана структура з використанням патерну Pimple[18]. У зовнішньому класі Pool міститься посилання на об’єкт даної структури. В класі Pool реалізовані функції для отримання адреси початку пулу, отримання розмірності пулу та отримання указника на наступну вільну ділянку пам’яті пулу. Створити об’єкт даного класу можливо через конструктор, передавши тому параметром число – розмірність пулу, який потрібно створити. Також в класі Pool реалізований оператор присвоєння-переміщенням[19].

Додатково в проекті наявні перевизначений оператор створення нового об’єкту new. Даний оператор приймає аргументами два числа, перший: розмірність – підставляється автоматично при виклику даного оператора[20]; другий: ціле число, яке визначає стратегію створення об’єкту. Виклик оператора new для створення нового об’єкту виглядає

наступним чином: new(0) A, де 0 – число, передане параметром. Дана архітектура зумовлена також тим, що перевизначення оператору new з двома аргументами дозволяє користувачеві в програмі використовувати стандартний оператор new від одного аргументу для створення об’єктів. Наразі реалізована лише одна стратегія створення об’єктів, тому передане число не впливає на перебіг виконання програми, проте теоретично можливі способи прискорення часу створення екземплярів об’єктів будуть наведені в розділі ‘Теоретичні напрями покращення реалізації’.

***Опис алгоритму роботи програми***

Першим чином вводиться наступне правило: всі розумні вказівники, створені користувачем на стеці, або в структурі даних повинні бути опорними, а всі указники, які знаходяться в тілі довільного об’єкту повинні бути внутрішніми. Створення об’єкту та присвоєння його адреси указнику продемонстровано в наступному прикладі:

**RootPtr pa = new(0) A;** - для опорних указників.

**InnerPtr pa = new(0) A;** - для внутрішніх указників.

Хоч клас RootPtr та InnerPtr є шаблонними, використовуючи можливість стандарту C++17 Class template argument deduction (CTAD)[23]:

```
template <typename T>
InnerPtr(const InnerPtr<T>& other)->InnerPtr<T>;
template <typename T>
RootPtr(const RootPtr<T>& other)->RootPtr<T>;

template <typename T>
InnerPtr(T* arg)->InnerPtr<T>;
template <typename T>
RootPtr(T* arg)->RootPtr<T>;
```

користувачеві не обов’язково вказувати тип указника при створенні, адже тип шаблону визначатиметься з параметра в конструкторі. Єдиним виключенням є створення указника використовуючи порожній конструктор без параметрів. В такому випадку потрібно обов’язково явно задавати тип указника.

При виклику оператору new, в тілі оператору відбувається звернення до статичного об’єкту manager, класу GarbageCollector, та виклик методу getNextFreeAddress, який повертає указник довільного типу на наступну вільну ділянку пам’яті в пулі, куди й буде поміщено об’єкт. Після цього створюється новий екземпляр класу ObjectMetadata, в конструктор якого параметрами передаються указник на ділянку пам’яті, куди був записаний щойно створений об’єкт, та його розмір. Наступним кроком буде звернення до об’єкту manager з викликом його функції addNewObject, передавши методу параметром указник на створений об’єкт типу ObjectMetadata, аби додати інформацію про об’єкт до загального реєстру об’єктів. Завершується робота оператора поверненням указника довільного типу, який вказує на місцезнаходження об’єкту в пулі. Далі повернутий оператором new указник неявно перетворюється до типу об’єкту який ми створюємо.

Метод збору сміття програмісту потрібно викликати явно, звернувшись до статичного об’єкту manager, класу GarbageCollector, коли в цьому є потреба. При спробі створити об’єкт тоді, коли досягнуто максимальний розмір пулу виклик методу CollectGarbage() відбудеться автоматично, а розмір пулу збільшиться вдвічі з урахуванням розмірності створюваного об’єкту. Сам метод збору сміття працює в шість етапів:

- Видалення інформації про об’єкти, на які ніхто не посилається.
- Маркування досяжних об’єктів.
- Видалення інформації про недосяжні об’єкти.
- Створення нового пулу.

- Перенесення об’єктів з старого пулу до нового та зміна посилань на них(об’єкти при перенесені розташовуються послідовно).
- Присвоєння полю pool класу GarbageCollector адресу новоствореного пулу та звільнення пам’яті, зайнятої старим пулом.

Перевірка на те, чи на об’єкт хтось посилається відбувається шляхом звернення до реєстру об’єктів з метою отримання метаінформації про цей об’єкт та перевіркою розмірності вектору pointers, який зберігає указники, які посилаються на даний об’єкт. Якщо розмірність цього вектору рівна нулеві – це означає, що на даний об’єкт не посилається жоден з указників й інформацію про нього можливо видалити.

Наступний крок – маркування об’єктів відбувається згідно з алгоритмом наведеним у статті Зінковича Андрія, описаним раніше в розділі ‘Опис патерну, запропонованого Зінковичем Андрієм’. Критерієм недосяжності об’єкту є відмінність значення його мітки в реєстрі об’єктів від поточної. Таким чином алгоритм проходить по записам всіх об’єктів та зіставляє значення поточної мітки з тією, яка зберігається в записі про об’єкт. Якщо мітки не співпадають – об’єкт вважається недосяжним, а інформація про нього видаляється з реєстру об’єктів. Перенесення об’єктів в новий пул відбувається за наступним алгоритмом:

- Всі досяжні об’єкти побітово копіюються в новий пул послідовно один за одним, а в реєстрі об’єктів змінюється значення адреси цього об’єкта на нове. Також пара значень старої адреси та указника на оновлену інформацію про об’єкт додаються до вектору transferedObjects, який буде використовуватися в наступному кроці алгоритму для зміни адрес внутрішніх указників даного об’єкту.
- Алгоритм перебирає всі значення з вектору transferedObjects, та перевіряє чи не існує внутрішніх вказівників, які знаходилися б в тілі поточного об’єкту. Якщо ж такий існує, нова адреса визначається шляхом бітової арифметики, та значення адреси

указника в реєстрі міняється на оновлену. Також оновлюються дані про адресу указника в метаінформації про об’єкт, на який цей указник посилається.

### *Схематичне зображення роботи методу GarbageCollector::CollectGarbage()*

Наступні діаграми демонструють роботу методу GarbageCollector::CollectGarbage(), якого викликали після виконання наступного коду:

```
RootPtr pa = new(0) A;
    pa->pb = new(0) B;
    pa->pc = new(0) C;
    pa->pc->pd = new(0) D;
// створення циклічної залежності між C та D
    pa->pc->pd->pc = pa->pc;
```

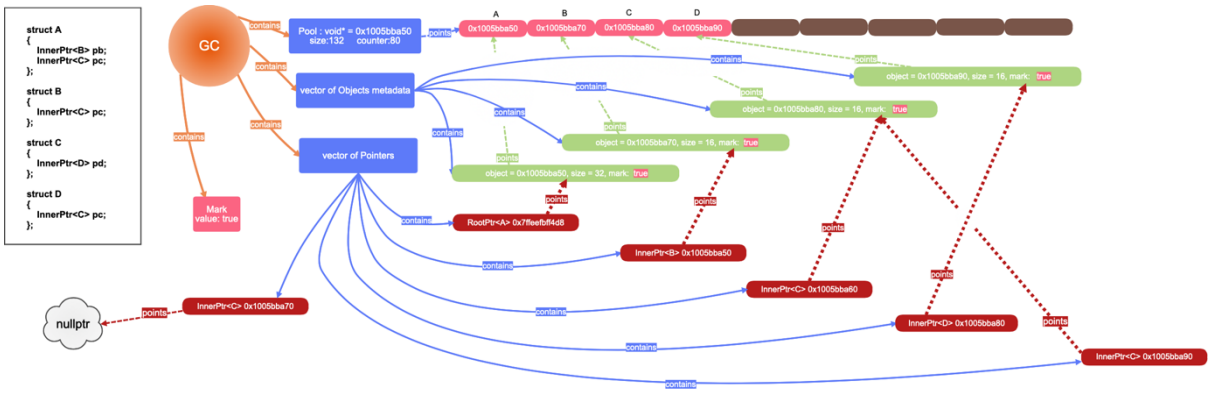


*Рисунок 12 - Початковий стан ієрархії. В схемі присутні спрощення деяких вузлів та виділення основної інформації для покращення розуміння роботи застосунку [self]*

Реєстр об'єктів(GC) містить в собі значення поточної мітки(true), пул пам'яті, де зберігаються об'єкти(Pool), вектор з посиланнями на метаінформацію про об'єкти та вектор з посиланнями на розумні вказівники. В записах про об’єкт (на схемі – зелені області) зберігаються адреса об’єкту, кількість зайнятої ним пам’яті та значення поточної мітки

цього об’єкту. Пул об’єктів Pool містить в собі посилання на початок пулу, його розмірність та кількість зайнятої пам’яті. В розумних указниках зберігається їх тип(опорні/внутрішні), адреса цього указника(this) та посилання на запис про об’єкт в реєстрі. Всі об’єкти розташовані в пулі один за одним.

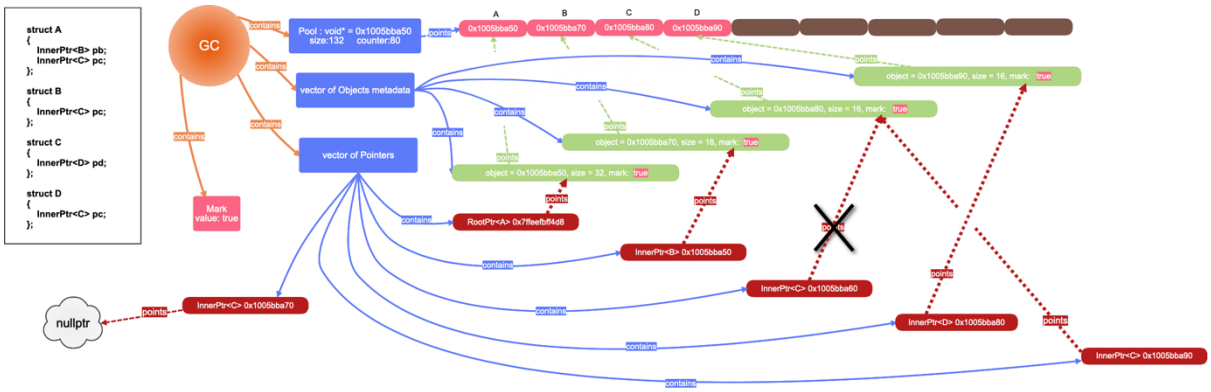


*Рисунок 13 - прибираємо зв'язок між об'єктами B та C [self]*

Продемонструємо процес втрати зв’язку між об’єктами B та C, шляхом виклику наступного коду:

**pa->pc = nullptr;**

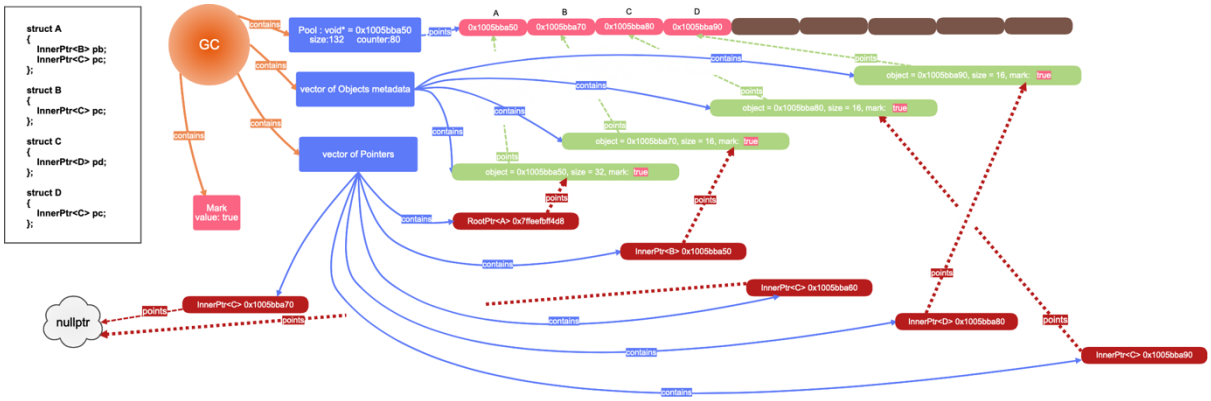


*Рисунок 14 - до об'єктів C та D неможливо доступитися від опорного указника типу A, проте між ними зберігається циклічна залежність [self]*

До об’єктів C та D тепер неможливо доступитися від основної програми, тому потрібно звільнити зайняту ними пам’ять шляхом виклику методу GarbageCollector::CollectGarbage() статичного об'єкту GarbageCollector::manager, прописавши в програмі наступний код:

**GarbageCollector::manager.CollectGarbage();**

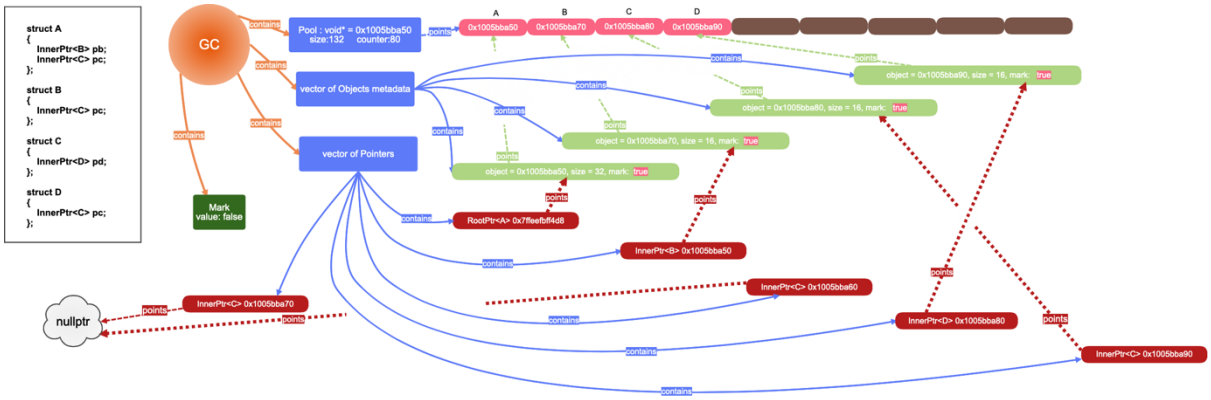


*Рисунок 15 - перший етап збору сміття. Зміна значення поточної мітки(true) на протилежне(false) [self]*

Перш за все, алгоритм змінює значення поточної мітки на протилежне. Для наочності, на схемі відбулась також зміна кольору поля мітки.

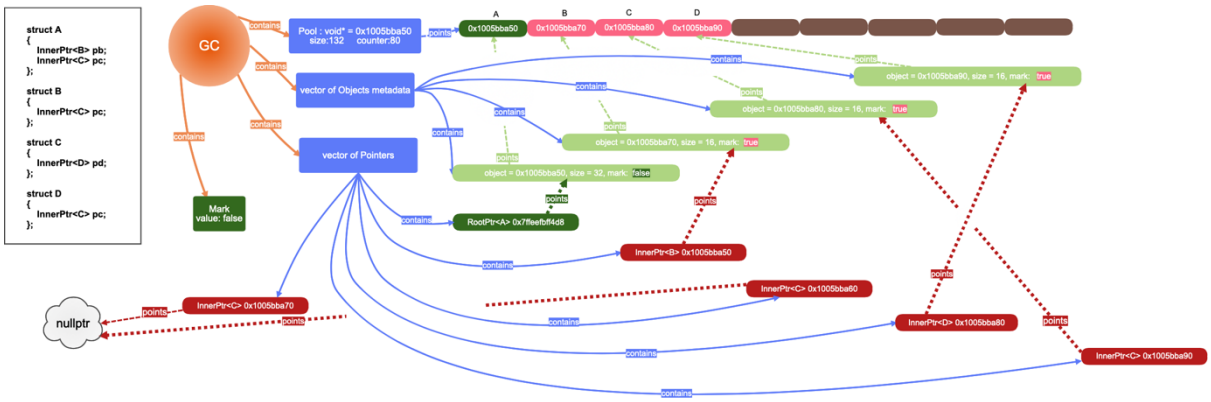


*Рисунок 16 – другий етап збору сміття. Маркування всіх опорних указників(RootPtr) [self]*

Наступним кроком роботи процесу збору сміття є знаходження всіх опорний указників, зміна їхньої мітки на поточну та запуск рекурсивного пошуку вкладених посилань, методом адресної арифметики.

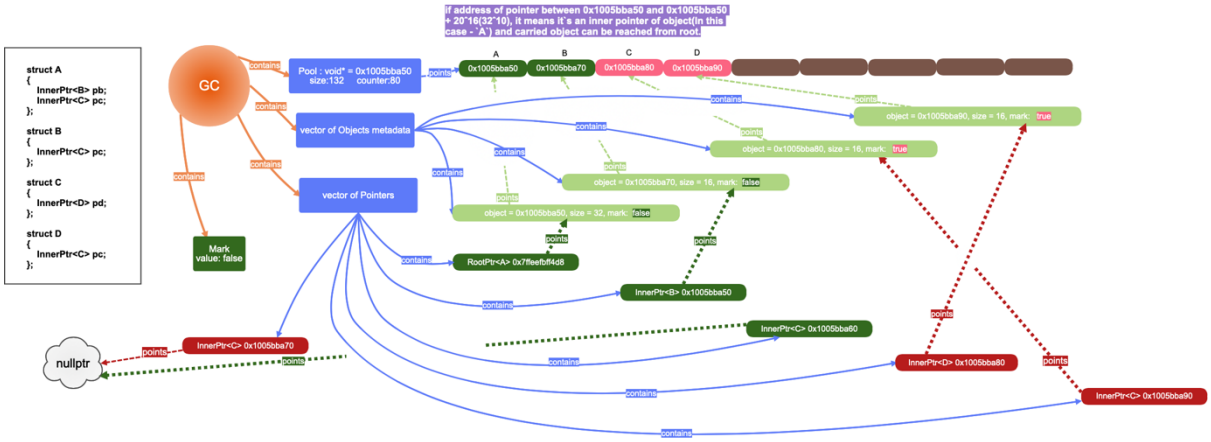


*Рисунок 17 – третій етап збору сміття. Рекурсивний пошук вкладених посилань об'єктів, які зберігаються в опорних указниках та маркування досяжних об'єктів [self]*

Алгоритм обраховує праву(початкову) та ліву(кінцеву) адреси об’єкту. Наступним кроком він порівнює безпосередні адреси(this) всіх вказівників на те, чи не належать вони цим межам. У випадку відповідності даній умові, очевидно, що указник знаходиться в тілі даного об’єкту, тому до нього можливо доступитися. Далі алгоритм знаходить межі об’єкту, на який посилається цей, вкладений, указник й знову запускає процес перевірки вказівників на вкладеність.

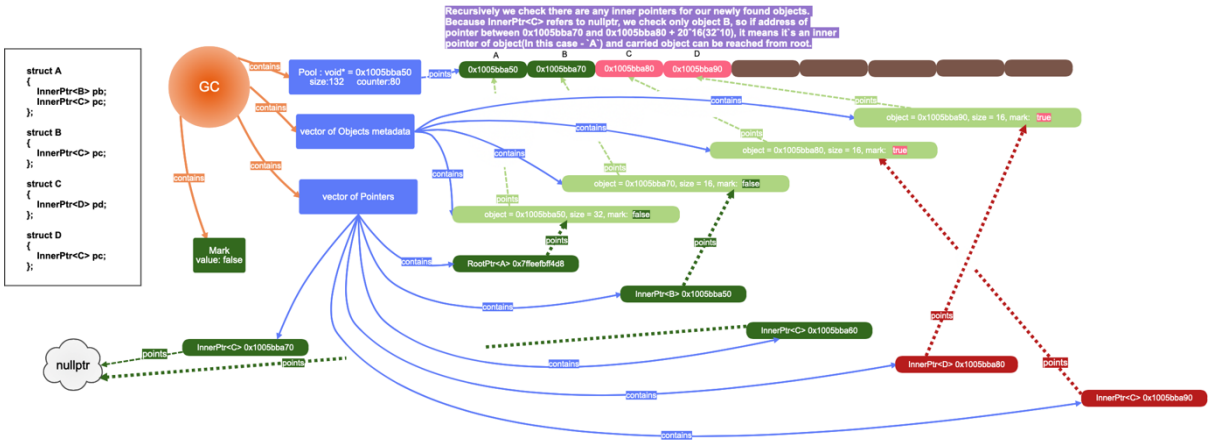


*Рисунок 18 - завершення процесу перевірки досяжності об'єктів. Всі об'єкти, до яких можливо доступитися від опорних указників мають в реєстрі мітку, яка збігається з поточною [self]*

Таким чином, в кінці етапу маркування у всіх об’єктів, досяжних від опорних указників, збігається значення мітки з поточною(також, на рисунку 18 ці об’єкти залиті тим же кольором, як і колір поля мітки).

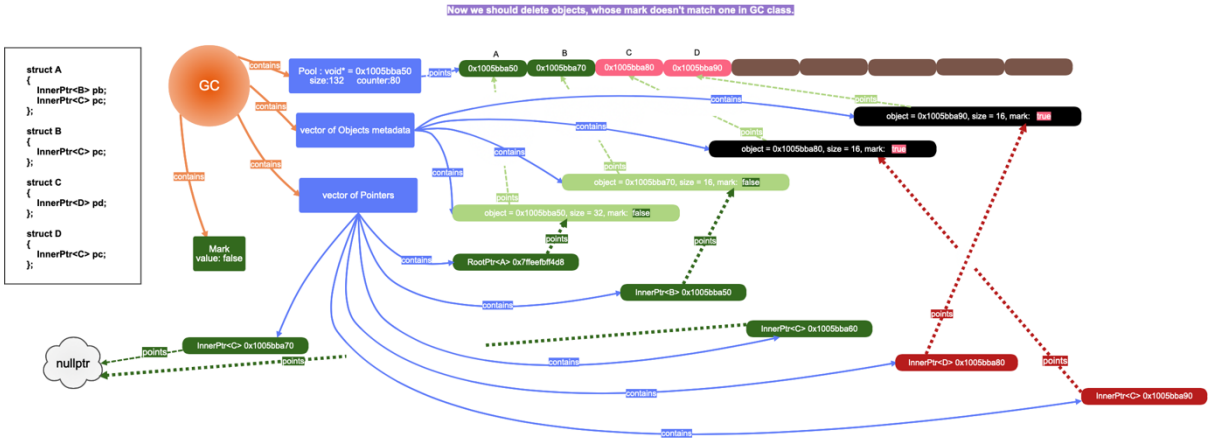


*Рисунок 19 – четвертий етап збору сміття. Всі об'єкти, значення мітки яких розходяться з поточною вважаються недосяжними та підлягають видаленню [self]*

У тих же об'єктів, до яких неможливо доступитися від опорних указників – значення мітки залишилося попереднім, тому перебравши в циклі всі записи в реєстрі можна поділити об'єкти на ті, які переходять далі та копіюються в нову ділянку пам'яті, та на ті, які підлягають видаленню.

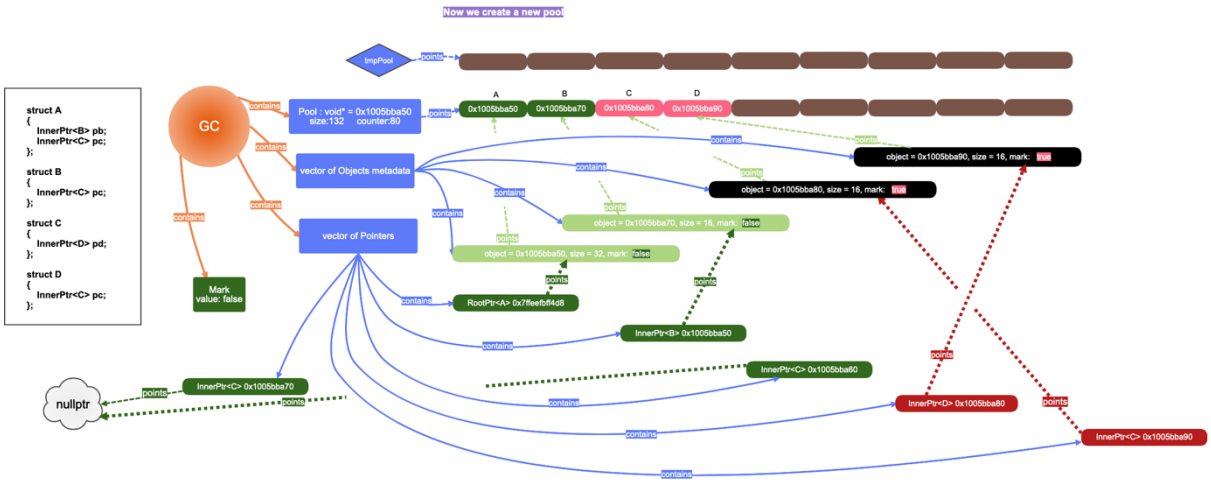


*Рисунок 20 - п'ятий етап процесу збору сміття. Створення нового пулу, стирання інформації про недосяжні об'єкти з реєстру та виклик їх деструкторів [self]*

На наступному кроці роботи алгоритму створюється новий пул, куди будуть в подальшому перенесені лише ті об'єкти, які пережили епоху збору сміття. Також з реєстру прибираються записи про недосяжні об'єкти, а при видаленні метаданих про об'єкт, примусово викликається деструктор того об'єкту, дані про які видаляються. Також, якщо об'єкт

містив в собі вкладені указники, вони також прибираються з реєстру автоматично.

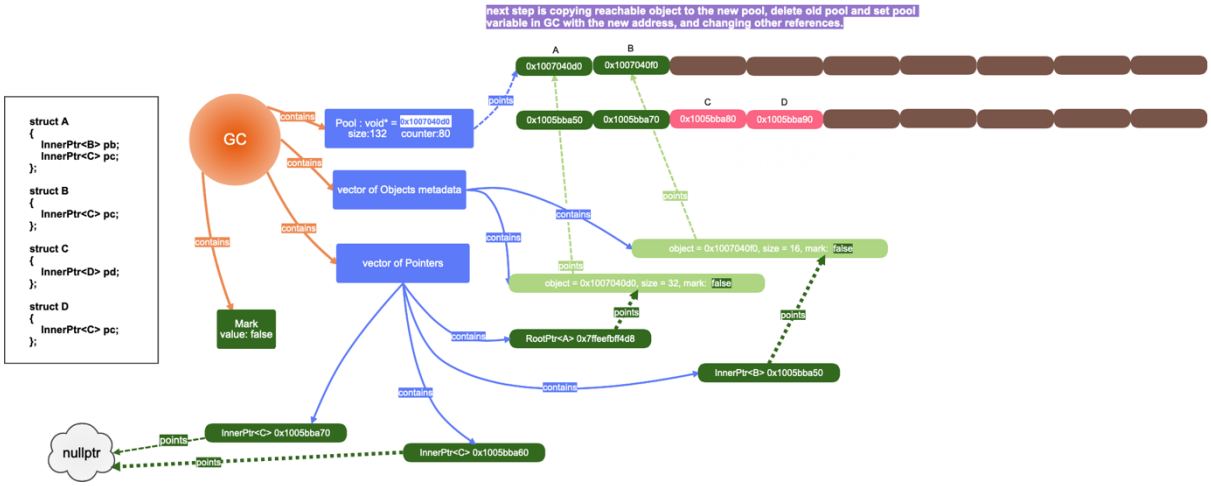


*Рисунок 21 - шостий етап процесу збору сміття. Перенесення досяжних об'єктів до нового пулу, та присвоєння об'єкту Pool в реєстрі нового значення посилання [self]*

Після того, як в реєстрі залишаться лише досяжні об'єкти, алгоритм побітово скопіює інформацію, яку містить кожен об'єкт з старого пулу в новий, розташувавши об'єкти один за одним. Також, на цьому етапі відбувається оновлення всіх посилань в програмі.

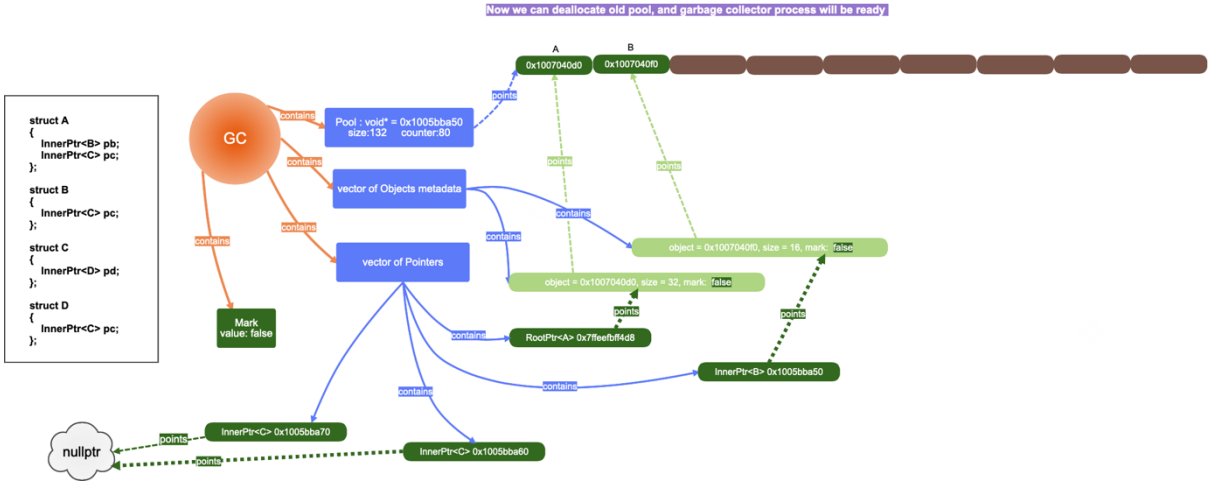


*Рисунок 22 - кінець роботи алгоритму. Фінальний стан програми [self]*

Останнім кроком буде звільнення пам'яті, яку займає старий пул та продовження роботи програми.

## Теоретичні напрями покращення реалізації

Алгоритм виділення пам’яті в наявній реалізації є досить неоптимальним рішенням, адже при створенні об’єкту відбувається процес пошуку в невідсортованому масиві, адже оператор new повертає посилання на новостворений об’єкт, а оскільки в класі розумного указника повинне зберігається посилання на метаінформацію про даний об’єкт, то при кожному виклику оператора new, в конструкторі, або ж операторі присвоєння розумних вказівників викликається пошук метаданих в реєстрі за значенням адреси пам’яті новоствореного об’єкту. Таким чином це приводить до складності алгоритму виділення пам’яті O(n), через що дана реалізація значно поступається стандартному оператору new у швидкості виділення пам’яттю. Значною мірою можна пришвидшити наявну реалізацію, якщо позбутися у процесі створення об’єкту етапу пошуку в масиві. Таким чином, я створив демонстративну реалізацію патерну, де при ініціалізації указника, або ж присвоєння йому посилання на об’єкт, замість пошуку метаданих в реєстрі в якості метаінформації про цей об’єкт береться останній запис в масиві об’єктів, припускаючи, що кожен новостворений об’єкт буде одразу ж присвоюватися указнику. Таким чином ми досягли складності виділення пам’яті O(1), чим в рази прискорили швидкість створення об’єктів. Також теоретично можна значно прискорити алокацію пам’яті, якщо не створювати об’єкти, які зберігають метадані через стандартний оператор new, а завчасно створити вектор з n порожніх об’єктів та при виділенні пам’яті збільшувати лічильник на один і записувати метадані в наступний елемент вектору.

## Виняткові ситуації

Наразі дана реалізація не може працювати з масивами та іншими структурами даних внутрішніх указників, адже перевірка на вкладеність відбувається шляхом бітової арифметики. Проте, якщо зберігати в структурі даних опорні указники - програма працює в звичайному режимі. При нормальній роботі програми, та не повинна перериватися аварійно, а єдиною винятковою ситуацією є намагання створити об’єкт такий, що після операції прибирання сміття, розмір об’єкту, який користувач намагається створити, та розмірність зайнятої пам’яті у пулі – більша за розмір максимально доступної ділянки пам’яті, яку можна отримати викликаючи команду malloc()[21]. В такому випадку відбудеться процес викиду помилки std::out_of_range[22].

# Порівняння швидкодії різних способів створення та видалення об’єктів

За основу порівняння візьмемо стандартний оператор new, нашу імплементацію патерну контролю за виділенням та звільненням пам’яті та експериментальну оптимізовану реалізацію, описану в розділі ‘ Теоретичні напрями покращення реалізації’.

100 objects

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| New | 39mcs | 39mcs | 40mcs | 38mcs | 39mcs | 41mcs | 38mcs | 38mcs | 40mcs |
| Fast GC | 183mcs | 176mcs | 166mcs | 156mcs | 157mcs | 157mcs | 158mcs | 157mcs | 156mc |
| Slow GC | 701mcs | 754mcs | 683mcs | 1247mcs | 798mcs | 888mcs | 695mcs | 991mcs | 710mc |

Таблиця 1

10000 objects

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Fast GC | 17.792ms | 15.771ms | 15.212ms | 14.862ms | 15.516ms | 15.323ms | 15.587ms | 15.584ms | 15.266ms |
| Slow GC | 5150.2ms | 4843.76ms | 5240.59ms | 5372.95ms | 5200.19ms | 5031.19ms | 4896.14ms | 4936.43ms | 4893.23ms |
| New | 2.45ms | 3.029ms | 2.734ms | 3.239ms | 2.541ms | 2.536ms | 2.528ms | 2.499ms | 2.624ms |

Таблиця 2

100000 objects

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Fast GC | 178.53ms | 163.165ms | 150.715ms | 160.714ms | 157.102ms | 161.91ms | 161.109ms | 161.392ms | 160.593ms |
| New | 25.361ms | 25.863ms | 26.655ms | 25.002ms | 25.719ms | 26.249ms | 25.221ms | 24.858ms | 26.886ms |

Таблиця 3

Згідно з даними таблиць 1, 2 та 3 можна стверджувати, що припущення про методи прискорення виділення пам’яті є істинним.

# Висновок

Наразі запропонована мною модель є повністю робочим продуктом, та може бути використана в проектах, де вимагається автоматизований контроль над створенням та видаленням об’єктів. Попри те, що вона поступається в швидкості стандартним методам виділення пам’яті, наявна реалізація є значно швидшою за ту, яка наведена в першоджерелі, а також, існує продемонстрована реалізація патерну така, що лише в кілька разів поступається в швидкості стандартним способам алокації, а також теоретично описана така, яка може бути швидшою за стандартні оператори алокації в залежності від ступеня сегментації пам’яті. Також, запропоновані методи реалізації не є кінцевими, а лише наводять одну з можливих імплементацій патерну. Таким чином, допускається можливість існування більш оптимальних реалізацій. Також, даний патерн є досить гнучким та простим в розумінні, тому програміст з легкістю може змінити архітектуру так, аби пристосувати створення об’єктів, або процес збору сміття, залежно від потреб його застосунку.

# Список джерел


1 - `What is a memory leak in C++, and provide an example?`
https://www.programmerinterview.com/c-cplusplus/what-is-a-memory-leak-in-c/

2 – ‘Блуждающие, дикие или зависшие указатели’, Джесс Ліберті
http://programming-lang.com/ru/comp_programming/liberti/0/j411.html

3 - `Garbage Collector & C++`, Зенкович Андрей, 26 квітня 2016
https://habr.com/ru/post/282544/

4 - `new and delete operators`, © Microsoft 2021
https://docs.microsoft.com/en-us/cpp/cpp/new-and-delete-operators?view=msvc-160

5 – `Smart pointers (Modern C++)`, 11/19/2019, © Microsoft 2021
https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160

6 – `Introduction to the Standard Template Library`, Copyright © 1999 Silicon Graphics, Inc.
https://justinmeiners.github.io/sgi-stl-docs/stl_introduction.html

7 - ` Shared pointer`, © cplusplus.com, 2000-2020
https://www.cplusplus.com/reference/memory/shared_ptr/

8 - `Allocate memory block`, © cplusplus.com, 2000-2020
http://www.cplusplus.com/reference/cstdlib/malloc/

9 - ` Reallocate memory block`, © cplusplus.com, 2000-2020
https://www.cplusplus.com/reference/cstdlib/realloc/

10 - ` Deallocate memory block`, © cplusplus.com, 2000-2020
https://www.cplusplus.com/reference/cstdlib/free/

11 - `Stack vs Heap Memory Allocation`, @Ankit_Bisht, 27 Apr, 2021
https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

12 - ` SOLID: The First 5 Principles of Object Oriented Design`, Samuel Oloruntoba, Validated on December 17, 2020
https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

13 - `Прекрасный Закон Деметры`, травень 29, 2017, © Copyright 2021 CoderLessons.com
https://coderlessons.com/articles/devops-articles/prekrasnyi-zakon-demetry

14 - `DESIGN PATTERNS - SINGLETON PATTERN`, Copyright © 2020, bogotobogo
https://www.bogotobogo.com/DesignPatterns/singleton.php

15 - `C++ Pointer Operators`, © Copyright 2021 tutorialspoint
https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm
16 - `Copy constructors`, 26 December 2020
https://en.cppreference.com/w/cpp/language/copy_constructor

17 - ` Destructors `, 1 October 2020
https://en.cppreference.com/w/cpp/language/destructor

18 - ` PImpl `, 2 October 2020
https://en.cppreference.com/w/cpp/language/pimpl

19 - ` Move Constructors and Move Assignment Operators (C++) `, 03/05/2018, © Microsoft 2021
https://docs.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp?view=msvc-160

20 - `new Operator (C++)`, 11/04/2016, © Microsoft 2021
https://docs.microsoft.com/en-us/cpp/cpp/new-operator-cpp?view=msvc-160

21 - `malloc() — Reserve Storage Block`, © Copyright IBM Corporation 2002, 2015
https://www.ibm.com/docs/en/i/7.3?topic=functions-malloc-reserve-storage-block

22 - `Out-of-range exception`, © cplusplus.com, 2000-2020
https://www.cplusplus.com/reference/stdexcept/out_of_range/

23 – 'Class template argument deduction (CTAD) (since C++17)', 27 April 2021
https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

# Додаток А

```
template<typename T>
class InnerPtr : public APtr
{
public:
   InnerPtr();
   InnerPtr(T* object);
   InnerPtr(const InnerPtr<T>& other);
   ~InnerPtr();
   T* Get() const;
   bool IsValid() const;
   bool IsRoot() const;
   operator bool();
   operator T* ();
   T* operator->();
   T& operator*();
   const T& operator*() const;
   InnerPtr<T>& operator=(const InnerPtr<T>& other);
   InnerPtr<T>& operator=(T* other);
};

template<typename T>
class RootPtr : public InnerPtr<T>
{
public:
   RootPtr();
   RootPtr(T* object);
   RootPtr(const InnerPtr<T>& other);
   bool IsRoot() const;
   operator bool();
   operator T* ();
   T* operator->();
   T& operator*();
   const T& operator*() const;
   RootPtr<T>& operator=(const InnerPtr<T>& other);
   RootPtr<T>& operator=(T* other);
};
```

# Додаток Б

```
void* operator new(size_t, int);

void* operator new(size_t size, int strategy)
{
   void* res = GarbageCollector::manager.getNextFreeAddress(size);
   ObjectMetadata* objInfo = new ObjectMetadata(res, size);
   GarbageCollector::manager.addNewObject(objInfo);
   return res;
}
```

# Додаток В

```
class GarbageCollector
{
private:
   size_t              poolSize;
   Pool                pool;
   mutable std::vector<ObjectMetadata*> objects;
   std::vector<APtr*>          pointers;
   size_t                  allocatedBytes;
   bool                    currentMark;
   void MarkInnerObjects(ObjectMetadata* info);
   void MarkObjects();
   void deleteUnattainableObjects();
   void transferObjectsToTheNewPool(Pool&);
   std::vector<ObjectMetadata*>& getObjects();
   std::vector<APtr*>& getPointers();
   void deleteInvanidObjects();
   GarbageCollector(size_t poolSize);
   ~GarbageCollector();
   void addAllocatedBytes(size_t);
   void minusAllocatedBytes(size_t);
   void copyobjects(Pool&, std::vector<std::pair<ObjectMetadata*, const void* const>>&);
   void changeReferences(ObjectMetadata*);
   void transferSinglePointer(APtr**, size_t, ObjectMetadata*);
   void changePointers(std::vector<std::pair<ObjectMetadata*, const void* const>>&);
public:
   ObjectMetadata* getLatestInfo(){return objects.back();}
   static GarbageCollector      manager;
   void setPoolSize(size_t);
   void CollectGarbage();
   size_t getAllocatedBytes();
   bool getCurrentMark();
   void* getNextFreeAddress(size_t);
   void addNewObject(ObjectMetadata*);
   void addPointer(APtr*);
   ObjectMetadata* getObjectInfo(void*);
   void removePointer(APtr*);
};
```

# Додаток Г

```
class Pool
{
private:
   struct PoolMem;
   PoolMem* pool;
public:
   Pool(size_t size_);
   ~Pool();
   void* allocate(size_t size);
   void* getPool();
   size_t getPoolSize();
   const Pool& operator =(Pool&&);
};

struct Pool::PoolMem
{
   void* pool;
   size_t counter;
   size_t size;
};
```

# Додаток Ґ

```
template<typename T>
struct DestroyPolicy
{
   static void Destroy(void* obj)
   {
      (*(T*)(obj)).~T();
   }
};

class ObjectMetadata
{
private:
   mutable void*       object;
   size_t              size_;
   mutable bool        mark_;
   mutable std::vector<APtr*> pointers;
   void(*destroy_)(void*) = nullptr;
public:
   ObjectMetadata(void*,size_t);
   ~ObjectMetadata();
   const void* const getObjectsAddress() const;
   void setNewAddressForObject(void*) const;
   size_t getSizeOfAllocatedMemory() const;
   bool mark() const;
   void mark(bool) const;
   std::vector<APtr*>& getPointersThatReferToThisObject();
   size_t getReferenceCounter();
   void addPointerThatReferToThisObject(APtr*);
   void removePointerThatReferToThisObject(APtr*);
   bool hasDestroyPolicy();
   void setDestroyPolicy(void (*newPolicy)(void*));
};
```