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

Особливості управління пам'яттю з розвитком версій C++

Текстова частина до курсової роботи за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
доцент Кафедри мультимедійних систем
Бублик В. В.
_____
(підпис)
“____” _____ 2022 р.
Виконав студент
Крейдун А. М.
“____” _____ 2022 р.

Київ 2022

Міністерствро освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
доцент Кафедри мультимедійних систем
Бублик В. В.

_____ *(підпис)*

«____» _____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу
студенту Крейдуну Андрію Миколайовичу факультету інформатики 4-го курсу
ТЕМА Особливості управління пам'яттю з розвитком версій C++

Зміст ТЧ до курсової роботи:
1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Memory resource і allocator
6. Методи динамічного управління пам’яттю
7. Розумні вказівники
8. Алокатори
9. Практична частина
10. Висновки
11. Список використаних джерел

Дата видачі “___” _____ 2022 р. Керівник Бублик В. В. (підпис)
Завдання отримав Крейдун Андрій Миколайович (підпис)

Календарний план виконання роботи

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової	01.11.2021	
2.	Вивчення теми та ознайомлення з ключовими її моментами, потрібними для написання роботи	01.01.2022	
3.	Написання текстової частини роботи	15.03.2022	
4.	Створення практичної частини	01.05.2022	
5.	Корегування роботи згідно із зауваженнями керівника	14.05.2021	

АНОТАЦІЯ

У цій роботі розглядаються особливості роботи з пам’яттю з розвитком C++ та виходом нових стандартів. Також проведений аналіз деяких інструментів мови, що часто використовуються, також нових інструментів, які нещодавно ввійшли в стандарт. Практичною частиною було комбінування ефективного методу управління пам’яті для малих об’єктів й використання програмованих ресурсів пам’яті.

Ключові слова: ресурс пам’яті, алокатор, динамічна пам’ять, розумні вказівники.

ЗМІСТ

ВСТУП

В C++ існує 3 типи пам’яті: static/global, stack, heap.

Static/Global - це та область пам’яті, яка контролюється компілятором і виділяється до початку виконання програми, за нею закріплюється деяка стала

адреса(зазвичай в самому виконуваному файлі), яка залишається незмінною під час виконання програми. Дані, які є static або global “живуть”до кінця виконання програми і пам’ять, виділена для них, очищується після завершення її виконання.

Stack - це пам’ять, яка виділяється і очищується автоматично під час виконання програми. За управління цією пам’яттю відповідає компілятор.

Heap - це пам’ять, за виділення і очищення якої відповідає програміст. Він явно викликає функції виділення й очищення пам’яті під час виконання програми. Зазвичай кількість пам’яті у купі(heap) набагато більша, ніж у стеку(stack).

Усі типи пам’яті представляють собою так званий memory resource, шматки якого і видаються під змінні, масиви або інші об’єкти програми.

У цій роботі ми розглядатимемо таку роботу з пам’яттю, яку контролює програміст, особливості такої роботи, яка пов’язана з випуском нових версій стандарту. Також подивимося на те, як з пам’яттю, яка виділена не із heap можна працювати як нібито її виділили з heap: тобто самостійно виділяти, очищувати пам’ять з memory resource-у, який може бути взятий, наприклад, зі stack. Такий підхід роботи з пам’яттю(мануальний або динамічний) є дуже важливим, бо існують ситуації, коли наперед невідомо, скільки пам’яті знадобиться задля задоволення певної потреби.

1. MEMORY RESOURCE і ALLOCATOR

Для початку треба розібратися з двома важливими концепціями:

- Ресурсами пам’яті(memory resource);
- Алокаторами(allocators).

Memory resource - це деякий об’єкт, який має довгий час існування і який за потреби виділяє певну частину пам’яті, яку в нього було запрошено. Він створюється один раз і ніколи не копіюється(copy) і не переміщується(move) і якщо кажуть, що деякі ресурси пам’яті однакові, то це значить, що вони є одним і тим самим об’єктом пам’яті. Така поведінка називається об’єктно орієнтованою семантикою(object-oriented semantics).

Allocator - це свого роду вказівник на певний раніше обраний ресурс пам’яті(memory resource). На відміну від ресурсів пам’яті, алокатори можуть бути скопійовані, переміщені. Вони мають вказівникову семантику(pointer semantics).

Таким чином, такі інструменти мов C і C++, як malloc/free(також доступні в C++) та new/delete є алокаторами з ресурсу пам’яті, який є купою(heap). Тобто на запит виділення пам’яті вони перевіряють, чи можливо задовольнити його, і якщо так, то повертають вказівник на виділений шматок пам’яті з memory resource-у, до якого алокатори під’єднані(у випадку з malloc/free і new/delete - купа).

2. МЕТОДИ ДИНАМІЧНОГО УПРАВЛІННЯ ПАМ’ЯТТЮ

2.1. Концепти allocate/deallocate, construct/destruct.

Для того, щоб створити деякий об’єкт у програмі та користуватися ним, треба по порядку зробити 2 речі:

- Виділити пам’ять, достатню для зберігання цього об’єкту(allocate);
- Сконструювати об’єкт(construct).

Після того, як об’єкт закінчує своє існування, треба також по порядку зробити 2 речі:

- Знищити об’єкт(destruct);
- Повернути виділену раніше пам’ять назад до ресурсу пам’яті(deallocate).

В C++ кожен об’єкт має тип. Також розрізняють class types(програмовані типи) і non-class types(фундаментальні типи, масиви, відсилки, вказівники). Останні не мають конструкторів, хоча синтаксис мови дозволяє користуватися ними, ніби вони є. Тому при створенні не програмованих типів для них лише виділяється пам’ять, саме значення зазвичай вважається невизначеним(undefined) і задля коректного використання об’єкту має бути змінене в подальшому. Хоча інструментами мови можна одразу ініціалізувати ці об’єкти певним значенням і це виглядатиме як-ніби було викликано конструктор, але це лише синтаксичний цукор.

Отже, для не програмованих типів можна говорити тільки про виділення і вивільнення пам’яті(allocate/deallocate), в той час як робота з програмованими типами попри це включає в себе роботу з конструкторами і деструкторами(construct/destruct), і після виділення достатньої кількості пам’яті під деякий об’єкт, викликом конструктора цей самий об’єкт має ініціалізуватися, за потреби мають бути виконані певні операції задля коректної побудови, конструювання цього об’єкту. В той же час деструктори викликаються перед вивільненням пам’яті, задля коректного вивільнення ресурсів, якими користувався об’єкт. Такий підхід ще називається RAII[1].

2.2. C-style malloc, free.

Ще за часів мови С у бібліотеці <stdlib.h> були доступні такі функції для роботи з пам’яттю, як malloc, free і деякі інші. В той час не було таких понять, як конструктори, деструктори, тож основною і єдиною функцією, яку виконував цей ряд доступних інструментів було виділення і звільнення пам’яті з купи. Вони доступні і зараз у програмах, компільованих під C++ у тій самій бібліотеці(або <cstdlib>). Як і майже все у цьому світі, такий підхід був не ідеальним і мав свої недоліки. Двома основними функціями, відповідальними за виділення і звільнення пам’яті були malloc, free з такими сигнатурами:

```
void* malloc(size_t size);

void free(void* p);
```

Через те, що той, хто викликає функцію вивільнення пам’яті, не зазначає розмір блоку, на який вказує p, у роботі з пам’яттю виникають деякі труднощі. Насамперед - питання продуктивності. Однією з основних проблем такого підходу роботи з пам’яттю є можлива її фрагментація під час управління нею засобами malloc, free[2]. Це призводить до того, що під час виділення або вивільнення чергового блоку пам’яті, пошук цього блоку може зайняти надто багато часу. Отже, враховуючи це доходимо до висновку, що повернення алокаторами блоку пам’яті назад у ресурс пам’яті має якимось чином знати про розмір цього блоку і використовувати цю інформацію для більш ефективного користування. Але зберігати розмір виділеного блоку поруч з цим самим блоком може бути надто дорого, тож очевидно є потреба у новому рішенні.

2.3. C++-style new, delete.

З початку існування C++ у стандарті були присутні більш підходящі під об’єктно-орієнтовану парадигму інструменти динамічної роботи з пам’яттю,

аналоги malloc/free: new/delete. За своєю суттю вони є лише надбудовою над malloc/free, але з деякими доданими особливостями.

По-перше, було додано специфікацію типів. На відміну від malloc, оператор new знає про тип об’єкту, пам’ять для якого виділяє. Тобто визначення розміру пам’яті для виділення покладається на компілятор. Також вказівник на обраний тип повертається у якості результату.

По-друге, оператори new/delete викликають відповідно конструктор і деструктор об’єктів, з пам’яттю яких працюють.

По-третє, якщо під час виділення пам’яті, виклику конструкторів об’єктів сталася помилка, то, на відміну від malloc-у, оператор new створює виняткову ситуацію(throws exception).

Серед усього іншого, набір операторів new/delete дозволяє явно працювати з масивами, більше того є можливість задати параметри вирівнювання(alignment) об’єктів всередині масиву. Також існує можливість розміщення певного об’єкту по раніше виділеній пам’яті(placement new operator). Також з виходом нових стандартів деякі типи операторів delete змінили свою поведінку з throw на noexcept. Більш детальна інформація щодо operator new[3] і operator delete[4].

Таким чином, C++ запропонував новий механізм для динамічної роботи з пам’яттю, який більше підходить під особливості мови і який вирішував деякі проблеми, присутні у старого методу управління динамічною пам’яттю(хоча проблеми ефективності, все ж таки, залишилися). Також з виходом нових версій стандарту мови до операторів new/delete додавалися нові специфікації(operator new - для роботи з user-defined аргументами: починаючи з C++17[3], operator delete - починаючи з C++20 є можливість задати як саме буде викликатися деструктор об’єкту, з подальшим

вивільненням пам’яті[5], та інші), змінювалися старі, і цей механізм продовжуватиме адаптуватися до можливих потреб користувачів мови задля покращення процесу і якості розробки.

3. РОЗУМНІ ВКАЗІВНИКИ(SMART POINTERS)

3.1. Проблеми new/delete.

Оператори new/delete дозволяють напряму працювати з ресурсом пам’яті - купою, що створює певні проблеми й накладає деякі обов’язки на користувача. Якщо була виділена пам’ять і певний об’єкт був створений за допомогою оператора new, то після завершення використання цього об’єкту він має бути знищений і використані ресурси мають бути явно повернені назад до ресурсу пам’яті за допомогою оператора delete. Такий принцип не дуже

гарний в контексті побудови програм, бо програміст може забути викликати відповідний оператор очищення ресурсів. Також присутня можливість, коли оператор очищення ресурсу був викликаний, але виконання програми не дійшло до нього, бо функція, де було очищення ресурсу, повернула значення раніше, або створила виняткову ситуацію. Також так як робота з окремими об’єктами й масивами об’єктів виконується за допомогою синтаксично різних операторів(new – new[], delete - delete[]), то це означає, що присутня можливість неправильного використання механізму - викликавши operator delete(замість operator delete[]) на створений масив об’єктів за допомогою operator new[] призведе до витоку пам’яті(memory leak[6]). Ще однією проблемою є можливість використання вказівника на вже очищений раніше об’єкт(use-after-free). Ці та деякі інші проблеми змушують шукати нового рішення щодо динамічного управління пам’яттю.

3.2. auto_ptr.

До C++11 рішенням цієї проблеми могло стати використання розумного вказівника auto_ptr. Він управляв об’єктом, створеним за допомогою оператора new і звільняв ресурси, які використовувалися цим об’єктом, по завершенню свого особистого існування[7].

Але через особливості копіювання цього розумного вказівника, в подальших версіях стандарту його було спочатку додано до застарілих(C++11), а потім і взагалі прибрали з мови(C++17).

Попри це auto_ptr поклав початок ідеї використання розумних вказівників задля комбінування динамічного управління пам’яттю й техніки RAII, що наразі є загальноприйнятим гарним підходом роботи з використовуваними ресурсами.

3.3. C++11 smart pointers.

З виходом нового стандарту у 2011 році попередній варіант розумного вказівника був замінений на декілька нових, більш зручних і підходящих під інструменти мови. Цими новими розумними вказівниками були:

- unique_ptr. Цей тип розумного вказівника є єдиним власником деякого динамічно створеного об’єкту(ним також може бути масив об’єктів). "Бути власником” означає мати унікальні права на управління об’єктом. Тобто ніхто, окрім власника, не може звільнити виділений ресурс. Це також означає, що unique_ptr не може бути скопійований(копіювальним конструктором чи відповідним оператором присвоєння), а тільки переміщеним(move). Також цей вид розумного вказівника підтримує operator*. Для того, щоб передати деякий об’єкт на управління цьому розумному вказівнику, в бібліотеці присутня функція make_unique, яка доступна з C++14. Також, починаючи зі стандарту 2020 року в бібліотеці доступні декілька make_unique_for_overwrite функцій, які виконують ту ж функцію, що й make_unique, але без ініціалізації значень[8].
- shared_ptr. Цей тип розумного вказівника дозволяє управляти об’єктом(або масивом об’єктів) у вигляді так званої спільної власності. Це означає, що управляти об’єктом може одночасно декілька власників, які колективно вирішують, коли саме треба звільнити ресурси, виділені їм на використання, в залежності від

кількості посилань на контрольований об’єкт. Цей підхід називається reference counting[9] і досягається за допомогою спеціального об’єкту - блоку контролю, який і рахує кількість посилань на об’єкт і в залежності від цього приймає рішення, чи звільняти ресурси.

shared_ptr також має operator*. Так само, як і unique_ptr, даний тип розумного вказівника має інструменти передачі власності на об’єкт: make_shared(для масивів - починаючи з C++20) та make_shared_for_overwrite(для будь-яких даних, починаючи з C++20 [10]. Важливим правилом при користуванні shared_ptr є не мати декілька контролюючих блоків на один і той самий ресурс, тобто не створювати декілька shared_ptr об’єктів з одного й того ж “сирого” вказівника(raw pointer).

- weak_ptr не є у прямому розумінні вказівником, бо не має operator*. Але у системі розумних вказівників він займає своє заслужене місце. Особливість цього класу - це можливість відслідковувати час існування об’єкту, який зберігається в програмі під керівництвом групи shared_ptr об’єктів, які ним володіють, при чому не володіючи об’єктом. Weak_ptr використовують як “квиток на спектакль”, де квитком є доступ, а спектаклем - контрольований групою shared_ptr-ів об’єкт[11].

4. АЛОКАТОРИ

Ми дійшли до висновку, що динамічна робота з пам’яттю в C++ є невід’ємною частиною мови, але запропоновані стандартом інструменти такого управління мають деякі недоліки. До перелічених раніше проблем слід додати таку: існують такі вбудовані середовища(embedded environments) як, наприклад, програмне забезпечення для польотів, у яких взагалі нема такого поняття, як купа: в такому випадку, уся використовувана в програмі пам’ять має бути завантажена до початку виконання, ще на етапі компіляції. Це призводить до того, що якщо знадобиться динамічно виділяти пам’ять, то має бути механізм, який дозволив би виділити деякий великий блок під час компіляції, і під час роботи програми брати шматки пам’яті звідти. Звісно, було б не дуже гарно, якби єдине вирішення проблеми призвело до того, що програміст не мав би можливості використовувати особливості мови, які пропонує стандарт як, наприклад, std::vector або деякий алгоритм, який

використовує динамічно виділену пам’ять. Тож починаючи зі станадарту 1998 року стандартні контейнери й алгоритми мають такий механізм, як allocator-awareness. Це означає, що будь-який контейнер або алгоритм, якому треба динамічно працювати з пам’яттю, знає звідки цю пам’ять йому взяти. Цю функцію беруть на себе алокатори.

4.1. Алокатори і поліморфізм.

В C++ програмування алокаторів тісно пов’язано з поліморфізмом. Це вирішує велику кількість проблем в організації самої програми та розумінні коду.

В C++ існує 2 різні способи роботи з поліморфізмом: узагальнене програмування(статичний, часу компіляції) - який полягає у створенні поліморфного інтерфейсу у вигляді концептів(concept) і його моделей(models) за допомогою шаблонів, і класичний поліморфізм(динамічний, поліморфізм часу виконання) - спирається на створення базового класу, його наслідування й перевизначення його віртуальних методів класами-нащадками.

Щодо узагальненого програмування алокаторів в C++, потрібно розрізняти 2 речі: allocator concept, що визначає інтерфейс, і allocator model(як std::allocator), що є реалізацією інтерфейса allocator concept.

Більше того, allocator concept є сім’єю шаблонів концептів. Це означає, що Allocator<int> і Allocator<char> є двома різними концептами, а std::allocator<int> є моделлю концепту Allocator<int>, але не є моделлю концепту Allocator<char>.

Кожен алокатор об’єктів типу T(Allocator<T>) зобов’язаний забезпечити користувача можливістю виділення пам’яті, за допомогою функції allocate. Тобто виклик a.allocate(n) має повернути вказівник на достатню кількість

пам’яті для n об’єктів типу T, і ця пам’ять виділяється з ресурсу пам’яті, до якого цей алокатор під’єднаний.

Таким чином, обидві моделі алокаторів відповідають концепту Allocator<int>:

```
struct int_allocator_2014 {
        int *allocate(size_t n, const void *hint = nullptr);
};
struct int_allocator_2017 {
        int *allocate(size_t n);
};
```

Ці два алокатори є моделями концепту Allocator<int>, бо в обох випадках компілятор правильно обробить виклик a.allocate(n). Такий підхід і є узагальненим програмуванням(generic programming).

На відміну від узагальненого програмування, використовуючи класичний поліморфізм, програміст має задати фіксовану сигнатуру методів базового класу, і усі класи-нащадки не мають права змінювати цю сигнатуру:

```
struct classical_base {
        virtual int *allocate(size_t n) = 0;
};
struct classical_derived : public classical_base {
        int *allocate(size_t n) override;
};
```

Класу-нащадку не дозволено змінити список аргументів функції виділення пам’яті, значення, яке воно повертає. Інтерфейс у класичному поліморфізмі є більш визначеним, аніж в узагальненому програмуванні.

4.2. Визначення ресурсу пам’яті за допомогою memory_resource

Як вже було зазначено у цьому розділі, існують випадки, коли звичайні інструменти динамічної роботи з пам'яттю не можуть бути використані. Для вирішення цієї проблеми треба створити свій варіант купи й динамічно управляти ресурсами функціями allocate, deallocate, цим самим симулювавши використання глобальної купи:

```
static char big_buffer[10000];
static size_t index = 0;
void *allocate(size_t bytes) {
        if (bytes > sizeof big_buffer - index) {
                throw std::bad_alloc();
        }
        index += bytes;
        return &big_buffer[index - bytes];
}
void deallocate(void *p, size_t bytes) {
        // drop it on the floor
}
```

Так як функції allocate і deallocate працюють з одними й тими самими даними(big_buffer, index), то з'являється бажання об'єднати все в один клас, завданням якого буде управляти пам'яттю, виділену в big_buffer.

Починаючи з C++17 така можливість підтримується стандартом за допомогою std::pmr::memory_resource:

```
class memory_resource {
        virtual void *do_allocate(size_t bytes, size_t align) = 0;
        virtual void do_deallocate(void *p, size_t bytes, size_t align)=0;
        virtual bool do_is_equal(const memory_resource& rhs) const = 0;
public:
        void *allocate(size_t bytes, size_t align) {
```

```
        return do_allocate(bytes, align);
    }
    void deallocate(void *p, size_t bytes, size_t align) {
        return do_deallocate(p, bytes, align);
    }
    bool is_equal(const memory_resource& rhs) const {
        return do_is_equal(rhs);
    }
};
```

Так як std::pmr::memory_resource є інтерфейсом, то щоб його використати, потрібно його конкретизувати. Наприклад, так:

```
class chunk_resource : public std::pmr::memory_resource
{
    alignas(std::max_align_t) char big_buffer[10000];
    size_t index = 0;

    void *do_allocate(size_t bytes, size_t align) override
    {
        if (align > alignof(std::max_align_t) ||
            (-index % align) > sizeof big_buffer - index ||
            bytes > sizeof big_buffer - index - (-index % align))
        {
            throw std::bad_alloc();
        }
        index += (-index % align) + bytes;
        return &big_buffer[index - bytes];
    }

    void do_deallocate(void *, size_t, size_t) override
```

```
        {
            // drop it on the floor
        }

        bool do_is_equal(const memory_resource& rhs) const override
        {
            return this == &rhs;
        }
};
```

5. ПРАКТИЧНА ЧАСТИНА

За основу практичної роботи було взято приклад алокатора для малих об’єктів(Small Object Allocator) Александреску[12]: Chapter 4 – Small Object Allocation.

Цей алокатор призначений для швидкого управління пам’яттю об’єктів невеликого розміру. Він складається з таких об’єктів: Chunk, FixedAllocator, SmallObjAllocator, SmallAllocator.

Chunk є найменшою одиницею у роботі з пам’яттю - він виділяє об’єкти єдиного сталого розміру і має верхню границю по кількості виділених об’єктів цього розміру.

```
struct FixedAllocator::Chunk
{
    // Constructors and destructor are not defined due to performance concerns
    void init(const std::size_t blockSize, const unsigned char blocks);
    void* allocate(const std::size_t blockSize);
    void deallocate(void* p, const std::size_t blockSize);
    void release();
    inline bool isFilled() const;
    inline bool hasAddress(void* p, const std::size_t blockSize);
    inline std::size_t addressOffset(void *p, const std::size_t blockSize);
    inline bool validOffset(std::size_t offset, const std::size_t blockSize);

    chunk_resource allocator;
    unsigned char* _allocator;
    unsigned char _firstAvailableBlock;
    unsigned char _blocksAvailable;
};
```

FixedAllocator управляє об’єктами типу Chunk і також працює з виділенням пам’яті фіксованого розміру.

```
class FixedAllocator
{
private:
    struct Chunk;

    std::size_t _blockSize;
    unsigned char _numBlocks;
    std::vector<Chunk> _chunks;

    // Pointers to the last chunks that were used for allocation and deallocation
    // play the role of a simple, but efficient cache
    Chunk* _allocChunk = nullptr;
    Chunk* _deallocChunk = nullptr;

    std::size_t _totalBlocks = 0;
    std::size_t _allocatedBlocks = 0;

    void prepareChunk();

public:
    FixedAllocator() { };
    ~FixedAllocator();
    FixedAllocator(const FixedAllocator &) = delete;
    FixedAllocator& operator=(const FixedAllocator &) = delete;

    // Move constructor and assignment for efficient swap and emplace
    FixedAllocator(FixedAllocator &&);
    FixedAllocator& operator=(FixedAllocator &&);

    void initialize(std::size_t blockSize, const unsigned char numBlocks);
    void* allocate();
    void deallocate(void* p);

    std::size_t blockSize() const { return _blockSize; };
};
```

SmallObjAllocator управляє FixedAllocator-ами й працює вже з різними розмірами пам’яті об’єктів.

```
class SmallObjAllocator
{
private:
    std::vector<FixedAllocator> _pool;
    /**
     * Max size of the objects that are considered small
     */
    std::size_t _maxObjectSize;
    std::size_t _chunkSize;

    // "Cache" for the FixedAllocators
    FixedAllocator* _lastAlloc = nullptr;
    FixedAllocator* _lastDealloc = nullptr;

    // Find the position of the required allocator using binary search
    std::vector<FixedAllocator>::iterator findFixedAllocatorPosition(std::size_t numBytes);

public:
    SmallObjAllocator(std::size_t chunkSize, std::size_t maxObjectSize);
    ~SmallObjAllocator();

    void* allocate(std::size_t numBytes);
    void deallocate(void* p, std::size_t size);

};
```

SmallObject є сервісом до побудованої структури й надає можливість працювати з нею.

```
template <std::size_t chunkSize, std::size_t maxSmallObjectSize>
class SmallObject
{
public:
    static void* operator new(std::size_t size);
    /**
     * Little but important trick happens here:
     * C++ already has a way to get any object's size in runtime
     * and we can access it by overloading the delete operator with
     * the second argument - size_t size
     */
    static void operator delete(void* p, std::size_t size);
    virtual ~SmallObject() {}
};

template <std::size_t chunkSize, std::size_t maxSmallObjectSize>
void* SmallObject<chunkSize, maxSmallObjectSize>::operator new(std::size_t size)
{
    return SmallObjAllocatorSingleton<chunkSize, maxSmallObjectSize>::instance().allocate(size);
}

template <std::size_t chunkSize, std::size_t maxSmallObjectSize>
void SmallObject<chunkSize, maxSmallObjectSize>::operator delete(void* p, std::size_t size)
{
    SmallObjAllocatorSingleton<chunkSize, maxSmallObjectSize>::instance().deallocate(p, size);
}
```

SmallObjAllocatorSingleton - це, як зрозуміло з назви, синглтон, який надає можливість користуватися SmallObjAllocator-ом з будь-якого місця програми.

```
template <std::size_t chunkSize, std::size_t maxSmallObjectSize>
class SmallObjAllocatorSingleton
{
public:
    SmallObjAllocatorSingleton() = delete;
    static SmallObjAllocator& instance();
};

template <std::size_t chunkSize, std::size_t maxSmallObjectSize>
SmallObjAllocator& SmallObjAllocatorSingleton<chunkSize, maxSmallObjectSize>::instance()
{
    static SmallObjAllocator allocator(chunkSize, maxSmallObjectSize);
    return allocator;
}
```

Попри те, що даний алокатор є ефективним у роботі з об’єктами малого розміру, він все ж таки використовує динамічну пам’ять при ініціалізації Chunk-ів, що потім повертає користувачу. У цій роботі було знайдено альтернативне рішення - не використовувати динамічно виділену пам’ять, а скористатися програмованим ресурсом пам’яті chunk_resource з минулого розділу, який є нащадком std::pmr::memory_resource-у, доступного у 17 стандарті мови саме задля таких цілей. Тож місця запрошення і повернення пам’яті всередині Chunk-ів зміниться з

```
_allocator = new unsigned char[blockSize * blocks];
delete [] _allocator;
```

на

```
_allocator = allocator.allocate(blockSize * blocks);
allocator.deallocate(_allocator, 0);
```

Де allocator - це chunk_resource, який зберігається всередині Chunk-у. Цим самим інструментами мови було досягнуто комбінування ефективного алокатора для малих об’єктів й можливості уникнути використання динамічної пам’яті, що дозволяє користуватися ним навіть у дуже обмежених середовищах.

6. ВИСНОВКИ

Таким чином, у цій роботі було розглянуто різні типи пам’яті у C++, особливості й можливості їх використання. Один з цих типів - динамічна робота з пам’яттю є дуже важливим, невід’ємним аспектом у програмуванні на C++, який задовольняє велику кількість потреб, але в той же час має багато недоліків, які по можливості виправлялися з виходом нових стандартів(розумні вказівники і тд).

Через особливості реалізації динамічної роботи з пам’яттю, існують випадки, коли програмовані алокатори є більш ефективними для вирішення певної задачі. Більше того, існують системи, в яких просто неможливо працювати з купою. Таким чином, було потрібно знайти рішення цієї проблеми. Цим рішенням виявилася можливість, починаючи з C++17 самостійно створювати програмовані купи за допомогою std::pmr::memory_resource-у, і працювати з ними нібито як з глобальною купою, виділяючи й вивільняючи пам’ять за потреби. Також у стандартній бібліотеці доступні такі варіанти ресурсів пам’яті, похідних від std::pmr::memory_resource-у, як new_delete_resource(для роботи з new/delete), null_memory_resource(який завжди повертає nullptr на запрошення пам’яті) та деякі інші, корисні у певних випадках.

Таким чином, в C++ стає більш зручно працювати з пам’яттю, використовуючи різні інструменти мови, які можна підлаштувати під свої конкретні потреби, аніж це було ще декілька стандартів тому.

7. СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] https://en.cppreference.com/w/cpp/language/raii

[2] https://en.wikipedia.org/wiki/Fragmentation_(computing)

[3] https://en.cppreference.com/w/cpp/memory/new/operator_new

[4] https://en.cppreference.com/w/cpp/memory/new/operator_delete

[5] https://en.cppreference.com/w/cpp/memory/new/destroying_delete

[6] https://en.wikipedia.org/wiki/Memory_leak

[7] https://en.cppreference.com/w/cpp/memory/auto_ptr

[8] https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique

[9] https://en.wikipedia.org/wiki/Reference_counting

[10] https://en.cppreference.com/w/cpp/memory/shared_ptr

[11] https://en.cppreference.com/w/cpp/memory/weak_ptr

[12] https://www.mimuw.edu.pl/~mrp/cpp/SecretCPP/Addison-Wesley%20-%20Modern%20C++%20Design.%20Generic%20Programming%20and%20Design%20Patterns%20Applied.pdf