Тема
Робота з динамічною пам’яттю

Мета
Навчитися виділяти місце під об’єкти динамічно. Навчитися створювати та використовувати конструктор копіювання та переміщення, перевантажувати оператор присвоєння та переміщення. Ознайомитися з принципами створення та функціонування деструкторів.

Теоретичні відомості
Види пам’яті
Кожна змінна (об’єкт) програми розміщується в адресному просторі програми в одному з видів пам’яті:

- Статична (глобальна)
- Локальна (стек, автоматична)
- Динамічна (купа).

В статичній пам’яті розміщуються глобальні змінні (оголошені поза всіма блоками – функцією, методом, класом) і статичні змінні (перед типом яких вказується ключове слово static, при цьому змінна може знаходиться де завгодно, в тому числі і в тілі функції, методу чи класу). Різниця між статичною та глобальною змінними проявляється, коли програма складається з декількох файлів: нестатичні глобальні змінні доступні в будь-яких файлах вихідного коду, а статичні – тільки в тому файлі, де були оголошені. Об’єкти, розміщені у цій пам’яті, доступні протягом усього часу виконання програми. Однак в статичній пам'яті не рекомендується тримати великі об'єкти (наприклад, масиви). Також, порядок створення об’єктів у статичній пам’яті не визначений і може бути довільним. Це означає, код конструкторів таких об’єктів не має звертатись до інших глобальних об’єктів, бо вони ще можуть бути не створеними. Тому хорошим кодом з використанням ООП вважається програмний код, в якому використання глобальних і статичних змінних зведено до мінімуму.

Локальна пам’ять або стек – частина адресного простору програми, де розміщуються змінні функцій та методів. Пам’ять для них виділяється при вході в блок програми і вивільняється при виході з нього. Також в ній розміщуються значення фактичних параметрів функції та додаткова інформація необхідна для роботи механізму виклику функцій. Виділення та звільнення такої пам’яті надзвичайно швидке. Проте її розмір дуже обмежений (від десятків кілобайт до десятків мегабайт), тому не рекомендується розміщувати в ній великі об’єкти, а також потрібно мінімізувати глибину вкладеності викликів функцій (обмежувати глибину рекурсії, використати ітераційні алгоритми замість рекурсивних, не викликати без необхідності функції).

Динамічна пам’ять або купа – решта адресного простору програми, де можуть бути розміщені дані. Вона виділяється і вивільняється за допомогою спеціальних інструкцій. Це дозволяє виділити пам’ять, розмір якої відомий лише під час виконання, наприклад – залежить від вхідних даних. Також це дозволяє зменшити використання стеку та звільнювати пам’ять, в якій більше нема потреби. Розмір доступної динамічної пам’яті залежить від архітектури комп’ютера, але для поширених сьогодні 64-бітних архітектур її розмір залежить лише від кількості доступної пам’яті у системі. Але відповідальність за звільнення цієї пам’яті лежить цілковито на розробнику.

Пам’ять виділена динамічно не має імені, а доступ до неї можливий тільки через вказівники, які програміст може зв’язувати з виділеною ділянкою пам’яті.

В таблиці нижче зведено переваги та недоліки кожного виду пам’яті.

Таблиця 1. Види пам’яті.

Тип пам’яті	Переваги	Недоліки
Статична	• Виділення та звільнення відбувається автоматично. • Доступна під час всього життя програми • Може бути доступна з будь-якого місця програми	• Розмір має бути фіксований та відомий на етапі компіляції • Збільшує розмір виконавчого файлу/бібліотеки (особливо якщо це ініціалізована не нулями пам’ять) • Порядок створення та знищення об’єктів у цій пам’ять не визначений • Збільшує час початку роботи програми (особливо, якщо створюються об’єкти із складними конструкторами)
Локальна	• Надзвичайно швидке виділення та звільнення • Автоматичне виділення та звільнення	• Дуже обмежений розмір, який ще й до того ж невідомий • Може викликати переповнення стеку
Динамічна	• Дозволяє виділити пам’ять лише за потреби та у потрібній кількості. • Дозволяє звільнити більше не потрібну пам’ять • Розмір доступної пам’яті обмежується лише середовищем виконання (скільки пам’яті доступно в системі)	• Необхідно вручну виділяти та звільняти, що може призвести до витоку пам’яті, подвійних звільнень та інших помилок. • Операції виділення та звільнення займають відносно багато часу.

Динамічна пам’ять в мові C++ виділяється за допомогою операторів new та new [] і звільняється за допомогою операторів delete та delete []. Оператори new/delete використовуються для створення та знищення об’єктів, а new [] / delete [] – для створення та знищення масивів об’єктів.

Після власне виділення пам’яті оператори new та new [] викликають конструктори об’єктів, а оператори delete та delete [] викликають деструктори об’єктів, а тоді звільняють пам’ять.

Існує кілька варіантів оператора new:

1. Оператор new: стандартний оператор new, який виділяє блок пам'яті певного розміру і повертає вказівник на перший байт пам'яті.

```
// Виділення пам'яті для одного об'єкта
int* p = new int;
*p = 42;
std::cout << *p << '\n';

// Виділення пам'яті для масиву об'єктів
int* arr = new int[10];
for (int i = 0; i < 10; ++i) {
    arr[i] = i;
    std::cout << arr[i] << " ";
}
std::cout << '\n';
```

Рис. 1. Оператор new

2. Оператор placement new: цей оператор не виділяє пам'ять, але використовує вказівник на вже виділене місце під об'єкт. Він викликає конструктор об'єкта і повертає вказівник на його початок.

```
// Використання placement new
void* buffer = malloc(sizeof(MyClass)); // Виділення пам'яті
MyClass* obj = new (buffer) MyClass(); // Створення об'єкта за допомогою placement new
obj->do_something(); // Використання об'єкта
obj->~MyClass(); // Виклик деструктора об'єкта
free(buffer); // Звільнення пам'яті
```

Рис. 2. Оператор placement new

3. Оператор nothrow new: цей оператор поводиться так само, як стандартний оператор new, але він не генерує виключення у випадку неуспішного виділення пам'яті. Замість цього, він повертає нульовий вказівник.

```
// Використання nothrow new
int* p = new (std::nothrow) int[1000000000000000000]; // Виділення пам'яті з nothrow new
if (p == nullptr)
{
    std::cout << "Memory allocation failed\n";
}
else
{
    std::cout << "Memory allocation successful\n";
    delete[] p; // Звільнення пам'яті
}
```

Рис. 3. Оператор nothrow new

Можна також використовувати функції C, такі як malloc, calloc, realloc для виділення/перевиділення пам’яті і відповідну їм функцію free для звільнення виділеної пам’яті. Проте специфіка роботи оператора new і наведених вище функцій відрізняється. Тому не можна змішувати виклики оператора new і функції free, чи навпаки функції malloc, наприклад, і оператора delete.

Якщо не звільняти виділену динамічну пам’ять, то вона буде зайнята до закінчення програми, що зменшує доступний обсяг вільної пам’яті і може призводити до некоректної роботи програми чи до її непередбачуваного завершення. Тому завжди, як тільки виділена пам’ять стає непотрібною, її необхідно звільняти.

Об’єкти в статичній, локальній та динамічній пам’яті

Об’єкти, як і змінні будь-якого стандартного типу, можна виділяти в усіх трьох видах пам’яті.

Приклади роботи з об’єктами в різних видах пам’яті наведено в лістингу:

```
#include <iostream>

using namespace std;

class MyClass
{
private:
    static size_t m_allocated_objects_number;
    size_t m_object_id;

public:
    MyClass()
    {
        m_object_id = m_allocated_objects_number++;
        cout << "MyClass instance created with default constructor. Id = "
            << m_object_id << '\n';
    }

    MyClass(const MyClass& another)
    {
```

```
        m_object_id = m_allocated_objects_number++;
        cout << "MyClass instance created as copy of instance ("
            << another.m_object_id << "). Id = " << m_object_id << '\n';
    }

    MyClass(int p)
    {
        m_object_id = m_allocated_objects_number++;
        cout << "MyClass instance created with int parameter "
            << p << ". Id = " << m_object_id << '\n';
    }

    ~MyClass()
    {
        cout << "MyClass instance destroyed. Id = " << m_object_id << '\n';
    }
};

size_t MyClass::m_allocated_objects_number = 0;

MyClass g_my_object_1;
MyClass g_my_object_2(1);
static MyClass g_my_object_3(2);

int main(int, const char*[])
{
    cout << "Stage 1:\n";
    MyClass my_object_4(4);
    MyClass my_objects_1[3];

    {
        cout << "Stage 2:\n";
        MyClass my_object_5(5);
        MyClass my_objects_2[4];
        cout << "Stage 3:\n";
    }

    cout << "Stage 4:\n";
    MyClass* my_object_6 = new MyClass(6);
    MyClass* my_object_7 = nullptr;
    MyClass* my_objects_3 = new MyClass[3];
    MyClass  my_object_8 = my_object_4;
    MyClass  my_object_9 = *my_object_6;
    my_object_7 = new MyClass(3);

    {
        cout << "Stage 5:\n";
        MyClass  my_object_10 = my_objects_3[2];
        MyClass* my_object_11 = new MyClass(my_object_9);
        MyClass* my_object_12 = new MyClass(*my_object_7);

        cout << "Stage 6:\n";
        delete my_object_11;
    }

    cout << "Stage 7:\n";
    delete my_object_6;
    delete[] my_objects_3;

    cout << "Stage 8:\n";

    return 0;
}
```

Рис. 4. Приклад роботи з об’єктами в різних типам пам’яті

В результаті виконання програми матимемо наступний вивід на консоль:

```
MyClass instance created with default constructor. Id = 0
MyClass instance created with int parameter 1. Id = 1
MyClass instance created with int parameter 2. Id = 2
Stage 1:
MyClass instance created with int parameter 4. Id = 3
MyClass instance created with default constructor. Id = 4
MyClass instance created with default constructor. Id = 5
MyClass instance created with default constructor. Id = 6
Stage 2:
MyClass instance created with int parameter 5. Id = 7
MyClass instance created with default constructor. Id = 8
MyClass instance created with default constructor. Id = 9
MyClass instance created with default constructor. Id = 10
MyClass instance created with default constructor. Id = 11
Stage 3:
MyClass instance destroyed. Id = 11
MyClass instance destroyed. Id = 10
MyClass instance destroyed. Id = 9
MyClass instance destroyed. Id = 8
MyClass instance destroyed. Id = 7
Stage 4:
MyClass instance created with int parameter 6. Id = 12
MyClass instance created with default constructor. Id = 13
MyClass instance created with default constructor. Id = 14
MyClass instance created with default constructor. Id = 15
MyClass instance created as copy of instance (3). Id = 16
MyClass instance created as copy of instance (12). Id = 17
MyClass instance created with int parameter 3. Id = 18
Stage 5:
MyClass instance created as copy of instance (15). Id = 19
MyClass instance created as copy of instance (17). Id = 20
MyClass instance created as copy of instance (18). Id = 21
Stage 6:
MyClass instance destroyed. Id = 20
MyClass instance destroyed. Id = 19
Stage 7:
MyClass instance destroyed. Id = 12
MyClass instance destroyed. Id = 15
MyClass instance destroyed. Id = 14
MyClass instance destroyed. Id = 13
Stage 8:
MyClass instance destroyed. Id = 17
MyClass instance destroyed. Id = 16
MyClass instance destroyed. Id = 6
MyClass instance destroyed. Id = 5
MyClass instance destroyed. Id = 4
MyClass instance destroyed. Id = 3
MyClass instance destroyed. Id = 2
MyClass instance destroyed. Id = 1
MyClass instance destroyed. Id = 0
```

Рис. 5. Вивід програми із Рис.4.

Варто звернути увагу на те, що при роботі із об’єктами my_object_7 та my_object_12 були допущені помилки, оскільки об’єкти не були видалені після використання, про що свідчить вивід програми.

Робота з динамічною пам’яттю в об’єктах

Якщо в об’єкті виділяється динамічна пам’ять, на яку вказує його поле-вказівник (об’єкт володіє виділеною пам’яттю), то ця пам’ять обов’язково має бути звільнена об’єктом або передана у володіння в інше місце програми. При цьому потрібно пам’ятати, що стандартні конструктор копіювання та оператор присвоєння не виконують “глибокого” копіювання (не копіюють виділену пам’ять, а копіюють тільки вказівники на неї). Тому при їх виконанні можливі ситуації, коли різні об’єкти міститимуть вказівники на одну і ту ж ділянку пам’яті. При цьому, якщо об’єкти не знатимуть про таку ситуацію, можливе повторне звільнення уже звільненої пам’яті, що приведе до фатальної помилки виконання програми. Тому, якщо екземпляри класів виділяють динамічну пам’ять і

зберігають вказівники на неї у своїх полях, необхідно для них перевизначати конструктор копіювання та оператор присвоєння, а для звільнення пам’яті при потребі – деструктор. Це правило називають Правило трьох (Rule of three).

Починаючи із C++11 є також можливість визначати конструктор переміщення та оператор переміщення. Тому правило називають Правило п’яти (Rule of Five). На відміну від Правила трьох, не визначення конструктора переміщення та оператора переміщення є не помилкою, а втраченою можливістю оптимізації.

Існує також Правило нуля (Rule of Zero) - це принцип, який рекомендує використовувати вбудовані типи та стандартні класи C++, якщо це можливо, та відмовлятися від явного управління пам'яттю, що дозволяє зменшити кількість помилок та полегшити написання коду.

При використанні правила нуля класи можуть використовувати вбудовані типи даних та/або стандартні класи C++ (такі як std::vector, std::string тощо), які вже мають вбудовану підтримку автоматичного управління пам'яттю. Це дозволяє зменшити кількість помилок, пов'язаних з управлінням пам'яттю, та полегшити написання коду.

Альтернативою перевизначенню конструктора копіювання, оператора присвоєння та деструктора є використання спеціальних вказівників (std::auto_ptr до C++11, std::shared_ptr, std::unique_ptr для C++11 і вище).

‘std::unique_ptr’ у C++ - це "власник" об'єкта, який забезпечує автоматичне управління пам'яттю для ресурсу, на який він вказує. Унікальний вказівник відрізняється від спільного вказівника ‘std::shared_ptr’ тим, що він гарантує, що об'єкт, на який він вказує, має тільки один власник. Тому коли унікальний вказівник виходить за межі області видимості то об'єкт буде автоматично звільнений.

‘std::unique_ptr’ використовують, коли потрібно забезпечити автоматичне управління пам'яттю для одного об'єкта в одному місці коду. Якщо об'єкт потрібно передати в інший об'єкт або функцію, то можна використовувати std::move для передачі власності унікального вказівника.

```
#include <iostream>
#include <memory>

class CMyClass
{
public:
    CMyClass() { std::cout << "CMyClass constructor called.\n"; }
    ~CMyClass() { std::cout << "CMyClass destructor called.\n"; }

    void SayHello() const { std::cout << "Hello from CMyClass!\n"; }
};

int main()
{
    std::unique_ptr<CMyClass> ptr(new CMyClass());
    ptr->SayHello();
    return 0;
}
```

Рис. 6. Приклад використання std::unique_ptr

У цьому прикладі ми створюємо об'єкт класу CMyClass за допомогою оператора ‘new’, і передаємо його адресу конструктору ‘std::unique_ptr’. За допомогою стрілочного оператора -> ми можемо отримати доступ до методів класу через вказівник ptr. Після під час завершення функції main, коли об'єкт ptr виходить за межі області видимості, автоматично викликається деструктор класу CMyClass, що виводить повідомлення на екран.

Вивід програми на екран буде наступний:

```
CMyClass constructor called.
Hello from CMyClass!
CMyClass destructor called.
```

Рис. 7. Вивід програми з Рис.6.

Конструктор копіювання та переміщення

Конструктор копіювання та переміщення - це спеціальні конструктори класу, які дозволяють створювати нові об'єкти на основі існуючих.

Конструктор копіювання - створює новий об'єкт, копіюючи значення всіх членів існуючого об'єкта в новий об'єкт. Цей конструктор викликається, коли створюється новий об'єкт з існуючого об'єкту. Конструктор копіювання має наступну сигнатуру:

```
ClassName(const ClassName & other);
```

Рис. 8. Сигнатура конструктора копіювання

Конструктор переміщення - створює новий об'єкт, переміщуючи всі члени існуючого об'єкту в новий об'єкт і "викрадаючи" ресурси існуючого об'єкту. Цей конструктор викликається, коли створюється новий об'єкт з існуючого об'єкту, але існуючий об'єкт після цього стає недійсним. Конструктор переміщення має наступну сигнатуру:

```
ClassName(ClassName && other) noexcept;
```

Рис. 9. Сигнатура конструктора переміщення

Нижче наведено приклад реалізації конструкторів копіювання та переміщення:

```
#include <iostream>
#include <cstring>

class MyClass {
public:
    // Конструктор за замовчуванням
    MyClass() : data(nullptr), size(0) {}

    // Конструктор копіювання
    MyClass(const MyClass& other) : size(other.size) {
        data = new int[size];
        std::memcpy(data, other.data, size * sizeof(int));
    }

    // Конструктор переміщення
    MyClass(MyClass&& other) noexcept : data(nullptr), size(0) {
        std::swap(data, other.data);
        std::swap(size, other.size);
    }

    // Деструктор
    ~MyClass() {
        delete[] data;
    }

private:
    int* data;
    int size;
};

int main() {
    MyClass a;
    MyClass b(a); // Конструктор копіювання
    MyClass c(std::move(b)); // Конструктор переміщення
```

```
    return 0;
}
```

Рис. 10. Приклад класу із коструктором копіювання та переміщення

Оператори копіювання та переміщення

Оператор копіювання (copy assignment operator) та оператор переміщення (move assignment operator) є спеціальними операторами класу у мові C++, які використовуються для присвоєння одного об'єкту іншому.

Оператор копіювання зазвичай копіює дані з одного об'єкту до іншого, в той же час оператор переміщення виконує переміщення (тобто "викрадення") ресурсів з одного об'єкту до іншого. Це може бути корисним, наприклад, коли ми маємо клас, що керує динамічно виділеними ресурсами, такими як пам'ять або файли, і ми хочемо передати управління цими ресурсами з одного об'єкту до іншого, замість копіювання цих ресурсів.

Оператор копіювання має наступну сигнатуру:

```
ClassName& operator=(const ClassName& other)
```

Рис. 11. Сигнатура оператора копіювання

Оператор переміщення має наступну сигнатуру:

```
ClassName& operator=(ClassName&& other) noexcept
```

Рис. 12. Сигнатура оператора переміщення

Нижче подано приклад класу, в якому перевизначені оператори копіювання та переміщення:

```
#include <iostream>
#include <cstring>

class MyClass {
public:
    // Конструктор за замовчуванням
    MyClass() : data(nullptr), size(0) {}

    // Деструктор
    ~MyClass() {
        delete[] data;
    }

    // Оператор копіювання
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            std::memcpy(data, other.data, size * sizeof(int));
        }
        return *this;
    }

    // Оператор переміщення
    MyClass& operator=(MyClass&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = nullptr;
            size = 0;
            std::swap(data, other.data);
            std::swap(size, other.size);
        }
```

```
        return *this;
    }

private:
    int* data;
    int size;
};

int main() {
    MyClass a, b, c;
    a = c; // Оператор копіювання
    b = std::move(a); // Оператор переміщення
    return 0;
}
```

Рис. 13. Приклад класу із оператором копіювання та переміщення

Ідіома RAII (Resource Acquisition Is Initialization)

Згідно з ідіомою RAII утримання ресурсу (пам’ять, файли, мережеві з’єднання, блокування м’ютексів та інші) тісно пов’язане із часом життя об’єкта. Отримання ресурсу відбувається під час створення об’єкта, а звільнення – під час знищення об’єкта за допомогою деструктора.

Використання RAII має такі переваги:

1. Інкапсуляція отримання та звільнення ресурсів. Код, який відповідає за отримання та звільнення ресурсів пишуть в тому ж класі.
2. Безпека при виняткових ситуаціях для локальних ресурсів. При виникнення виняткової ситуації, усі об’єкти утворені в локальних областях видимості, будуть знищені автоматично, а їхні конструктори відповідно звільнять усі отримані ресурси.
3. Зручність використання та спрощення коду.

Приклад використання RAII розглядається у лабораторній роботі №8.

Завдання

1. Переглянути код в прикладі. Пояснити вивід програми.
2. Створити клас відповідно до варіанту.
3. Розробити для класу конструктор за замовчуванням та декілька звичайних конструкторів. Реалізувати функції-члени та перевантажити оператори відповідно до завдання (див. Додаток). При цьому вибір механізму перевантаження обрати самостійно (чи метод, чи дружня-функція).
4. Створити конструктор копіювання.
5. Створити конструктор переміщення.
6. Перевантажити оператор присвоєння.
7. Перевантажити оператор переміщення.
8. Створити деструктор для вивільнення динамічно виділеної пам’яті.
9. Об'єкти класу розмістити в динамічній пам'яті.
10. Продемонструвати розроблені можливості класу завдяки створеним тестам для Google Test.
11. Розробити інший аналогічний клас, в якому для роботи із динамічною пам’яттю використати std::unique_ptr. Цей клас має мати деструктор за замовчуванням. Звільнення пам’яті забезпечить std::unique_ptr.
12. Переконатись, що тести для першого класу успішно проходять і для другого.
13. Оформити звіт до лабораторної роботи.

Варіанти завдань

Варіант	Завдання
1	Клас CMatrix – двовимірна матриця дійсних чисел. Пам’ять під елементи матриці повинна виділятися динамічно. Елементи матриці повинні зберігатися у двовимірному масиві Реалізувати такі функції члени: • Знаходження максимального значення матриці. • Знаходження мінімального значення матриці. • Знаходження максимального значення в заданому рядку. • Знаходження мінімального значення в заданому стовпці. • Зміна розмірів матриці. Перевантажити оператори: • Додавання • Віднімання • Множення • Множення на скаляр. • Введення матриці з std::istream (>>) • Виведення матриці у std::ostream (<<) Забезпечити можливість отримання значення елементу [i][j] подібно до доступу до елементів звичайного двовимірного масиву.
2	Клас CArray – одновимірний масив дійсних чисел. Пам’ять під елементи масиву повинна виділятися динамічно. Реалізувати такі функції члени: • Знаходження максимального значення. • Знаходження мінімального значення. • Знаходження середнього арифметичного значення масиву. • Сортування елементів масиву методом вибірки за спаданням. • Сортування елементів масиву методом бульбашки за зростанням. • Зміна розмірів масиву. Перевантажити оператори: • Додавання (почленне додавання елементів масиву) • Віднімання (почленне віднімання елементів масиву)

	• Множення на скаляр. • Введення масиву з std::istream (>>) • Виведення масиву у std::ostream (<<) Забезпечити можливість отримання значення елементу [i] подібно до доступу до елементів звичайного одновимірного масиву.
3	Клас CSet – множина цілочисельних значень. Пам'ять під елементи множини повинна виділятися динамічно. Реалізувати такі функції члени: • Перевірка на належність елемента множині. • Знаходження потужності множини. • Знаходження максимального значення серед значень множини. • Знаходження мінімального значення серед значень множини. • Додавання елемента до множини. • Видалення елемента з множини. Перевантажити оператори: • Додавання (об'єднання множин). • Віднімання (перетин множин). • Ділення (різниця множин) • Введення множини з std::istream (>>) • Виведення множини у std::ostream (<<)
4	Клас CMultiSet – множина цілочисельних значень, які можуть повторюватись. Реалізувати такі функції члени: • Кількість елементів певного значення у множині. • Знаходження потужності множини. • Знаходження максимального значення серед значень множини. • Знаходження мінімального значення серед значень множини. • Додавання елемента до множини. • Видалення елемента з множини. Перевантажити оператори: • Додавання (об'єднання множин). • Віднімання (перетин множин). • Ділення (різниця множин) • Введення множини з std::istream (>>) • Виведення множини у std::ostream (<<)
5	Клас CDictionaryIntToInt – асоціативний масив із цілочисельними ключами та значеннями. Реалізувати такі функції члени: • додати елемент до масиву (ключ та значення) • знайти значення із заданим ключем • вилучити значення із заданим ключем Перевантажити оператори: • Додавання (об'єднання масивів). Якщо ключ присутніх в обох, то взяти значення із другого • Віднімання (перетин множин ключів). • Ділення (різниця множин ключів). • Введення масиву з std::istream (>>) • Виведення масиву у std::ostream (<<)
6	Клас CString – стрічка символів (масив елементів символьного типу). Пам'ять під елементи масиву повинна виділятися динамічно.

	Реалізувати такі функції члени: • Видалення заданого символу із стрічки. • Знаходження кількості входжень заданого символу у стрічку. • Сортування символів стрічки в алфавітному порядку у порядку зростання. • Сортування символів стрічки в алфавітному порядку у порядку спадання. • Вставлення однієї стрічки у іншу з заданої позиції. • Пошук підстрічки у стрічці. Перевантажити оператори: • Додавання (конкатенація стрічок) • Віднімання (видалення із першої стрічки першого входження другої) • Введення стрічки з std::istream (>>) • Виведення стрічки у std::ostream (<<) Забезпечити можливість отримання значення елементу [i] подібно до доступу до елементів звичайного одновимірного масиву.
7	Клас CSingleLinkedList – однозв'язний список дійсних чисел. Пам’ять під елементи списку повинна виділятися динамічно. Реалізувати такі функції члени: • Отримання першого елемента • Отримання останнього елемента • Отримання кількості елементів у списку • Знаходження максимального значення. • Знаходження мінімального значення. • Знаходження середнього арифметичного значення списку. • Сортування елементів списку методом вибірки за спаданням. • Сортування елементів списку методом бульбашки за зростанням. Перевантажити оператори: • Додавання (почленне додавання елементів до списку) • Віднімання (почленне видалення елементів списку) • Множення на скаляр. • Введення списку з std::istream (>>) • Виведення списку у std::ostream (<<) Забезпечити можливість отримання значення елементу [i] подібно до доступу до елементів звичайного одновимірного масиву.
8	Клас CDoubleLinkedList – двозв'язний список дійсних чисел. Пам’ять під елементи списку повинна виділятися динамічно. Реалізувати такі функції члени: • Отримання першого елемента зліва. • Отримання першого елемента справа. • Отримання кількості елементів у списку • Знаходження середнього арифметичного значення списку. • Сортування елементів списку методом вибірки за спаданням. • Сортування елементів списку методом бульбашки за зростанням. • Налаштування виведення списку (зліва на право чи справа на ліво). Перевантажити оператори: • Додавання (почленне додавання елементів до списку) • Віднімання (почленне видалення елементів списку) • Множення на скаляр. • Введення списку з std::istream (>>) • Виведення списку у std::ostream (<<) Забезпечити можливість отримання значення елементу [i] подібно до доступу до елементів звичайного одновимірного масиву.

9	Клас CStack – стек дійсних чисел. Пам’ять під елементи стеку повинна виділятися динамічно. Реалізувати такі функції члени: • Додавання елементу у стек. • Видалення елементу зі стеку. • Отримання значення останнього доданого елементу. • Очищення стеку. • Перевірка, чи стек порожній. • Перевірка, чи стек заповнений. Перевантажити оператори: • Додавання (почленне додавання елементів до стеку) • Віднімання (почленне видалення елементів зі стеку) • Множення на скаляр. • Введення стеку з std::istream (>>) • Виведення стеку у std::ostream (<<)
10	Клас CQueue – однобічна черга дійсних чисел. Пам’ять під елементи черги повинна виділятися динамічно. Реалізувати такі функції члени: • Отримання кількості елементів у черзі. • Знаходження максимального значення. • Знаходження мінімального значення. • Знаходження середнього арифметичного значення черги. • Очищення черги. • Перевірка, чи черга порожня. Перевантажити оператори: • Додавання (почленне додавання елементів до черги) • Віднімання (почленне видалення елементів з черги) • Множення на скаляр. • Введення черги з std::istream (>>) • Виведення черги у std::ostream (<<)
11	Клас CDeque – двобічна черга дійсних чисел. Пам’ять під елементи черги повинна виділятися динамічно. Реалізувати такі функції члени: • Отримання кількості елементів у черзі. • Знаходження максимального значення. • Знаходження мінімального значення. • Знаходження середнього арифметичного значення черги. • Очищення черги. • Перевірка, чи черга порожня. Перевантажити оператори: • Додавання справа (почленне додавання елементів до черги), оператор + • Віднімання зліва (почленне видалення елементів з черги), оператор - • Множення на скаляр. • Введення черги з std::istream (>>) • Виведення черги у std::ostream (<<)
12	Клас CBitArray – одновимірний масив бітів. Пам’ять під елементи масиву повинна виділятися динамічно. Кількість бітів задається у конструкторі. Для збереження бітів використовувати 32 або 64-бітні цілі числа. Реалізувати такі функції члени:

	• Встановлення значення біта з вказаним номером. • Отримання значення біта з вказаним номером. • Знаходження кількості бітів із вказаним значенням. Перевантажити оператори: • Побітове АБО (|) • Побітове І (&) • Побітове додавання за модулем 2 (^) • Побітовий зсув вліво (++). • Побітовий зсув вправо (--). • Введення масиву з std::istream (>>) • Виведення масиву у std::ostream (<<) Забезпечити можливість отримання значення біту [i] подібно до доступу до елементів звичайного одновимірного масиву.
13	Клас CTable – таблиця із колонками та рядками. Містить опис колонок та рядки із даними. Пам'ять під елементи масиву повинна виділятися динамічно. Реалізувати такі функції члени: • Кількість колонок • Назва колонки по індексу • Тип даних колонки по індексу • Кількість рядків • Значення заданої колонки у заданому рядку (по індексах) Перевантажити оператори: • Додавання (об'єднання рядків результатів на зразок UNION ALL). • Множення (декартовий добуток результатів на зразок CROSS JOIN) • Введення результату з std::istream (>>). • Виведення результату у std::ostream (<<).
14	Клас CEncryptor – клас для шифрування/дешифрування тексту простим зсувом символів по алфавіту. Пам'ять під текст повинна виділятись динамічно. Реалізувати такі функції члени: • Зашифрувати текст. • Розшифрувати текст. • Задавання кількості літер, на яку відбувається сзув. Перевантажити оператори: • Інкремент – шифрування тексту (++) • Декремент – розшифрування тексту (--) • Рівність (==) • Введення тексту з std::istream (>>). • Виведення тексту у std::ostream (<<).
15	Клас CString – стрічка символів (масив елементів символьного типу). Пам'ять під елементи масиву повинна виділятися динамічно. Реалізувати такі функції члени: • Видалення заданого символу із стрічки. • Заміна всіх входжень одного символу в стрічці заданим символом. • Сортування символів стрічки в алфавітному порядку у порядку зростання. • Вставлення однієї стрічки у іншу з заданої позиції. • Зміна розміру стрічки. Перевантажити оператори: • Додавання (конкатенація стрічок) • Порівняння стрічок(< та >) (більшою вважається та, яка йде першою в алфавітному порядку)

	• Рівність (==) • Введення стрічки з std::istream (>>) • Виведення стрічки у std::ostream (<<) Забезпечити можливість отримання значення елементу [i] подібно до доступу до елементів звичайного одновимірного масиву.

Контрольні питання

1. В яких види пам’яті можуть розміщуватись змінні (об’єкти) програми?
2. Які переваги та недоліки статичної (глобальної) пам’яті?
3. Які переваги та недоліки локальної (автоматичної) пам’яті?
4. Які переваги та недоліки локальної (динамічної) пам’яті?
5. Які оператори мови C++ використовуються для роботи із динамічною пам’яттю?
6. Які є різновиди оператора new?
7. В чому відмінність роботи операторів new/delete та відповідних функцій із бібліотеки C – malloc/free?
8. Для чого використовується коструктор копіювання та оператор копіювання? В яких випадках викликається конструктор, а коли оператор?
9. Для чого використовується коструктор переміщення та оператор переміщення?
10. В чому полягає ідіома RAII (Resource Acquisition Is Initialization)?