Міністерство освіти і науки України

Національний університет “Львівська політехніка”

Інститут комп’ютерних наук та інформаційних технологій

Кафедра програмного забезпечення



**Звіт**

Про виконання лабораторної роботи №5

**На тему:**

“Робота з динамічною пам’яттю”

**Лектор:**

доц. каф. ПЗ

Коротєєва Т.О.

**Виконав:**

ст. гр. ПЗ-18

Юшкевич А.І.

**Прийняв:**

асис. каф. ПЗ

Далявський В.С.

« … » … 2023 р.

∑ = \_\_\_\_\_\_\_\_\_\_\_\_

Львів – 2023

**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

1. Динамічна пам’ять в мові С++ 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.
2. 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.
3. 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:
4. 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.

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.

Варіант №12:

|  |  |
| --- | --- |
| 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 (<<). |

# Код програми

## Lab\_05\_OOP:

Table\_Header.h:

#pragma once

#include <iostream>

#include <vector>

#include <variant>

using namespace std;

class CTable {

size\_t number\_of\_columns;

vector<variant<int, double, char, string>>\* table{nullptr};

vector<string> columns\_name;

size\_t max\_size\_of\_column;

void AddColumn(vector<variant<int, double, char, string>> column);

template <class T>

void GetNewColumn(T foo, int current\_index, istream& is) {

for (int i = 0; i < table[current\_index].size(); i++) {

cout << "\nEnter " << i + 1 << "element: ";

is >> foo;

table[current\_index][i] = foo;

}

}

public:

CTable();

CTable(CTable& r\_other);

CTable(CTable&& rr\_other) noexcept;

~CTable();

CTable& operator= (const CTable& r\_other);

friend ostream& operator<< (ostream& os, const CTable& r\_other);

friend istream& operator>> (istream& os, CTable& r\_other);

CTable operator+(const CTable& r\_other);

CTable operator\*(const CTable& r\_other);

template <class T>

void CreateNewColumn(T arr[], size\_t number\_of\_elements, string column\_name) {

if (number\_of\_elements > max\_size\_of\_column)

max\_size\_of\_column = number\_of\_elements;

vector<variant<int, double, char, string>> new\_column(number\_of\_elements);

for (int i = 0; i < number\_of\_elements; i++) {

new\_column[i] = arr[i];

}

AddColumn(new\_column);

columns\_name.push\_back(column\_name);

}

template<class T>

T GetValueOfTable(int column, int row) const {

T result{ 0 };

if (const T\* value\_2 = get\_if<T>(&table[column][row])) {

result = \*value\_2;

}

return result;

}

void ShowTable() const;

int GetNumOfRows() const;

int GetNumOfColumns() const;

string GetNameOfColumn(int index) const;

string GetTypeOfColumn(int index) const;

};

### Table\_Header\_Unique.h:

#pragma once

#include <iostream>

#include <vector>

#include <variant>

using namespace std;

class CTable {

size\_t number\_of\_columns;

unique\_ptr<vector<variant<int, double, char, string>>[]> table{ nullptr };

vector<string> columns\_name;

size\_t max\_size\_of\_column;

void AddColumn(vector<variant<int, double, char, string>> column);

template <class T>

void GetNewColumn(T foo, int current\_index, istream& is) {

for (int i = 0; i < table[current\_index].size(); i++) {

cout << "\nEnter " << i + 1 << "element: ";

is >> foo;

table[current\_index][i] = foo;

}

}

public:

CTable();

CTable(CTable& r\_other);

CTable(CTable&& rr\_other) noexcept;

//~CTable();

CTable& operator= (const CTable& r\_other);

friend ostream& operator<< (ostream& os, const CTable& r\_other);

friend istream& operator>> (istream& os, CTable& r\_other);

CTable operator+(const CTable& r\_other);

CTable operator\*(const CTable& r\_other);

template <class T>

void CreateNewColumn(T arr[], size\_t number\_of\_elements, string column\_name) {

if (number\_of\_elements > max\_size\_of\_column)

max\_size\_of\_column = number\_of\_elements;

vector<variant<int, double, char, string>> new\_column(number\_of\_elements);

for (int i = 0; i < number\_of\_elements; i++) {

new\_column[i] = arr[i];

}

AddColumn(new\_column);

columns\_name.push\_back(column\_name);

}

template<class T>

T GetValueOfTable(int column, int row) const {

T result{ 0 };

if (const T\* value\_2 = get\_if<T>(&table[column][row])) {

result = \*value\_2;

}

return result;

}

void ShowTable() const;

int GetNumOfRows() const;

int GetNumOfColumns() const;

string GetNameOfColumn(int index) const;

string GetTypeOfColumn(int index) const;

};

Table\_Functions.cpp:

#include "Table\_Header.h"

CTable::CTable() : max\_size\_of\_column(0), number\_of\_columns(0) {};

CTable::CTable(CTable&& rr\_other) noexcept : max\_size\_of\_column(rr\_other.max\_size\_of\_column), number\_of\_columns(rr\_other.number\_of\_columns) {

this->table = rr\_other.table;

this->columns\_name = rr\_other.columns\_name;

rr\_other.table = nullptr;

};

CTable::CTable(CTable& r\_other) : max\_size\_of\_column(r\_other.max\_size\_of\_column), number\_of\_columns(r\_other.number\_of\_columns) {

vector<variant<int, double, char, string>>\* new\_table = new vector<variant<int, double, char, string>>[number\_of\_columns];

for (int i = 0; i < this->number\_of\_columns; i++) {

new\_table[i] = r\_other.table[i];

}

if (this->table != nullptr)

delete[] this->table;

this->columns\_name = r\_other.columns\_name;

this->table = new\_table;

}

CTable::~CTable() {

if (this->table != nullptr)

delete[] this->table;

this->table = nullptr;

}

ostream& operator<<(ostream& os, const CTable& c\_table) {

for (int i = 0; i < c\_table.number\_of\_columns; i++) {

os << c\_table.columns\_name[i] << "\t\t";

}

os << endl << endl;

for (int i = 0; i < c\_table.max\_size\_of\_column; i++) {

for (int j = 0; j < c\_table.number\_of\_columns; j++) {

if (i >= c\_table.table[j].size()) {

os << " \t\t";

continue;

}

if (auto value = get\_if<int>(&c\_table.table[j][0])) {

os << \*get\_if<int>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<double>(&c\_table.table[j][0])) {

os << \*get\_if<double>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<char>(&c\_table.table[j][0])) {

os << \*get\_if<char>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<string>(&c\_table.table[j][0])) {

os << \*get\_if<string>(&c\_table.table[j][i]) << "\t\t";

}

else break;

}

os << endl;

}

os << endl;

return os;

}

istream& operator>> (istream& is, CTable& c\_table) {

c\_table.number\_of\_columns = 0;

cout << "Enter number of columns: ";

is >> c\_table.number\_of\_columns;

cout << endl;

if (c\_table.number\_of\_columns <= 0) {

c\_table.number\_of\_columns = 0;

return is;

}

if (c\_table.table != nullptr)

delete[] c\_table.table;

c\_table.table = new vector<variant<int, double, char, string>>[c\_table.number\_of\_columns];

c\_table.columns\_name.clear();

c\_table.columns\_name.resize(c\_table.number\_of\_columns);

c\_table.max\_size\_of\_column = 0;

int num\_of\_elements{ 0 };

string type{ 0 };

for (int i = 0; i < c\_table.number\_of\_columns; i++) {

cout << "\n\nEnter number of elements in " << i + 1 << "column: ";

is >> num\_of\_elements;

cout << endl;

if (c\_table.max\_size\_of\_column < num\_of\_elements)

c\_table.max\_size\_of\_column = num\_of\_elements;

cout << "Enter type of elements in this column: ";

is >> type;

cout << endl;

c\_table.table[i].resize(num\_of\_elements);

if (type == "int" || type == "Int" || type == "INT" || type == "Int32") {

int foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "double" || type == "Double" || type == "DOUBLE") {

double foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "char" || type == "Char" || type == "CHAR") {

char foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "string" || type == "String" || type == "STRING") {

string foo{ "\0" };

c\_table.GetNewColumn(foo, i, is);

}

else {

cout << "\n\nWrong type, try again: \n\n";

i--;

continue;

}

c\_table.columns\_name[i] = type;

num\_of\_elements = 0;

}

cout << endl << endl;

}

CTable& CTable::operator= (const CTable& c\_table) {

this->max\_size\_of\_column = c\_table.max\_size\_of\_column;

this->number\_of\_columns = c\_table.number\_of\_columns;

vector<variant<int, double, char, string>>\* new\_table = new vector<variant<int, double, char, string>>[this->number\_of\_columns];

for (int i = 0; i < this->number\_of\_columns; i++) {

new\_table[i] = c\_table.table[i];

}

if (this->table != nullptr)

delete[] this->table;

this->columns\_name = c\_table.columns\_name;

this->table = new\_table;

return \*this;

}

CTable CTable::operator+(const CTable& r\_other) {

CTable c\_result\_table;

c\_result\_table.table = new vector<variant<int, double, char, string>>[number\_of\_columns];

int i = 0;

for (int j = 0; j < this->number\_of\_columns; j++, i++) {

c\_result\_table.AddColumn(this->table[j]);

c\_result\_table.columns\_name.push\_back(this->columns\_name[j]);

}

i--;

for (int j = 0; j < r\_other.number\_of\_columns; j++, i++) {

c\_result\_table.AddColumn(r\_other.table[j]);

c\_result\_table.columns\_name.push\_back(r\_other.columns\_name[j]);

}

return c\_result\_table;

}

CTable CTable::operator\*(const CTable& r\_other) {

CTable additional\_table;

for (int i = 0; i < r\_other.number\_of\_columns; i++) {

for (int j = 0; j < r\_other.table[i].size(); j++) {

vector<variant<int, double, char, string>> new\_column;

for (int k = 0; k < this->max\_size\_of\_column; k++) {

new\_column.push\_back(r\_other.table[i][j]);

}

additional\_table.AddColumn(new\_column);

additional\_table.columns\_name.push\_back(r\_other.columns\_name[i]);

new\_column.clear();

}

}

return \*this + additional\_table;

}

void CTable::ShowTable() const {

for (int i = 0; i < number\_of\_columns; i++) {

cout << columns\_name[i] << "\t\t";

}

cout << endl << endl;

for (int i = 0; i < max\_size\_of\_column; i++) {

for (int j = 0; j < number\_of\_columns; j++) {

if (i >= table[j].size()) {

cout << " \t\t";

continue;

}

if (auto value = get\_if<int>(&table[j][0])) {

cout << \*get\_if<int>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<double>(&table[j][0])) {

cout << \*get\_if<double>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<char>(&table[j][0])) {

cout << \*get\_if<char>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<string>(&table[j][0])) {

cout << \*get\_if<string>(&table[j][i]) << "\t\t";

}

else break;

}

cout << endl;

}

}

int CTable::GetNumOfColumns() const{

return number\_of\_columns;

}

int CTable::GetNumOfRows() const{

return max\_size\_of\_column;

}

string CTable::GetNameOfColumn(int index) const {

return columns\_name[index];

}

string CTable::GetTypeOfColumn(int index) const {

string result = "unknown";

if (auto value = get\_if<int>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<double>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<char>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<string>(&table[index][0])) {

result = typeid(\*value).name();

}

return result;

}

void CTable::AddColumn(vector<variant<int, double, char, string>> column) {

number\_of\_columns++;

vector<variant<int, double, char, string>>\* new\_table = new vector<variant<int, double, char, string>>[number\_of\_columns];

int i = 0;

for (; i < number\_of\_columns - 1; i++) {

new\_table[i] = table[i];

}

new\_table[i] = column;

if (column.size() > max\_size\_of\_column)

max\_size\_of\_column = column.size();

if (this->table != nullptr)

delete[] table;

table = new\_table;

}

Table\_Functions\_Unique.cpp:

#include "Table\_Header\_Unique.h"

CTable::CTable() : max\_size\_of\_column(0), number\_of\_columns(0) {};

CTable::CTable(CTable&& rr\_other) noexcept : max\_size\_of\_column(rr\_other.max\_size\_of\_column), number\_of\_columns(rr\_other.number\_of\_columns) {

this->table = unique\_ptr<vector<variant<int, double, char, string>>[]>(new vector<variant<int, double, char, string>>[rr\_other.number\_of\_columns]);

this->columns\_name = rr\_other.columns\_name;

rr\_other.table = nullptr;

};

CTable::CTable(CTable& r\_other) : max\_size\_of\_column(r\_other.max\_size\_of\_column), number\_of\_columns(r\_other.number\_of\_columns) {

unique\_ptr<vector<variant<int, double, char, string>>[]> new\_table(new vector<variant<int, double, char, string>>[r\_other.number\_of\_columns]);

for (int i = 0; i < this->number\_of\_columns; i++) {

new\_table[i] = r\_other.table[i];

}

if (this->table != nullptr)

this->table.reset();

this->columns\_name = r\_other.columns\_name;

this->table.swap(new\_table);

}

//CTable::~CTable() {

// if (this->table != nullptr)

// this->table.reset();

// this->table = nullptr;

//

//}

ostream& operator<<(ostream& os, const CTable& c\_table) {

for (int i = 0; i < c\_table.number\_of\_columns; i++) {

os << c\_table.columns\_name[i] << "\t\t";

}

os << endl << endl;

for (int i = 0; i < c\_table.max\_size\_of\_column; i++) {

for (int j = 0; j < c\_table.number\_of\_columns; j++) {

if (i >= c\_table.table[j].size()) {

os << " \t\t";

continue;

}

if (auto value = get\_if<int>(&c\_table.table[j][0])) {

os << \*get\_if<int>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<double>(&c\_table.table[j][0])) {

os << \*get\_if<double>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<char>(&c\_table.table[j][0])) {

os << \*get\_if<char>(&c\_table.table[j][i]) << "\t\t";

}

else if (auto value = get\_if<string>(&c\_table.table[j][0])) {

os << \*get\_if<string>(&c\_table.table[j][i]) << "\t\t";

}

else break;

}

os << endl;

}

os << endl;

return os;

}

istream& operator>> (istream& is, CTable& c\_table) {

c\_table.number\_of\_columns = 0;

cout << "Enter number of columns: ";

is >> c\_table.number\_of\_columns;

cout << endl;

if (c\_table.number\_of\_columns <= 0) {

c\_table.number\_of\_columns = 0;

return is;

}

if (c\_table.table != nullptr)

c\_table.table.reset();

c\_table.table = unique\_ptr<vector<variant<int, double, char, string>>[]>(new vector<variant<int, double, char, string>>[c\_table.number\_of\_columns]);

c\_table.columns\_name.clear();

c\_table.columns\_name.resize(c\_table.number\_of\_columns);

c\_table.max\_size\_of\_column = 0;

int num\_of\_elements{ 0 };

string type{ 0 };

for (int i = 0; i < c\_table.number\_of\_columns; i++) {

cout << "\n\nEnter number of elements in " << i + 1 << "column: ";

is >> num\_of\_elements;

cout << endl;

if (c\_table.max\_size\_of\_column < num\_of\_elements)

c\_table.max\_size\_of\_column = num\_of\_elements;

cout << "Enter type of elements in this column: ";

is >> type;

cout << endl;

c\_table.table[i].resize(num\_of\_elements);

if (type == "int" || type == "Int" || type == "INT" || type == "Int32") {

int foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "double" || type == "Double" || type == "DOUBLE") {

double foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "char" || type == "Char" || type == "CHAR") {

char foo{ 0 };

c\_table.GetNewColumn(foo, i, is);

}

else if (type == "string" || type == "String" || type == "STRING") {

string foo{ "\0" };

c\_table.GetNewColumn(foo, i, is);

}

else {

cout << "\n\nWrong type, try again: \n\n";

i--;

continue;

}

c\_table.columns\_name[i] = type;

num\_of\_elements = 0;

}

cout << endl << endl;

}

CTable& CTable::operator= (const CTable& c\_table) {

this->max\_size\_of\_column = c\_table.max\_size\_of\_column;

this->number\_of\_columns = c\_table.number\_of\_columns;

unique\_ptr<vector<variant<int, double, char, string>>[]> new\_table(new vector<variant<int, double, char, string>>[number\_of\_columns]);

for (int i = 0; i < this->number\_of\_columns; i++) {

new\_table[i] = c\_table.table[i];

}

if (this->table != nullptr)

this->table.reset();

this->columns\_name = c\_table.columns\_name;

this->table.swap(new\_table);

return \*this;

}

CTable CTable::operator+(const CTable& r\_other) {

CTable c\_result\_table;

c\_result\_table.table = unique\_ptr<vector<variant<int, double, char, string>>[]>(new vector<variant<int, double, char, string>>[number\_of\_columns]);

int i = 0;

for (int j = 0; j < this->number\_of\_columns; j++, i++) {

c\_result\_table.AddColumn(this->table[j]);

c\_result\_table.columns\_name.push\_back(this->columns\_name[j]);

}

i--;

for (int j = 0; j < r\_other.number\_of\_columns; j++, i++) {

c\_result\_table.AddColumn(r\_other.table[j]);

c\_result\_table.columns\_name.push\_back(r\_other.columns\_name[j]);

}

return c\_result\_table;

}

CTable CTable::operator\*(const CTable& r\_other) {

CTable additional\_table;

for (int i = 0; i < r\_other.number\_of\_columns; i++) {

for (int j = 0; j < r\_other.table[i].size(); j++) {

vector<variant<int, double, char, string>> new\_column;

for (int k = 0; k < this->max\_size\_of\_column; k++) {

new\_column.push\_back(r\_other.table[i][j]);

}

additional\_table.AddColumn(new\_column);

additional\_table.columns\_name.push\_back(r\_other.columns\_name[i]);

new\_column.clear();

}

}

return \*this + additional\_table;

}

void CTable::ShowTable() const {

for (int i = 0; i < number\_of\_columns; i++) {

cout << columns\_name[i] << "\t\t";

}

cout << endl << endl;

for (int i = 0; i < max\_size\_of\_column; i++) {

for (int j = 0; j < number\_of\_columns; j++) {

if (i >= table[j].size()) {

cout << " \t\t";

continue;

}

if (auto value = get\_if<int>(&table[j][0])) {

cout << \*get\_if<int>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<double>(&table[j][0])) {

cout << \*get\_if<double>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<char>(&table[j][0])) {

cout << \*get\_if<char>(&table[j][i]) << "\t\t";

}

else if (auto value = get\_if<string>(&table[j][0])) {

cout << \*get\_if<string>(&table[j][i]) << "\t\t";

}

else break;

}

cout << endl;

}

}

int CTable::GetNumOfColumns() const {

return number\_of\_columns;

}

int CTable::GetNumOfRows() const {

return max\_size\_of\_column;

}

string CTable::GetNameOfColumn(int index) const {

return columns\_name[index];

}

string CTable::GetTypeOfColumn(int index) const {

string result = "unknown";

if (auto value = get\_if<int>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<double>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<char>(&table[index][0])) {

result = typeid(\*value).name();

}

else if (auto value = get\_if<string>(&table[index][0])) {

result = typeid(\*value).name();

}

return result;

}

void CTable::AddColumn(vector<variant<int, double, char, string>> column) {

number\_of\_columns++;

unique\_ptr<vector<variant<int, double, char, string>>[]> new\_table(new vector<variant<int, double, char, string>>[number\_of\_columns]);

int i = 0;

for (; i < number\_of\_columns - 1; i++) {

new\_table[i] = table[i];

}

new\_table[i] = column;

if (column.size() > max\_size\_of\_column)

max\_size\_of\_column = column.size();

if (this->table != nullptr)

table.reset();

table.swap(new\_table);

}

### pch.h:

#pragma once

#include "gtest/gtest.h"

#include "../Lab\_05\_OOP/Table\_Header.h"

#include "../Lab\_05\_OOP/Table\_Header\_Unique.h"

### test.cpp:

#include "pch.h"

TEST(CTableTest, CreateNewColumn) {

CTable c\_table;

int arr[] = { 1, 2, 3, 4, 5 };

size\_t number\_of\_elements = 5;

string column\_name = "test\_column";

c\_table.CreateNewColumn(arr, number\_of\_elements, column\_name);

EXPECT\_EQ(c\_table.GetNumOfColumns(), 1);

EXPECT\_EQ(c\_table.GetNumOfRows(), 5);

EXPECT\_EQ(c\_table.GetNameOfColumn(0), column\_name);

EXPECT\_EQ(c\_table.GetValueOfTable<int>(0, 2), 3);

}

TEST(CTableTest, OperatorMultiplication) {

CTable c\_table1, c\_table2, c\_table3;

int arr1[] = { 1, 2, 3, 4, 5 };

int arr2[] = { 6, 7, 8, 9, 10 };

int arr3[] = { 11, 12, 13, 14, 15 };

c\_table1.CreateNewColumn(arr1, 5, "column1");

c\_table1.CreateNewColumn(arr2, 5, "column2");

c\_table2.CreateNewColumn(arr2, 5, "column2");

c\_table2.CreateNewColumn(arr3, 5, "column3");

c\_table3 = c\_table1 \* c\_table2;

EXPECT\_EQ(c\_table3.GetNumOfColumns(), 12);

EXPECT\_EQ(c\_table3.GetNumOfRows(), 5);

EXPECT\_EQ(c\_table3.GetNameOfColumn(0), "column1");

EXPECT\_EQ(c\_table3.GetNameOfColumn(1), "column2");

EXPECT\_EQ(c\_table3.GetNameOfColumn(2), "column2");

EXPECT\_EQ(c\_table3.GetNameOfColumn(3), "column2");

EXPECT\_EQ(c\_table3.GetNameOfColumn(4), "column2");

EXPECT\_EQ(c\_table3.GetNameOfColumn(5), "column2");

EXPECT\_EQ(c\_table3.GetNameOfColumn(6), "column2");

EXPECT\_EQ(c\_table3.GetNameOfColumn(7), "column3");

EXPECT\_EQ(c\_table3.GetNameOfColumn(8), "column3");

EXPECT\_EQ(c\_table3.GetNameOfColumn(9), "column3");

EXPECT\_EQ(c\_table3.GetNameOfColumn(10), "column3");

EXPECT\_EQ(c\_table3.GetNameOfColumn(11), "column3");

EXPECT\_EQ(c\_table3.GetValueOfTable<int>(0, 2), 3);

EXPECT\_EQ(c\_table3.GetValueOfTable<int>(1, 2), 8);

EXPECT\_EQ(c\_table3.GetValueOfTable<int>(2, 2), 6);

}

TEST(CTableTest, OverloadCout) {

CTable t1;

int arr[] = { 1, 2, 3, 4 };

t1.CreateNewColumn(arr, 4, "Column 1");

double arr2[] = { 2.5, 3.6, 7.8, 9.2 };

t1.CreateNewColumn(arr2, 4, "Column 2");

std::ostringstream oss;

oss << t1;

std::string expected\_output = "Column 1\t\tColumn 2\t\t\n\n1\t\t2.5\t\t\n2\t\t3.6\t\t\n3\t\t7.8\t\t\n4\t\t9.2\t\t\n\n";

EXPECT\_EQ(oss.str(), expected\_output);

}

TEST(CTableTest, OperatorAddition) {

CTable t1;

int arr[] = { 1, 2, 3, 4 };

t1.CreateNewColumn(arr, 4, "Column 1");

double arr2[] = { 2.5, 3.6, 7.8, 9.2 };

t1.CreateNewColumn(arr2, 4, "Column 2");

CTable t2;

int arr3[] = { 1, 2, 3, 4 };

t2.CreateNewColumn(arr3, 4, "Column 1");

double arr4[] = { 2.5, 3.6, 7.8, 9.2 };

t2.CreateNewColumn(arr4, 4, "Column 2");

CTable t3 = t1 + t2;

EXPECT\_EQ(t3.GetValueOfTable<int>(0, 2), 3);

EXPECT\_DOUBLE\_EQ(t3.GetValueOfTable<double>(1, 2), 7.8);

EXPECT\_EQ(t3.GetValueOfTable<int>(2, 3), 4);

EXPECT\_DOUBLE\_EQ(t3.GetValueOfTable<double>(3, 1), 3.6);

}

# Результат виконання

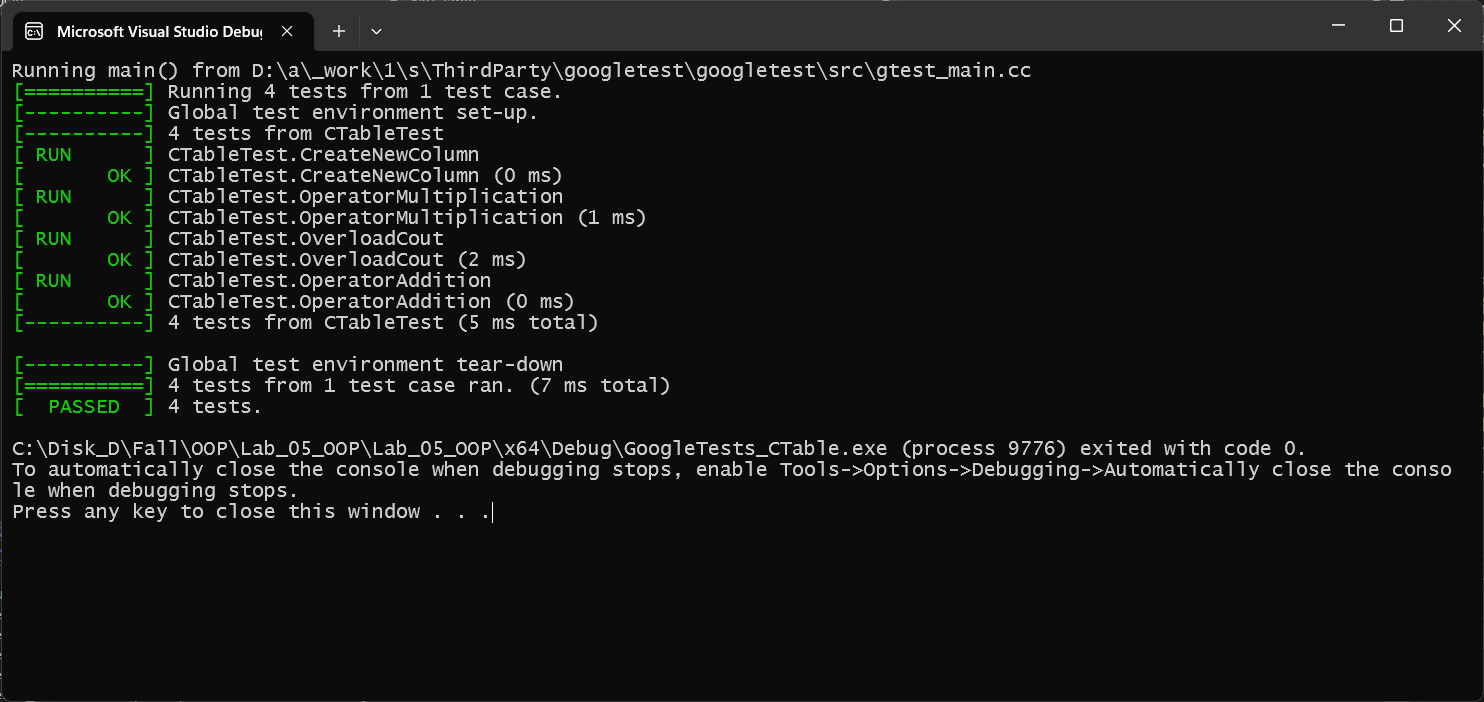


Рис. 1. Виконання тестів для функцій класу зі звичайним виділенням пам’яті.

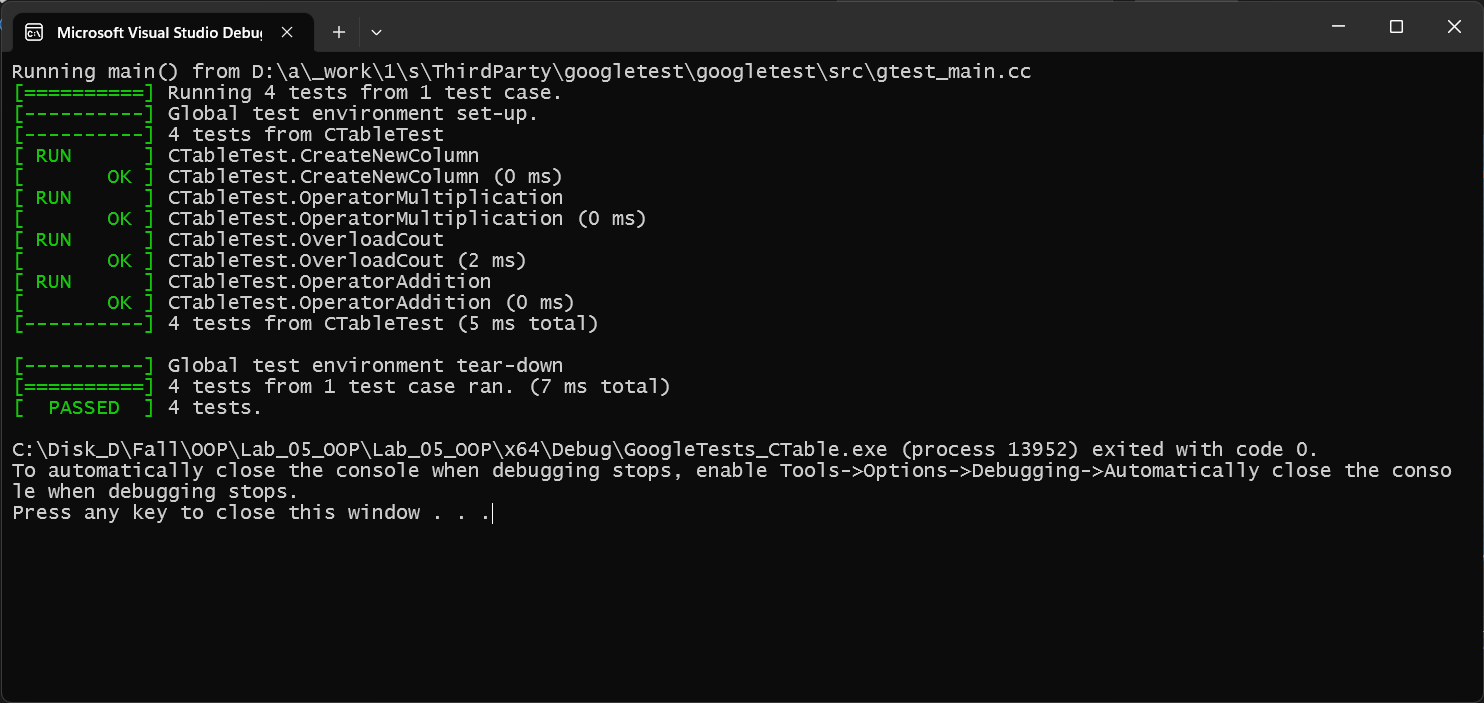


Рис. 2. Виконання тестів для функцій класу із виділенням пам’яті через unique\_ptr<T>.

# Висновок

Під час виконання лабораторної роботи я навчився виділяти місце під об’єкти динамічно, створювати та використовувати конструктор копіювання та переміщення, перевантажувати оператор присвоєння та переміщення. Ознайомився з принципами створення та функціонування деструкторів.