ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №2

за курсом «Об’єктно-орієнтоване програмування»

студента групи ПЗ-24у-1

Кондрачука Олександра Вадимовича

кафедра математичного забезпечення ЕОМ, ДНУ ім. О. Гончара

2024/2025 н.р.

Тема: Оперативна пам’ять: стратегії динамічного розподілення.

1. Постановка задачі

Розробити об’єктно-орієнтовану бібліотеку для роботи зі структурами даних за однією з нижченаведених тем у відповідності з нижченаведеними вимогами. Властивості та методи для класів розробити у відповідності з відомими визначеннями відповідних структур даних. Скласти тести для перевірки працездатності бібліотеки. Скласти програму, що демонструє можливості розробленої бібліотеки.

Загальні вимогу: в незалежності від індивідуального варіанта повинні бути реалізовані наступні можливості.

1. Реалізація методів ініціалізації (конструктор по замовчуванню та конструктор з параметрами), копіювання (конструктор копіювання).

2. Індексації (перевантаження []).

3. Присвоювання (перевантаження =).

4. Перевантаження потокового введення/виведення таким чином, що повинно працювати введення як з файлу, так і з консолі, виведення в файл та консоль. Демонструвати, як стан об’єкту може бути збережений на диск та відновлений зі збереженого стану.

5. Застосування вказаної структури даних для розв’язання типової задачі

В ході демонстрації роботи, програма дозволяє користувачеві в діалоговому режимі (за допомогою меню) виконувати операції над відповідними структурами даних. Використання структур даних з вбудованої бібліотеки шаблонів С++ в даній роботі не дозволяється.

Індивідуальне завдання: Оперативна пам’ять: стратегії динамічного розподілення. Розробка власного менеджера пам’яті. Виділення пам’яті за принципом: перший блок, що підходить за розміром; найбільш підходящий (залишок мінімального розміру); найменш підходящий (залишок максимального розміру); реалізувати стратегію відновлення: якщо два вільних блока пам’яті знаходяться рядом, то вони об’єднуються в один блок. Діалогове керування: виділити пам’ять, звільнити пам’ять, візуалізувати стан пам’яті (наприклад, у вигляді 00000011111000000000111, де 111 – зайняті блоки).

1. Опис розв’язку

Проект розподілений на декілька модулів:

* Heap.h. Являє собою набір публічних методів для використання менеджера у коді програми;
* ChunksContainer.h. Являє собою контейнер з усіма частинами пам’яті та маніпулює ними;
* HeapChunk.h. Є структурної для зберігання даних про частину пам’яті.
* main.cpp. Містить код для взаємодії з користувачем через меню та демонстрацію роботи основних функцій;

Реалізовано конструктори:

* Конструктор за замовчуванням;
* Конструктор із параметром для виділення певного об’єму пам’яті;
* Конструктор копіювання.

Реалізовано основні операції:

* Виділення певної кількості пам’яті.
* Виділення пам’яті під конкретний тип.
* Вивільнення памяті.

Перевантаження операторів:

* Оператор = для коректного присвоєння одного менеджера іншому.
* Оператори << і >> для роботи з потоком введення/виведення (з файла та консолі).
* Оператор [ ] для доступу до частини пам’яті по її індексу.

У файлі main.cpp реалізовано меню, яке дозволяє користувачу:

* Виділити певну кількість байтів.
* Вивільнити один з створених частин.
* Вивести поточний стан пам’яті.
* Зберегти поточний стан пам’яті у файл.
* Виділити певну кількість байтів, що зазначені у файлі.

1. Вихідний текст програми розв’язку задачі

Лістинг 2.1 –Текст програми Heap.h

#pragma once

#include <cstdlib>

#include "ChunksContainer.h"

#define MemorySize 1024

class Heap

{

private:

ChunksContainer \_chunks;

void\* allocatedMemory;

static void\* InternalAlloc(size\_t size)

{

return malloc(size);

}

public:

Heap(const Heap& other)

{

\_chunks = other.\_chunks;

}

Heap() : Heap(MemorySize)

{

}

Heap(size\_t memorySize)

{

size\_t size = ChunksContainer::GetActualMemorySize(memorySize);

allocatedMemory = InternalAlloc(size);

\_chunks = ChunksContainer(size, allocatedMemory);

}

~Heap()

{

free(allocatedMemory);

}

void\* Allocate(const size\_t size)

{

return \_chunks.GetFreeChunk(size);

}

template<typename T>

T\* Allocate()

{

return static\_cast<T\*>(Allocate(sizeof(T)));

}

void Free(const void\* ptr)

{

\_chunks.FreeChunk(ptr);

}

void Debug(std::ostream& out) const

{

\_chunks.DebugMemory(out);

}

Heap& operator=(const Heap& other)

{

\_chunks = other.\_chunks;

return \*this;

}

HeapChunk\* operator[](const int index) const

{

return \_chunks.GetByIndex(index);

}

friend std::ostream& operator<<(std::ostream& out, const Heap& heap)

{

heap.Debug(out);

return out;

}

friend std::istream& operator>>(std::istream& in, Heap& heap)

{

int size;

in >> size;

heap.Allocate(size);

return in;

}

};

Лістинг 2.2 –Текст програми ChunksContainer.h

#pragma once

#include <iostream>

#include "HeapChunk.h"

class ChunksContainer

{

private:

HeapChunk\* \_start;

size\_t \_memorySize;

HeapChunk\* CreateChunk(size\_t size, void\* memory)

{

HeapChunk\* chunk = static\_cast<HeapChunk\*>(memory);

chunk->InUse = false;

chunk->Size = size;

chunk->Next = nullptr;

chunk->SetMemory(memory);

return chunk;

}

bool CanCreateChunk(size\_t memorySize)

{

return memorySize > HeapChunk::GetFunctionalMemoryUsage();

}

void CreateFirstChunk(size\_t size, void\* memory)

{

HeapChunk\* chunk = CreateChunk(size, memory);

\_memorySize = chunk->Size;

\_start = chunk;

}

void AddChunkAfter(HeapChunk\* newChunk, HeapChunk\* oldChunk)

{

newChunk->Next = oldChunk->Next;

oldChunk->Next = newChunk;

}

void TruncateChunk(HeapChunk\* chunk, size\_t neededSize)

{

if (chunk->GetUnoccupiedMemorySize() == neededSize)

{

chunk->InUse = true;

return;

}

size\_t remainingSize = chunk->GetUnoccupiedMemorySize() - neededSize;

if (!CanCreateChunk(remainingSize))

{

chunk->InUse = true;

return;

}

void\* remainingMemory = chunk->MemoryRegion + GetActualMemorySize(neededSize);

HeapChunk\* unusedChunk = CreateChunk(remainingSize, remainingMemory);

AddChunkAfter(unusedChunk, chunk);

chunk->Size = GetActualMemorySize(neededSize);

chunk->InUse = true;

}

void TryCoalesceNearbyChunks(HeapChunk\* chunk)

{

HeapChunk\* previous = FindByNext(chunk);

HeapChunk\* next = chunk->Next;

TryCoalesceChunks(chunk, next);

TryCoalesceChunks(previous, chunk);

}

void TryCoalesceChunks(HeapChunk\* previous, HeapChunk\* next)

{

if (previous == nullptr || previous->InUse)

{

return;

}

if (next == nullptr || next->InUse)

{

return;

}

previous->Size += next->Size;

previous->Next = next;

previous->Next = next->Next;

}

HeapChunk\* FindByMemoryRegion(const void\* memoryRegion) const

{

HeapChunk\* current = \_start;

while (current != nullptr)

{

if (current->GetUnoccupiedMemory() == memoryRegion)

{

return current;

}

current = current->Next;

}

return nullptr;

}

HeapChunk\* FindByNext(const HeapChunk\* chunk) const

{

HeapChunk\* current = \_start;

while (current != nullptr)

{

if (current->Next == chunk)

{

return current;

}

current = current->Next;

}

return nullptr;

}

HeapChunk\* FindSmallest(size\_t size) const

{

HeapChunk\* current = \_start;

HeapChunk\* min = nullptr;

while (current != nullptr)

{

if (current->InUse)

{

current = current->Next;

continue;

}

if (current->GetUnoccupiedMemorySize() < size)

{

current = current->Next;

continue;

}

if (min == nullptr || current->Size < min->Size)

{

min = current;

}

current = current->Next;

}

return min;

}

public:

ChunksContainer() = default;

ChunksContainer(size\_t size, void\* memory)

{

CreateFirstChunk(size, memory);

}

static size\_t GetActualMemorySize(size\_t size)

{

return HeapChunk::GetFunctionalMemoryUsage() + size;

}

HeapChunk\* GetByIndex(int index) const

{

HeapChunk\* current = \_start;

int i = 0;

while (current != nullptr)

{

if (i == index)

{

return current;

}

i++;

current = current->Next;

}

return nullptr;

}

void\* GetFreeChunk(size\_t size)

{

HeapChunk\* chunk = FindSmallest(size);

if (chunk == nullptr)

{

throw std::exception("No free chunks");

}

TruncateChunk(chunk, size);

return chunk->GetUnoccupiedMemory();

}

void FreeChunk(const void\* memory)

{

HeapChunk\* chunk = FindByMemoryRegion(memory);

if (chunk == nullptr)

{

throw std::exception("No such memory exists");

}

if (!chunk->InUse)

{

throw std::exception("This memory region is not occupied");

}

chunk->InUse = false;

TryCoalesceNearbyChunks(chunk);

}

void DebugMemory(std::ostream& out) const

{

HeapChunk\* current = \_start;

int i = 0;

while (current != nullptr)

{

out << "Item: " << i << '\n';

out << "In Use: " << (current->InUse ? "true" : "false") << '\n';

out << "Size: " << current->Size << '\n';

out << "Unoccupied size: " << current->GetUnoccupiedMemorySize() << '\n';

out << "Functional size: " << HeapChunk::GetFunctionalMemoryUsage() << '\n';

out << '\n';

i++;

current = current->Next;

}

out << "\n";

}

};

Лістинг 2.3 –Текст програми HeapChunk.h

#pragma once

struct HeapChunk

{

size\_t Size;

bool InUse;

char\* MemoryRegion;

HeapChunk\* Next;

char\* GetUnoccupiedMemory() const

{

return MemoryRegion + sizeof(HeapChunk);

}

size\_t GetUnoccupiedMemorySize() const

{

return Size - sizeof(HeapChunk);

}

void SetMemory(void\* memory)

{

MemoryRegion = static\_cast<char\*>(memory);

}

static size\_t GetFunctionalMemoryUsage()

{

return sizeof(HeapChunk);

}

};

Лістинг 2.4 –Текст програми main.cpp

#include <fstream>

#include <sstream>

#include <string>

#include <vector>

#include "Heap.h"

void load\_from\_file(const char\* filename, Heap& heap);

void save\_to\_file(const char\* filename, const Heap& heap);

void run();

void allocate();

void free();

Heap heap\_(1024);

std::vector<std::pair<int, void\*>> allocatedMemory;

int main()

{

run();

}

void run()

{

while (true)

{

std::cout << "1 - Allocate memory\n";

std::cout << "2 - Free memory\n";

std::cout << "3 - Save to file\n";

std::cout << "4 - Load from file\n";

std::cout << "5 - Debug\n";

int num;

std::cin >> num;

std::cout << "\n";

switch (num)

{

case 1:

allocate();

break;

case 2:

free();

break;

case 3:

save\_to\_file("test1.txt", heap\_);

break;

case 4:

load\_from\_file("test1.txt", heap\_);

break;

case 5:

heap\_.Debug(std::cout);

break;

default:

break;

}

std::cout << "\n\n";

}

}

void allocate()

{

std::cout << "Amount to allocate: ";

int amount;

std::cin >> amount;

void\* ptr = heap\_.Allocate(amount);

allocatedMemory.emplace\_back(amount, ptr);

}

void free()

{

std::cout << "Select memory region\n";

for (int i = 0; i < allocatedMemory.size(); i++)

{

std::cout << i << ": " << allocatedMemory[i].first << "\n";

}

int index;

std::cin >> index;

heap\_.Free(allocatedMemory[index].second);

allocatedMemory.erase(allocatedMemory.begin() + index);

}

void save\_to\_file(const char\* filename, const Heap& heap)

{

std::ofstream file(filename);

if (file.is\_open())

{

file << heap;

file.close();

}

}

void load\_from\_file(const char\* filename, Heap& heap)

{

std::ifstream file(filename);

if (file.is\_open())

{

std::string line;

while (std::getline(file, line))

{

std::istringstream iss(line);

iss >> heap;

}

file.close();

}

else

{

std::cerr << "Error: File not found." << '\n';

}

}

1. Опис інтерфейсу
2. Запустити файл Lab\_2\_OOP.exe через провідник або командний рядок.
3. Після запуску перед користувачем з’являєтся меню, рисунок 2.1.

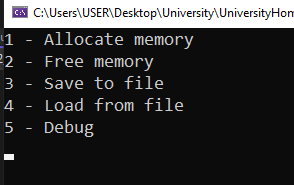


Рисунок 2.1 – Меню

1. Обравши 1 пункт користувач може виділити певну кількість пам’яті, рисунок 2.2.

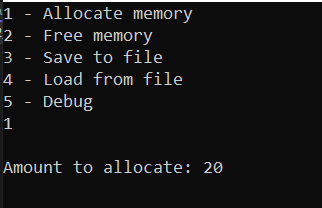


Рисунок 2.2 – Виділення пам’яті

1. Обравши 2 пункт користувач може вивільнити до цього виділену пам’ять, рисунок 2.3.

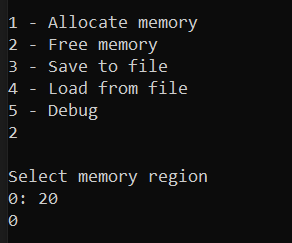


Рисунок 2.3 – Вивільнення пам’яті

1. Обравши пункт 5 виводимо поточний стан пам’яті, рисунок 2.4.

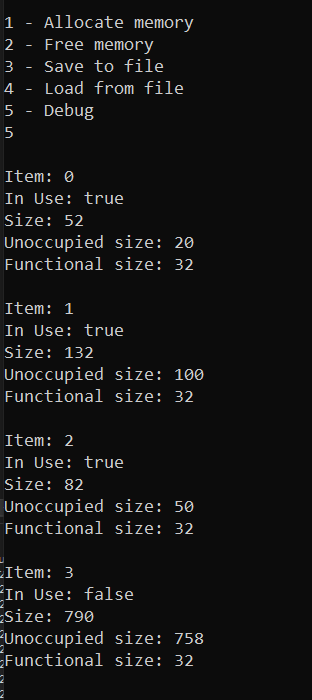


Рисунок 2.4 – Вивід поточного стану пам’яті

1. Обравши пункт 3 поточний стан у фай, рисунок 2.5.

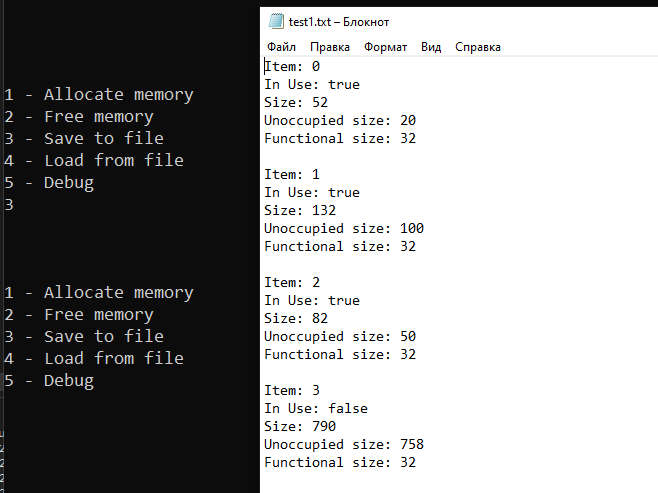


Рисунок 2.5 – Вивід поточного стану до файлу

1. Обравши пункт 4 виділяємо байти пам’яті, що зазначені у файлі, рисунок 2.6.

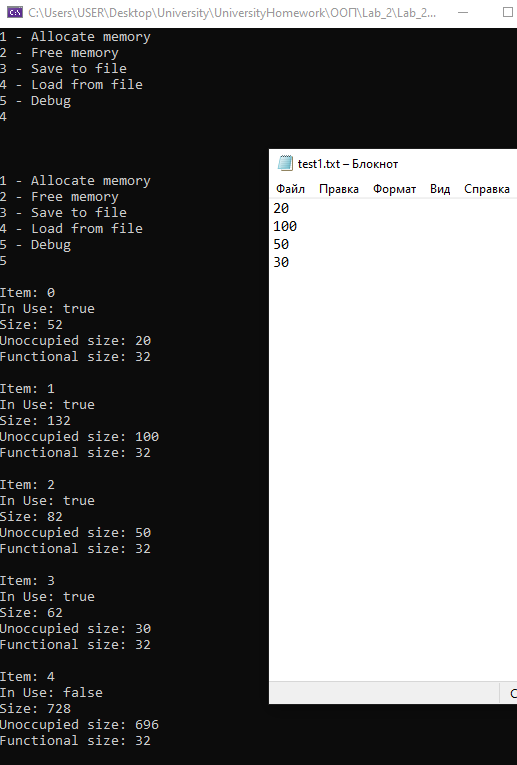
 

Рисунок 2.6 – Завантаження з файлу