**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе № 2.1**

**по дисциплине «Операционные системы»**

Тема: **«Управление памятью»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Баймухамедов Р. Р. |  |
| Преподаватель | Тимофеев А. В. |  |

Санкт-Петербург

2025

**Цель работы**

Исследовать механизмы управления виртуальной

памятью Win32.

**Задание**

Постановка задачи и описание решения

Для выполнения данной лабораторной работы необходимо разработать консольное приложение, которое позволяет:

* Получить информацию о вычислительной системе.
* Определить текущее состояние виртуальной памяти.
* Проанализировать статус заданного участка памяти.
* Выполнить резервирование и выделение физической памяти в разных режимах.
* Записывать данные по заданным адресам.
* Управлять защитой доступа к памяти.

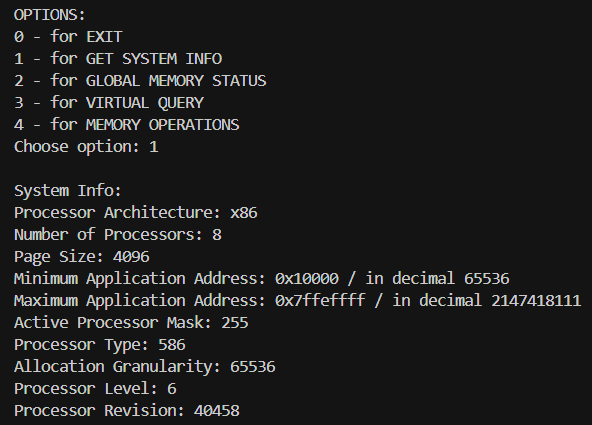
В ходе выполнения работы было исследовано управление виртуальной памятью в Windows. Данный механизм предоставляет широкий набор возможностей, включая резервирование областей памяти, выделение физической памяти, изменение прав доступа и мониторинг состояния системы.

Одним из ключевых аспектов является возможность гибкого управления выделением памяти: можно заранее зарезервировать область и затем передавать ей физические ресурсы по мере необходимости. Это позволяет оптимизировать использование памяти и повышает предсказуемость работы приложения.

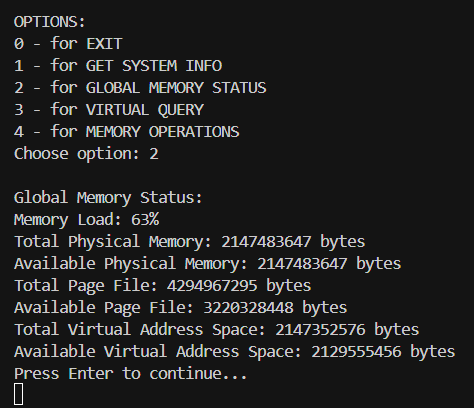
Также было исследовано получение системной информации, связанной с памятью, что позволяет анализировать доступные ресурсы и принимать более обоснованные решения при выделении памяти.

В целом, работа с виртуальной памятью через Win32 API даёт полный контроль над её использованием, что может быть полезно в различных сценариях, включая оптимизацию работы программ и обеспечение безопасности.

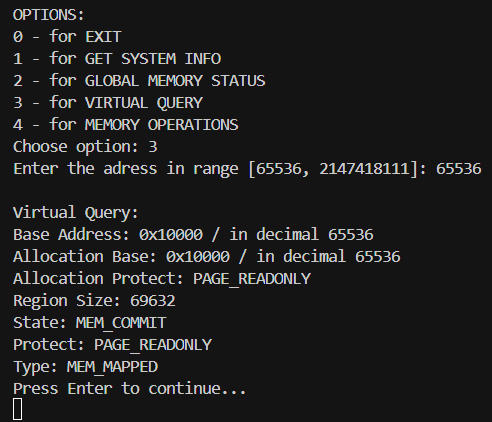
**Изображение работоспособности программы**



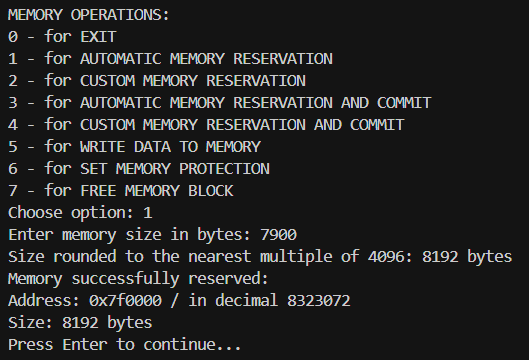
Изображение – GET SYSTEM INFO



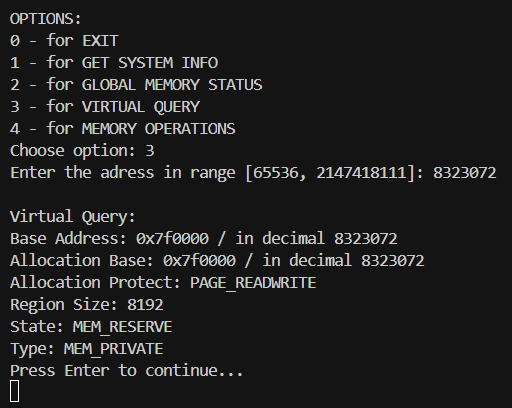
Изображение 2 – GLOBAL MEMORY STATUS



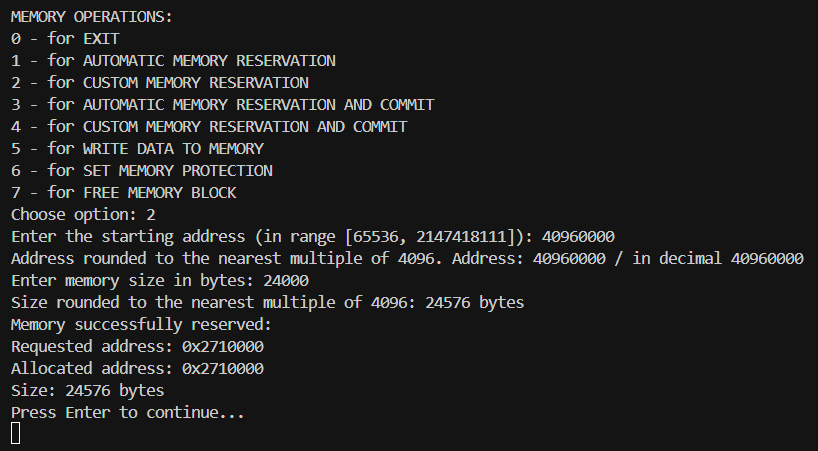
Изображение 3 – VIRTUAL QUERY



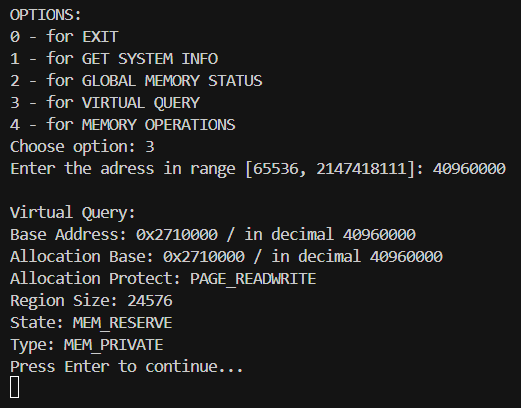
Изображение 4.1 – Автоматическое резервирование памяти



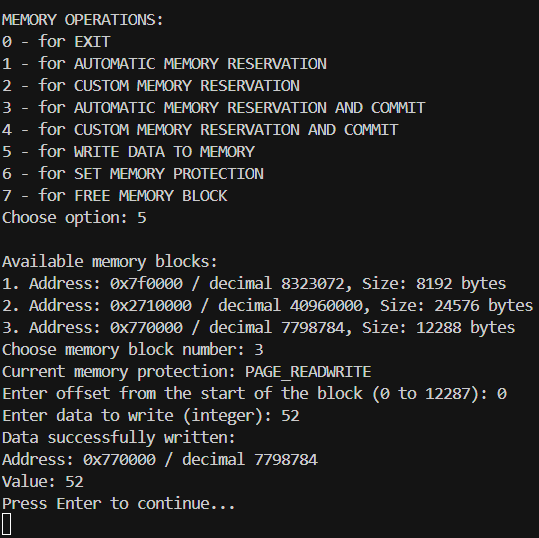
Изображение 4.2 – Проверка адреса памяти



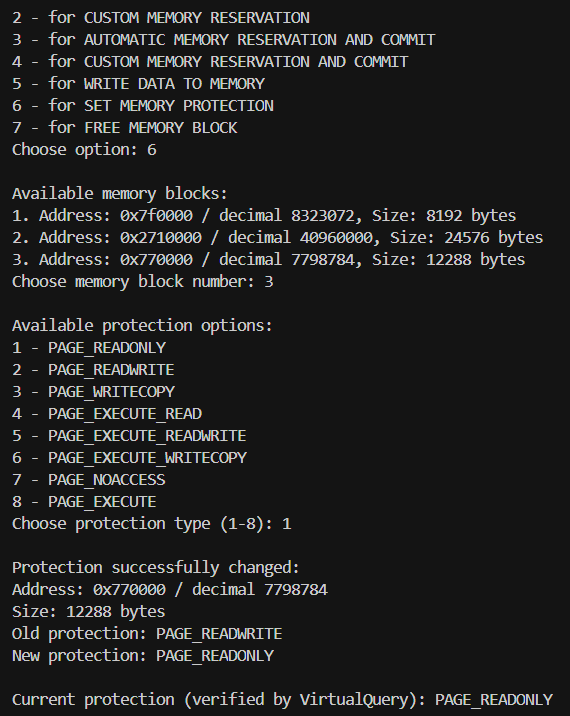
Изображение 5.1 – Резервирование адреса, вводимого пользователем



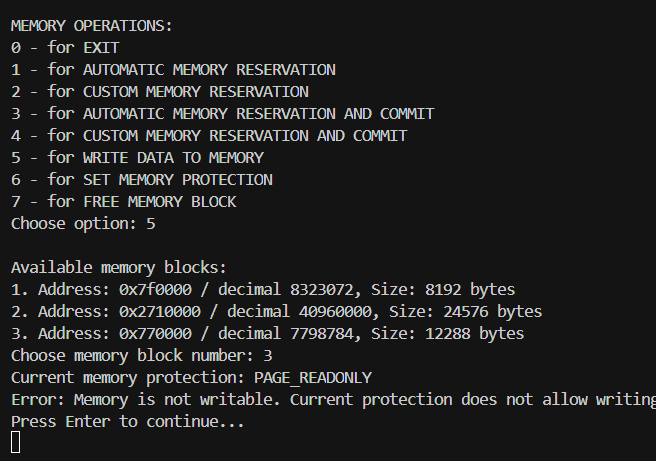
Изображение 5.2 – Проверка адреса памяти, резервируемого пользователем



Изображение 6.1 – Запись данных по адресу памяти



Изображение 6.2 – Изменение прав доступа к адресу памяти



Изображение 6.3 – Проверка изменения данных памяти по адресу с измененными правами доступа

**Заключение**

Работа с виртуальной памятью в Windows позволяет управлять выделением и защитой памяти на низком уровне. Можно как резервировать области памяти, так и передавать им физические ресурсы, а также менять права доступа, ограничивая чтение и запись. Кроме того, доступен инструмент для просмотра состояния памяти и общей загрузки системы, что помогает лучше понимать, как используются ресурсы. В целом, возможности управления памятью достаточно гибкие и позволяют более точно контролировать работу приложения.

**Код программы**

#include <string>

#include <iostream>

#include <windows.h>

#include <vector>

#include <limits>

using namespace std;

struct MemoryBlock {

    LPVOID address;

    SIZE\_T size;

    bool is\_committed;

};

vector<MemoryBlock> allocated\_memory;

int **enter\_integer**(const string& *message*, int *a*, int *b*) {

    int number;

    while (true) {

        cout << *message*;

        if (!(cin >> number)) {

            if (cin.eof()) {

                cout << "\nInput interrupted. Exiting...\n";

**exit**(1);

            }

            cout << "Invalid input! Please enter a number\n";

            cin.clear();

            cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        } else if (number >= *a* && number <= *b*) {

            return number;

        } else {

            cout << "Entered value is out of range [" << *a* << ", " << *b* << "]. Try again!\n";

        }

    }

}

void **clear\_screen**() {

#if defined(\_WIN32) || defined(\_WIN64)

**system**("cls");

#else

    system("clear");

#endif

}

int **main\_menu**() {

    cout << "\nOPTIONS:" << **endl**;

    cout << "0 - for EXIT" << **endl**;

    cout << "1 - for GET SYSTEM INFO" << **endl**;

    cout << "2 - for GLOBAL MEMORY STATUS" << **endl**;

    cout << "3 - for VIRTUAL QUERY" << **endl**;

    cout << "4 - for MEMORY OPERATIONS" << **endl**;

    return **enter\_integer**("Choose option: ", 0, 5);

}

void **get\_system\_info**(){

    SYSTEM\_INFO info;

**GetSystemInfo**(&info);

    cout << "\nSystem Info:" << **endl**;

    int processor\_architecture = info.wProcessorArchitecture;

    if (processor\_architecture == PROCESSOR\_ARCHITECTURE\_INTEL) cout << "Processor Architecture: x86" << **endl**;

    if (processor\_architecture == PROCESSOR\_ARCHITECTURE\_AMD64) cout << "Processor Architecture: x64" << **endl**;

    if (processor\_architecture == PROCESSOR\_ARCHITECTURE\_ARM) cout << "Processor Architecture: ARM" << **endl**;

    if (processor\_architecture == 12) cout << "Processor Architecture: ARM64" << **endl**;

    if (processor\_architecture == PROCESSOR\_ARCHITECTURE\_IA64) cout << "Processor Architecture based on Itanium" << **endl**;

    if (processor\_architecture == PROCESSOR\_ARCHITECTURE\_UNKNOWN) cout << "Unknown Processor Architecture" << **endl**;

    cout << "Number of Processors: " << info.dwNumberOfProcessors << **endl**;

    cout << "Page Size: " << info.dwPageSize << **endl**;

    cout << "Minimum Application Address: " << info.lpMinimumApplicationAddress << " / in decimal " << (DWORD)info.lpMinimumApplicationAddress << **endl**;

    cout << "Maximum Application Address: " << info.lpMaximumApplicationAddress << " / in decimal " << (DWORD)info.lpMaximumApplicationAddress << **endl**;

    cout << "Active Processor Mask: " << info.dwActiveProcessorMask << **endl**;

    cout << "Processor Type: " << info.dwProcessorType << **endl**;

    cout << "Allocation Granularity: " << info.dwAllocationGranularity << **endl**;

    cout << "Processor Level: " << info.wProcessorLevel << **endl**;

    cout << "Processor Revision: " << info.wProcessorRevision << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **global\_memory\_status**(){

    MEMORYSTATUS status;

**GlobalMemoryStatus**(&status);

    cout << "\nGlobal Memory Status:" << **endl**;

    cout << "Memory Load: " << status.dwMemoryLoad << "%" << **endl**;

    cout << "Total Physical Memory: " << status.dwTotalPhys << " bytes" << **endl**;

    cout << "Available Physical Memory: " << status.dwAvailPhys << " bytes" << **endl**;

    cout << "Total Page File: " << status.dwTotalPageFile << " bytes" << **endl**;

    cout << "Available Page File: " << status.dwAvailPageFile << " bytes" << **endl**;

    cout << "Total Virtual Address Space: " << status.dwTotalVirtual << " bytes" << **endl**;

    cout << "Available Virtual Address Space: " << status.dwAvailVirtual << " bytes" << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **get\_virtual\_query**(){

    SYSTEM\_INFO sys\_info;

**GetSystemInfo**(&sys\_info);

    DWORD min\_adress = (DWORD)sys\_info.lpMinimumApplicationAddress;

    DWORD max\_adress = (DWORD)sys\_info.lpMaximumApplicationAddress;

    int adress;

    adress = **enter\_integer**("Enter the adress in range [" + **to\_string**(min\_adress) + ", " + **to\_string**(max\_adress) + "]: ", min\_adress, max\_adress);

    MEMORY\_BASIC\_INFORMATION info;

    if (**VirtualQuery**((void\*)adress, &info, sizeof(info)) == 0) {

        cout << "Error! Invalid adress" << **endl**;

        return;

    }

    cout << "\nVirtual Query:" << **endl**;

    cout << "Base Address: " << info.BaseAddress << " / in decimal " << (DWORD)info.BaseAddress << **endl**;

    cout << "Allocation Base: " << info.AllocationBase << " / in decimal " << (DWORD)info.AllocationBase << **endl**;

    int protection = info.AllocationProtect;

    if (protection == PAGE\_READONLY) cout << "Allocation Protect: PAGE\_READONLY" << **endl**;

    if (protection == PAGE\_READWRITE) cout << "Allocation Protect: PAGE\_READWRITE" << **endl**;

    if (protection == PAGE\_WRITECOPY) cout << "Allocation Protect: PAGE\_WRITECOPY" << **endl**;

    if (protection == PAGE\_EXECUTE\_READ) cout << "Allocation Protect: PAGE\_EXECUTE\_READ" << **endl**;

    if (protection == PAGE\_EXECUTE\_READWRITE) cout << "Allocation Protect: PAGE\_EXECUTE\_READWRITE" << **endl**;

    if (protection == PAGE\_EXECUTE\_WRITECOPY) cout << "Allocation Protect: PAGE\_EXECUTE\_WRITECOPY" << **endl**;

    if (protection == PAGE\_NOACCESS) cout << "Allocation Protect: PAGE\_NOACCESS" << **endl**;

    if (protection == PAGE\_EXECUTE) cout << "Allocation Protect: PAGE\_EXECUTE" << **endl**;

    if (protection == 0x40000000) cout << "Allocation Protect: PAGE\_TARGETS\_INVALID" << **endl**;

    if (protection == 0x80000000) cout << "Allocation Protect: PAGE\_TARGETS\_NO\_UPDATE" << **endl**;

    cout << "Region Size: " << info.RegionSize << **endl**;

    int state = info.State;

    if (state == MEM\_COMMIT) cout << "State: MEM\_COMMIT" << **endl**;

    if (state == MEM\_FREE) cout << "State: MEM\_FREE" << **endl**;

    if (state == MEM\_RESERVE) cout << "State: MEM\_RESERVE" << **endl**;

    int protect = info.Protect;

    if (protect == PAGE\_READONLY) cout << "Protect: PAGE\_READONLY" << **endl**;

    if (protect == PAGE\_READWRITE) cout << "Protect: PAGE\_READWRITE" << **endl**;

    if (protect == PAGE\_WRITECOPY) cout << "Protect: PAGE\_WRITECOPY" << **endl**;

    if (protect == PAGE\_EXECUTE\_READ) cout << "Protect: PAGE\_EXECUTE\_READ" << **endl**;

    if (protect == PAGE\_EXECUTE\_READWRITE) cout << "Protect: PAGE\_EXECUTE\_READWRITE" << **endl**;

    if (protect == PAGE\_EXECUTE\_WRITECOPY) cout << "Protect: PAGE\_EXECUTE\_WRITECOPY" << **endl**;

    if (protect == PAGE\_NOACCESS) cout << "Protect: PAGE\_NOACCESS" << **endl**;

    if (protect == PAGE\_EXECUTE) cout << "Protect: PAGE\_EXECUTE" << **endl**;

    if (protect == 0x40000000) cout << "Protect: PAGE\_TARGETS\_INVALID" << **endl**;

    if (protect == 0x80000000) cout << "Protect: PAGE\_TARGETS\_NO\_UPDATE" << **endl**;

    int type = info.Type;

    if (type == MEM\_IMAGE) cout << "Type: MEM\_IMAGE" << **endl**;

    if (type == MEM\_MAPPED) cout << "Type: MEM\_MAPPED" << **endl**;

    if (type == MEM\_PRIVATE) cout << "Type: MEM\_PRIVATE" << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **memory\_operations\_menu**() {

    cout << "\nMEMORY OPERATIONS:" << **endl**;

    cout << "0 - for EXIT" << **endl**;

    cout << "1 - for AUTOMATIC MEMORY RESERVATION" << **endl**;

    cout << "2 - for CUSTOM MEMORY RESERVATION" << **endl**;

    cout << "3 - for AUTOMATIC MEMORY RESERVATION AND COMMIT" << **endl**;

    cout << "4 - for CUSTOM MEMORY RESERVATION AND COMMIT" << **endl**;

    cout << "5 - for WRITE DATA TO MEMORY" << **endl**;

    cout << "6 - for SET MEMORY PROTECTION" << **endl**;

    cout << "7 - for FREE MEMORY BLOCK" << **endl**;

}

void **automatic\_memory\_reservation**() {

    SYSTEM\_INFO sys\_info;

**GetSystemInfo**(&sys\_info);

    cout << "Enter memory size in bytes: ";

    SIZE\_T size;

    cin >> size;

    SIZE\_T remainder = size % 4096;

    if (remainder > 0) {

        size = size + (4096 - remainder);

        cout << "Size rounded to the nearest multiple of 4096: " << size << " bytes" << **endl**;

    }

    LPVOID address = **VirtualAlloc**(NULL, size, MEM\_RESERVE, PAGE\_READWRITE);

    if (address == NULL) {

        cout << "Error reserving memory. Error code: " << **GetLastError**() << **endl**;

        return;

    }

    MemoryBlock block;

    block.address = address;

    block.size = size;

    block.is\_committed = false;

    allocated\_memory.push\_back(block);

    cout << "Memory successfully reserved:" << **endl**;

    cout << "Address: " << address << " / in decimal " << (DWORD)address << **endl**;

    cout << "Size: " << size << " bytes" << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **custom\_memory\_reservation**() {

    SYSTEM\_INFO sys\_info;

**GetSystemInfo**(&sys\_info);

    DWORD min\_address = (DWORD)sys\_info.lpMinimumApplicationAddress;

    DWORD max\_address = (DWORD)sys\_info.lpMaximumApplicationAddress;

    cout << "Enter the starting address (in range [" << min\_address << ", " << max\_address << "]): ";

    DWORD address;

    cin >> address;

    DWORD remainder = address % 4096;

    if (remainder > 2048) {

        address = address + (4096 - remainder);

    } else {

        address = address - remainder;

    }

    cout << "Address rounded to the nearest multiple of 4096. Address: " << address << " / in decimal " << DWORD(address) << **endl**;

    if (address < min\_address || address > max\_address) {

        cout << "Error: address is out of valid range" << **endl**;

        return;

    }

    cout << "Enter memory size in bytes: ";

    SIZE\_T size;

    cin >> size;

    remainder = size % 4096;

    if (remainder > 0) {

        size = size + (4096 - remainder);

        cout << "Size rounded to the nearest multiple of 4096: " << size << " bytes" << **endl**;

    }

    LPVOID allocated\_address = **VirtualAlloc**((LPVOID)address, size, MEM\_RESERVE, PAGE\_READWRITE);

    if (allocated\_address == NULL) {

        cout << "Error reserving memory. Error code: " << **GetLastError**() << **endl**;

        return;

    }

    MemoryBlock block;

    block.address = allocated\_address;

    block.size = size;

    block.is\_committed = false;

    allocated\_memory.push\_back(block);

    cout << "Memory successfully reserved:" << **endl**;

    cout << "Requested address: " << (void\*)address << **endl**;

    cout << "Allocated address: " << allocated\_address << **endl**;

    cout << "Size: " << size << " bytes" << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **automatic\_memory\_reservation\_and\_commit**() {

    cout << "Enter memory size in bytes: ";

    SIZE\_T size;

    cin >> size;

    SIZE\_T remainder = size % 4096;

    if (remainder > 0) {

        size = size + (4096 - remainder);

        cout << "Size rounded to the nearest multiple of 4096: " << size << " bytes" << **endl**;

    }

    LPVOID address = **VirtualAlloc**(NULL, size, MEM\_RESERVE | MEM\_COMMIT, PAGE\_READWRITE);

    if (address == NULL) {

        cout << "Error allocating memory. Error code: " << **GetLastError**() << **endl**;

        return;

    }

    MemoryBlock block;

    block.address = address;

    block.size = size;

    block.is\_committed = true;

    allocated\_memory.push\_back(block);

    cout << "Memory successfully reserved and committed:" << **endl**;

    cout << "Address: " << address << "/ decimal " << DWORD(address) << **endl**;

    cout << "Size: " << size << " bytes" << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **custom\_memory\_reservation\_and\_commit**() {

    SYSTEM\_INFO sys\_info;

**GetSystemInfo**(&sys\_info);

    DWORD min\_address = (DWORD)sys\_info.lpMinimumApplicationAddress;

    DWORD max\_address = (DWORD)sys\_info.lpMaximumApplicationAddress;

    cout << "Enter the starting address (in range [" << min\_address << ", " << max\_address << "]): ";

    DWORD address;

    cin >> address;

    DWORD remainder = address % 4096;

    if (remainder > 2048) {

        address = address + (4096 - remainder);

    } else {

        address = address - remainder;

    }

    cout << "Address rounded to the nearest multiple of 4096. Address: " << address << " / in decimal " << DWORD(address) << **endl**;

    if (address < min\_address || address > max\_address) {

        cout << "Error: address is out of valid range" << **endl**;

        return;

    }

    cout << "Enter memory size in bytes: ";

    SIZE\_T size;

    cin >> size;

    remainder = size % 4096;

    if (remainder > 0) {

        size = size + (4096 - remainder);

        cout << "Size rounded to the nearest multiple of 4096: " << size << " bytes" << **endl**;

    }

    LPVOID allocated\_address = **VirtualAlloc**((LPVOID)address, size, MEM\_RESERVE | MEM\_COMMIT, PAGE\_READWRITE);

    if (allocated\_address == NULL) {

        cout << "Error allocating memory. Error code: " << **GetLastError**() << **endl**;

        return;

    }

    MemoryBlock block;

    block.address = allocated\_address;

    block.size = size;

    block.is\_committed = true;

    allocated\_memory.push\_back(block);

    cout << "Memory successfully reserved and committed:" << **endl**;

    cout << "Requested address: " << (void\*)address << **endl**;

    cout << "Allocated address: " << allocated\_address << **endl**;

    cout << "Size: " << size << " bytes" << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **write\_data\_to\_memory**() {

    if (allocated\_memory.empty()) {

        cout << "Error: No memory blocks allocated. Please allocate memory first" << **endl**;

        cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

        return;

    }

    cout << "\nAvailable memory blocks:" << **endl**;

    for (size\_t i = 0; i < allocated\_memory.size(); i++) {

        cout << i + 1 << ". Address: " << allocated\_memory[i].address << " / decimal " << (DWORD)allocated\_memory[i].address << ", Size: " << allocated\_memory[i].size << " bytes" << **endl**;

    }

    int block\_index = **enter\_integer**("Choose memory block number: ", 1, allocated\_memory.size()) - 1;

    MemoryBlock& block = allocated\_memory[block\_index];

    MEMORY\_BASIC\_INFORMATION mbi;

    if (**VirtualQuery**(block.address, &mbi, sizeof(mbi))) {

        cout << "Current memory protection: ";

        switch (mbi.Protect) {

            case PAGE\_READONLY: cout << "PAGE\_READONLY"; break;

            case PAGE\_READWRITE: cout << "PAGE\_READWRITE"; break;

            case PAGE\_WRITECOPY: cout << "PAGE\_WRITECOPY"; break;

            case PAGE\_EXECUTE\_READ: cout << "PAGE\_EXECUTE\_READ"; break;

            case PAGE\_EXECUTE\_READWRITE: cout << "PAGE\_EXECUTE\_READWRITE"; break;

            case PAGE\_EXECUTE\_WRITECOPY: cout << "PAGE\_EXECUTE\_WRITECOPY"; break;

            case PAGE\_NOACCESS: cout << "PAGE\_NOACCESS"; break;

            case PAGE\_EXECUTE: cout << "PAGE\_EXECUTE"; break;

            default: cout << "Unknown";

        }

        cout << **endl**;

        if (mbi.Protect == PAGE\_NOACCESS || mbi.Protect == PAGE\_READONLY || mbi.Protect == PAGE\_EXECUTE\_READ) {

            cout << "Error: Memory is not writable. Current protection does not allow writing" << **endl**;

            cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

            return;

        }

    }

    cout << "Enter offset from the start of the block (0 to " << block.size - 1 << "): ";

    SIZE\_T offset;

    cin >> offset;

    if (offset >= block.size) {

        cout << "Error: Offset is out of bounds!" << **endl**;

        cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

        return;

    }

    cout << "Enter data to write (integer): ";

    int data;

    cin >> data;

    if (!block.is\_committed) {

        cout << "Error: Memory is not committed. Please commit memory first" << **endl**;

        cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

        return;

    }

    int\* target\_address = (int\*)((BYTE\*)block.address + offset);

    \*target\_address = data;

    cout << "Data successfully written:" << **endl**;

    cout << "Address: " << target\_address << " / decimal " << (DWORD)target\_address << **endl**;

    cout << "Value: " << \*target\_address << **endl**;

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **free\_all\_memory**() {

    cout << "\nFreeing all allocated memory..." << **endl**;

    for (const auto& block : allocated\_memory) {

        if (**VirtualFree**(block.address, 0, MEM\_RELEASE)) {

            cout << "Successfully freed memory at address: " << block.address << " / in decimal " << DWORD(block.address) << **endl**;

        } else {

            cout << "Failed to free memory at address: " << block.address << " / in decimal " << DWORD(block.address) << ". Error code: " << **GetLastError**() << **endl**;

        }

    }

    allocated\_memory.clear();

}

void **set\_memory\_protection**() {

    if (allocated\_memory.empty()) {

        cout << "Error: No memory blocks allocated. Please allocate memory first" << **endl**;

        cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

        return;

    }

    cout << "\nAvailable memory blocks:" << **endl**;

    for (size\_t i = 0; i < allocated\_memory.size(); i++) {

        cout << i + 1 << ". Address: " << allocated\_memory[i].address

             << " / decimal " << (DWORD)allocated\_memory[i].address

             << ", Size: " << allocated\_memory[i].size << " bytes" << **endl**;

    }

    int block\_index = **enter\_integer**("Choose memory block number: ", 1, allocated\_memory.size()) - 1;

    MemoryBlock& block = allocated\_memory[block\_index];

    cout << "\nAvailable protection options:" << **endl**;

    cout << "1 - PAGE\_READONLY" << **endl**;

    cout << "2 - PAGE\_READWRITE" << **endl**;

    cout << "3 - PAGE\_WRITECOPY" << **endl**;

    cout << "4 - PAGE\_EXECUTE\_READ" << **endl**;

    cout << "5 - PAGE\_EXECUTE\_READWRITE" << **endl**;

    cout << "6 - PAGE\_EXECUTE\_WRITECOPY" << **endl**;

    cout << "7 - PAGE\_NOACCESS" << **endl**;

    cout << "8 - PAGE\_EXECUTE" << **endl**;

    int protection\_choice = **enter\_integer**("Choose protection type (1-8): ", 1, 8);

    DWORD new\_protection;

    switch (protection\_choice) {

        case 1: new\_protection = PAGE\_READONLY; break;

        case 2: new\_protection = PAGE\_READWRITE; break;

        case 3: new\_protection = PAGE\_WRITECOPY; break;

        case 4: new\_protection = PAGE\_EXECUTE\_READ; break;

        case 5: new\_protection = PAGE\_EXECUTE\_READWRITE; break;

        case 6: new\_protection = PAGE\_EXECUTE\_WRITECOPY; break;

        case 7: new\_protection = PAGE\_NOACCESS; break;

        case 8: new\_protection = PAGE\_EXECUTE; break;

        default: new\_protection = PAGE\_READWRITE;

    }

    DWORD old\_protection;

    if (**VirtualProtect**(block.address, block.size, new\_protection, &old\_protection)) {

        cout << "\nProtection successfully changed:" << **endl**;

        cout << "Address: " << block.address << " / decimal " << (DWORD)block.address << **endl**;

        cout << "Size: " << block.size << " bytes" << **endl**;

        cout << "Old protection: ";

        switch (old\_protection) {

            case PAGE\_READONLY: cout << "PAGE\_READONLY"; break;

            case PAGE\_READWRITE: cout << "PAGE\_READWRITE"; break;

            case PAGE\_WRITECOPY: cout << "PAGE\_WRITECOPY"; break;

            case PAGE\_EXECUTE\_READ: cout << "PAGE\_EXECUTE\_READ"; break;

            case PAGE\_EXECUTE\_READWRITE: cout << "PAGE\_EXECUTE\_READWRITE"; break;

            case PAGE\_EXECUTE\_WRITECOPY: cout << "PAGE\_EXECUTE\_WRITECOPY"; break;

            case PAGE\_NOACCESS: cout << "PAGE\_NOACCESS"; break;

            case PAGE\_EXECUTE: cout << "PAGE\_EXECUTE"; break;

            default: cout << "Unknown";

        }

        cout << **endl**;

        cout << "New protection: ";

        switch (new\_protection) {

            case PAGE\_READONLY: cout << "PAGE\_READONLY"; break;

            case PAGE\_READWRITE: cout << "PAGE\_READWRITE"; break;

            case PAGE\_WRITECOPY: cout << "PAGE\_WRITECOPY"; break;

            case PAGE\_EXECUTE\_READ: cout << "PAGE\_EXECUTE\_READ"; break;

            case PAGE\_EXECUTE\_READWRITE: cout << "PAGE\_EXECUTE\_READWRITE"; break;

            case PAGE\_EXECUTE\_WRITECOPY: cout << "PAGE\_EXECUTE\_WRITECOPY"; break;

            case PAGE\_NOACCESS: cout << "PAGE\_NOACCESS"; break;

            case PAGE\_EXECUTE: cout << "PAGE\_EXECUTE"; break;

            default: cout << "Unknown";

        }

        cout << **endl**;

        MEMORY\_BASIC\_INFORMATION mbi;

        if (**VirtualQuery**(block.address, &mbi, sizeof(mbi))) {

            cout << "\nCurrent protection (verified by VirtualQuery): ";

            switch (mbi.Protect) {

                case PAGE\_READONLY: cout << "PAGE\_READONLY"; break;

                case PAGE\_READWRITE: cout << "PAGE\_READWRITE"; break;

                case PAGE\_WRITECOPY: cout << "PAGE\_WRITECOPY"; break;

                case PAGE\_EXECUTE\_READ: cout << "PAGE\_EXECUTE\_READ"; break;

                case PAGE\_EXECUTE\_READWRITE: cout << "PAGE\_EXECUTE\_READWRITE"; break;

                case PAGE\_EXECUTE\_WRITECOPY: cout << "PAGE\_EXECUTE\_WRITECOPY"; break;

                case PAGE\_NOACCESS: cout << "PAGE\_NOACCESS"; break;

                case PAGE\_EXECUTE: cout << "PAGE\_EXECUTE"; break;

                default: cout << "Unknown";

            }

            cout << **endl**;

        }

    } else {

        cout << "Error changing protection. Error code: " << **GetLastError**() << **endl**;

    }

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **free\_memory\_block**() {

    if (allocated\_memory.empty()) {

        cout << "Error: No memory blocks allocated. Please allocate memory first" << **endl**;

        cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

        return;

    }

    cout << "\nAvailable memory blocks:" << **endl**;

    for (size\_t i = 0; i < allocated\_memory.size(); i++) {

        cout << i + 1 << ". Address: " << allocated\_memory[i].address

             << " / decimal " << (DWORD)allocated\_memory[i].address

             << ", Size: " << allocated\_memory[i].size << " bytes" << **endl**;

    }

    int block\_index = **enter\_integer**("Choose memory block number to free (1-" + **to\_string**(allocated\_memory.size()) + "): ", 1, allocated\_memory.size()) - 1;

    MemoryBlock& block = allocated\_memory[block\_index];

    if (**VirtualFree**(block.address, 0, MEM\_RELEASE)) {

        cout << "Successfully freed memory block:" << **endl**;

        cout << "Address: " << block.address << " / decimal " << (DWORD)block.address << **endl**;

        cout << "Size: " << block.size << " bytes" << **endl**;

        allocated\_memory.erase(allocated\_memory.begin() + block\_index);

        cout << "Memory block removed from the list" << **endl**;

    } else {

        cout << "Failed to free memory block. Error code: " << **GetLastError**() << **endl**;

    }

    cout << "Press Enter to continue..." << **endl**;

**getchar**();

**getchar**();

}

void **memory\_operations**() {

    int option;

    do {

**memory\_operations\_menu**();

        option = **enter\_integer**("Choose option: ", 0, 7);

        switch (option) {

            case 1:

**automatic\_memory\_reservation**();

                break;

            case 2:

**custom\_memory\_reservation**();

                break;

            case 3:

**automatic\_memory\_reservation\_and\_commit**();

                break;

            case 4:

**custom\_memory\_reservation\_and\_commit**();

                break;

            case 5:

**write\_data\_to\_memory**();

                break;

            case 6:

**set\_memory\_protection**();

                break;

            case 7:

**free\_memory\_block**();

                break;

        }

    } while (option != 0);

}

int **main**(){

    int option;

    do {

        option = **main\_menu**();

        switch (option) {

        case 1:

**get\_system\_info**();

            break;

        case 2:

**global\_memory\_status**();

            break;

        case 3:

**get\_virtual\_query**();

            break;

        case 4:

**memory\_operations**();

            break;

        default:

            cout << "Invalid option. Please choose a valid option" << **endl**;

            break;

        }

    } while (option!=0);

**free\_all\_memory**();

    cout << "Goodbye";

    return 0;

}