**МІНІСТЕРСТВО НАУКИ ТА ОСВІТИ УКРАЇНИ**

**ОДЕСЬКА НАЦІОНАЛЬНА АКАДЕМІЯ ЗВ'ЯЗКУ ІМ. О. С. ПОПОВА**

**ЗВІТ**

**до лабораторного завдання №5**

**з дисципліни «Мережеве програмування»**

Виконав

студент групи ІПЗ-3.02

Писарев С. Д.

Перевірив

Викладач Вороний С. М.

Задание

Разработайте серверную часть программы из лабораторной работы No3 в двух вариантах так, чтобы в одном из них работа с сокетами была реализована по модели WSAAsyncSelect, а в другом – по модели WSAEventSelect.

**Вариант №7**

Клиентская программа оправляет на сервер текстовый файл из N строк (путь к файлу задается через пользовательский интерфейс клиента). Сервер подсчитывает общее количество слов длиной не менее пяти символов в строках файла. Это количество возвращает клиенту.

Листинг

Server.cpp

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#define WIN32\_LEAN\_AND\_MEAN

#include <iostream>

#include <fstream>

#include <thread>

#include <winsock2.h>

#include <ws2tcpip.h>

#include "Server.h"

#include "Helpers.h"

#include <mswsock.h>

#pragma comment(lib, "Ws2\_32.lib")

//#pragma comment (lib, "mswsock.lib")

using namespace std;

constexpr auto DEFAULT\_BUFLEN = 512;

const string filename = "fileToProcess.txt";

SOCKET ListenSocket = INVALID\_SOCKET;

SOCKET ClientSocket = INVALID\_SOCKET;

constexpr auto TIMEOUT\_SEC = 150;

constexpr auto TIMEOUT\_MSEC = 0;

#define WM\_SOCKET (WM\_USER + 1)

typedef struct \_SOCKET\_INFORMATION{

    CHAR Buffer[DEFAULT\_BUFLEN];

    WSABUF DataBuf;

    SOCKET Socket;

    WSAOVERLAPPED Overlapped;

    DWORD BytesSEND;

    DWORD BytesRECV;

} SOCKET\_INFORMATION, \* LPSOCKET\_INFORMATION;

WSAEVENT EventArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

LPSOCKET\_INFORMATION SocketArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

DWORD EventTotal = 0;

int fileSize = 0;

void OutputClientState(SOCKET\* socket, bool isConnected)

{

    int foreground\_color, background\_color;

    string state;

    struct sockaddr\_in sin;

    socklen\_t len = sizeof(sin);

    getsockname(\*socket, (struct sockaddr\*)&sin, &len);

    if (isConnected)

    {

        foreground\_color = FOREGROUND\_GREEN;

        background\_color = BACKGROUND\_GREEN;

        state = "connected";

    }

    else

    {

        foreground\_color = FOREGROUND\_RED;

        background\_color = BACKGROUND\_RED;

        state = "disconnected";

    }

    HANDLE hConsoleOutput = GetStdHandle(STD\_OUTPUT\_HANDLE);

    SetConsoleTextAttribute(hConsoleOutput, foreground\_color);

    cout << "The client with an IP address ";

    SetConsoleTextAttribute(hConsoleOutput, background\_color);

    cout << inet\_ntoa(sin.sin\_addr);

    SetConsoleTextAttribute(hConsoleOutput, foreground\_color);

    cout << " was " << state << " via " << htons(sin.sin\_port) << " port." << endl;

    SetConsoleTextAttribute(hConsoleOutput, FOREGROUND\_BLUE | FOREGROUND\_GREEN | FOREGROUND\_RED);

}

Server::Server(string& serverPort)

{

    CHAR AcceptBuffer[2 \* (sizeof(SOCKADDR\_IN) + 16)];

    DWORD Ret;

    SOCKADDR \_INInternetAddr;

    HWND Window;

    WSADATA wsaData;

    WSAOVERLAPPED ListenOverlapped;

    DWORD RecvBytes, SendBytes;

    DWORD dwBytes;

    DWORD Index;

    DWORD Flags;

    DWORD Bytes;

    DWORD BytesTransferred;

    LPSOCKET\_INFORMATION SI;

    // Initialize Winsock

    int iResult;

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0) {

        printf("WSAStartup failed: %d\n", iResult);

        return;

    }

    SOCKET ClientSocket = INVALID\_SOCKET;

    struct addrinfo\* result = NULL, \* ptr = NULL, hints;

    ZeroMemory(&hints, sizeof(hints));

    hints.ai\_family = AF\_INET;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_protocol = IPPROTO\_TCP;

    hints.ai\_flags = AI\_PASSIVE;

    // Resolve the local address and port to be used by the server

    iResult = getaddrinfo(NULL, serverPort.c\_str(), &hints, &result);

    if (iResult != 0) {

        printf("getaddrinfo failed: %d\n", iResult);

        WSACleanup();

        return;

    }

    // Create a SOCKET for the server to listen for client connections

    if ((ListenSocket = WSASocket(result->ai\_family, result->ai\_socktype, result->ai\_protocol, NULL, 0,

        WSA\_FLAG\_OVERLAPPED)) == INVALID\_SOCKET)

    {

        printf("Failed to get a socket %d\n", WSAGetLastError());

        return;

    }

    else

        printf("WSASocket() is OK!\n");

    /\*InternetAddr.sin\_family = AF\_INET;

    InternetAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    InternetAddr.sin\_port = htons(PORT);\*/

    // Setup the TCP listening socket

    // Without a double colon behind the bind function it won't work. It means that it will use the bind func from the global namespace.

    iResult = ::bind(ListenSocket, result->ai\_addr, (int)result->ai\_addrlen);

    if (iResult == SOCKET\_ERROR) {

        printf("bind failed with error: %d\n", WSAGetLastError());

        freeaddrinfo(result);

        closesocket(ListenSocket);

        WSACleanup();

        return;

    }

    //freeaddrinfo(result);

    auto CtrlHandler = [](DWORD fdwCtrlType)

    {

        SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

        closesocket(ListenSocket);

        WSACleanup();

        return FALSE;

    };

    SetConsoleCtrlHandler(CtrlHandler, true);

    if (listen(ListenSocket, SOMAXCONN) == SOCKET\_ERROR) {

        printf("Listen failed with error: %ld\n", WSAGetLastError());

        closesocket(ListenSocket);

        WSACleanup();

        return;

    }

    // Setup the listening socket for connections

    SOCKET AcceptSocket;

    if ((AcceptSocket = WSASocket(AF\_INET, SOCK\_STREAM, 0, NULL, 0,

        WSA\_FLAG\_OVERLAPPED)) == INVALID\_SOCKET)

    {

        printf("Failed to get a socket %d\n", WSAGetLastError());

        return;

    }

    else

        printf("WSASocket() is OK!\n");

    ZeroMemory(&ListenOverlapped, sizeof(OVERLAPPED));

    if ((EventArray[0] = ListenOverlapped.hEvent = WSACreateEvent()) == WSA\_INVALID\_EVENT)

    {

        printf("WSACreateEvent() failed with error %d\n", WSAGetLastError());

        return;

    }

    else

        printf("WSACreateEvent() is OK!\n");

    EventTotal = 1;

    if (AcceptEx(ListenSocket, AcceptSocket, (PVOID)AcceptBuffer, 0, sizeof(SOCKADDR\_IN) + 16, sizeof(SOCKADDR\_IN) + 16, &Bytes, &ListenOverlapped) == FALSE)

    {

        if (WSAGetLastError() != ERROR\_IO\_PENDING)

        {

            printf("AcceptEx() failed with error %d\n", WSAGetLastError());

            return;

        }

        else

            printf("AcceptEx() is OK!\n");

    }

    // Process asynchronous AcceptEx, WSASend, WSARecv requests

    while (TRUE)

    {

        if ((Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA\_INFINITE, FALSE)) == WSA\_WAIT\_FAILED)

        {

            printf("WSAWaitForMultipleEvents()failed %d\n", WSAGetLastError());

            return;

        }

        else

            printf("\nWSAWaitForMultipleEvents() is OK!\n");

        // If the event triggered was zero then a connection attempt was made

        // on our listening socket

        if ((Index - WSA\_WAIT\_EVENT\_0) == 0)

        {

            // Check the returns from the overlapped I/O operation on the listening socket

            if (WSAGetOverlappedResult(ListenSocket, &(ListenOverlapped), &BytesTransferred, FALSE, &Flags) == FALSE)

            {

                printf("WSAGetOverlappedResult() failed with error %d\n", WSAGetLastError());

                return;

            }

            else

                printf("WSAGetOverlappedResult() is OK!\n");

            printf("Socket %d got connected...\n", (int)AcceptSocket);

            if (EventTotal > WSA\_MAXIMUM\_WAIT\_EVENTS)

            {

                printf("Too many connections - closing socket.\n");

                closesocket(AcceptSocket);

                continue;

            }

            else

            {

                // Create a socket information structure to associate with the accepted socket

                if ((SocketArray[EventTotal] = (LPSOCKET\_INFORMATION)GlobalAlloc(GPTR, sizeof(SOCKET\_INFORMATION))) == NULL)

                {

                         printf("GlobalAlloc() failed with error %d\n", GetLastError());

                    return;

                }

                else

                    printf("GlobalAlloc() for LPSOCKET\_INFORMATION is OK!\n");

                // Fill in the details of our accepted socket

                SocketArray[EventTotal]->Socket = AcceptSocket;

                ZeroMemory(&(SocketArray[EventTotal]->Overlapped), sizeof(OVERLAPPED));

                SocketArray[EventTotal]->BytesSEND = 0;

                SocketArray[EventTotal]->BytesRECV = 0;

                SocketArray[EventTotal]->DataBuf.len = DEFAULT\_BUFLEN;

                SocketArray[EventTotal]->DataBuf.buf = SocketArray[EventTotal]->Buffer;

                if ((SocketArray[EventTotal]->Overlapped.hEvent = EventArray[EventTotal] = WSACreateEvent()) == WSA\_INVALID\_EVENT)

                {

                    printf("WSACreateEvent() failed with error %d\n", WSAGetLastError());

                    return;

                }

                else

                    printf("WSACreateEvent() is OK!\n");

                // Post a WSARecv request to to begin receiving data on the socket

                if (WSARecv(SocketArray[EventTotal]->Socket, &(SocketArray[EventTotal]->DataBuf), 1, &RecvBytes, &Flags, &(SocketArray[EventTotal]->Overlapped), NULL) == SOCKET\_ERROR)

                {

                    if (WSAGetLastError() != ERROR\_IO\_PENDING)

                    {

                        printf("WSARecv() failed with error %d\n", WSAGetLastError());

                        return;

                    }

                }

                else

                {

                    printf("WSARecv() is OK!\n");

                    //cout << "bytes\_read = " << RecvBytes << endl;

                    //

                    //string actualData(SocketArray[EventTotal]->DataBuf.buf, SocketArray[EventTotal]->DataBuf.buf + SocketArray[EventTotal]->BytesRECV);

                    //if (fileSize == 0)

                    //{

                    //    fileSize = stoi(actualData);

                    //    SocketArray[EventTotal]->BytesRECV = 0;

                    //    continue;

                    //}

                    //string fileData;

                    //fileData.append(actualData);

                    //fileSize -= RecvBytes;

                    //if (fileSize == 0)

                    //{

                    //    // A file receiving is complete.

                    //    // Writes file contents in binary to a new file.

                    //    fstream myfile(filename, ios::binary | ios::out | ios::trunc);

                    //    myfile.write(fileData.c\_str(), fileData.length());

                    //    myfile.close();

                    //    myfile.open(filename, ios::in);

                    //    string word;

                    //    int numOfWordsGTEQFiveLetters = 0;

                    //    // Reads the file word by word.

                    //    while (myfile >> word)

                    //    {

                    //        if (word.length() >= 5)

                    //        {

                    //            numOfWordsGTEQFiveLetters++;

                    //        }

                    //    }

                    //    string valueToSend = to\_string(numOfWordsGTEQFiveLetters);

                    //    SocketArray[EventTotal]->DataBuf.buf = (char\*)valueToSend.c\_str();

                    //    SocketArray[EventTotal]->DataBuf.len = valueToSend.length();

                    //    SendBytes = valueToSend.length();

                    //    if (WSASend(SocketArray[EventTotal]->Socket, &(SocketArray[EventTotal]->DataBuf), 1, &SendBytes, 0, NULL, NULL) == SOCKET\_ERROR)

                    //    {

                    //        if (WSAGetLastError() != WSAEWOULDBLOCK)

                    //        {

                    //            printf("WSASend() failed with error %d\n", WSAGetLastError());

                    //            return;

                    //        }

                    //        // A WSAEWOULDBLOCK error has occurred. An FD\_WRITE event will be posted

                    //        // when more buffer space becomes available

                    //    }

                    //    else

                    //    {

                    //        printf("WSASend() is fine! Thank you...\n");

                    //        SocketArray[EventTotal]->BytesRECV = 0;

                    //    }

                    //    myfile.close();

                    //    fileData.clear();

                    //}

                    EventTotal++;

                }

                // Make a new socket for accepting future connections and post another AcceptEx call

                if ((AcceptSocket = WSASocket(AF\_INET, SOCK\_STREAM, 0, NULL, 0, WSA\_FLAG\_OVERLAPPED)) == INVALID\_SOCKET)

                {

                    printf("Failed to get a socket %d\n", WSAGetLastError());

                    return;

                }

                else

                    printf("WSASocket() is OK!\n");

                WSAResetEvent(EventArray[0]);

                ZeroMemory(&ListenOverlapped, sizeof(OVERLAPPED));

                ListenOverlapped.hEvent = EventArray[0];

                if (AcceptEx(ListenSocket, AcceptSocket, (PVOID)AcceptBuffer, 0,

                    sizeof(SOCKADDR\_IN) + 16, sizeof(SOCKADDR\_IN) + 16, &Bytes, &ListenOverlapped) == FALSE)

                {

                    if (WSAGetLastError() != ERROR\_IO\_PENDING)

                    {

                        printf("AcceptEx() failed with error %d\n", WSAGetLastError());

                        return;

                    }

                }

                else

                    printf("AcceptEx() is OK!\n");

                continue;

            }

            SI = SocketArray[Index - WSA\_WAIT\_EVENT\_0];

            WSAResetEvent(EventArray[Index - WSA\_WAIT\_EVENT\_0]);

            if (WSAGetOverlappedResult(SI->Socket, &(SI->Overlapped), &BytesTransferred, FALSE, &Flags) == FALSE)

            {

                printf("WSAGetOverlappedResult() failed with error %d\n", WSAGetLastError());

                return;

            }

            else

                printf("WSAGetOverlappedResult() is OK!\n");

            // First check to see if the peer has closed the connection and if so

            // then close the socket and cleanup the SOCKET\_INFORMATION structure

            // associated with the socket

            if (BytesTransferred == 0)

            {

                printf("Closing socket %d\n", (int)SI->Socket);

                if (closesocket(SI->Socket) == SOCKET\_ERROR)

                {

                    printf("closesocket() failed with error %d\n", WSAGetLastError());

                }

                else

                    printf("closesocket() is OK!\n");

                GlobalFree(SI);

                WSACloseEvent(EventArray[Index - WSA\_WAIT\_EVENT\_0]);

                // Cleanup SocketArray and EventArray by removing the socket event handle

                // and socket information structure if they are not at the end of the arrays

                if ((Index - WSA\_WAIT\_EVENT\_0) + 1 != EventTotal)

                    for (int i = Index - WSA\_WAIT\_EVENT\_0; i < EventTotal; i++)

                    {

                        EventArray[i] = EventArray[i + 1];

                        SocketArray[i] = SocketArray[i + 1];

                    }

                EventTotal--;

                continue;

            }

            // Check to see if the BytesRECV field equals zero. If this is so, then

            // this means a WSARecv call just completed so update the BytesRECV field

            // with the BytesTransferred value from the completed WSARecv() call

            if (SI->BytesRECV == 0)

            {

                SI->BytesRECV = BytesTransferred;

                SI->BytesSEND = 0;

            }

            else

            {

                SI->BytesSEND += BytesTransferred;

            }

            if (SI->BytesRECV > SI->BytesSEND)

            {

                // Post another WSASend() request

                // Since WSASend() is not guaranteed to send all of the bytes requested,

                // continue posting WSASend() calls until all received bytes are sent

                ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));

                SI->Overlapped.hEvent = EventArray[Index - WSA\_WAIT\_EVENT\_0];

                SI->DataBuf.buf = SI->Buffer + SI->BytesSEND;

                SI->DataBuf.len = SI->BytesRECV - SI->BytesSEND;

                if (WSASend(SI->Socket, &(SI->DataBuf), 1, &SendBytes, 0, &(SI->Overlapped), NULL) == SOCKET\_ERROR)

                {

                    if (WSAGetLastError() != ERROR\_IO\_PENDING)

                    {

                        printf("WSASend() failed with error %d\n", WSAGetLastError());

                        return;

                    }

                }

                else

                    printf("WSASend() is OK!\n");

            }

            else

            {

                SI->BytesRECV = 0;

                // Now that there are no more bytes to send post another WSARecv() request

                Flags = 0;

                ZeroMemory(&(SI->Overlapped), sizeof(WSAOVERLAPPED));

                SI->Overlapped.hEvent = EventArray[Index - WSA\_WAIT\_EVENT\_0];

                SI->DataBuf.len = DEFAULT\_BUFLEN;

                SI->DataBuf.buf = SI->Buffer;

                if (WSARecv(SI->Socket, &(SI->DataBuf), 1, &RecvBytes, &Flags, &(SI->Overlapped), NULL) == SOCKET\_ERROR)

                {

                    if (WSAGetLastError() != ERROR\_IO\_PENDING)

                    {

                        printf("WSARecv() failed with error %d\n", WSAGetLastError());

                        return;

                    }

                }

                else

                    printf("WSARecv() is OK!\n");

            }

        }

    }

}

ClientServer.cpp

#undef UNICODE

#define WIN32\_LEAN\_AND\_MEAN

#include <iostream>

#include <stdio.h>

#include <algorithm>

#include <iterator>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <boost/program\_options.hpp>

#include "Client.h"

#include "Server.h"

#include "ClientServer.h"

#include "Helpers.h"

#pragma comment(lib, "Ws2\_32.lib")

#pragma comment (lib, "Mswsock.lib")

#pragma comment (lib, "AdvApi32.lib")

using namespace std;

namespace po = boost::program\_options;

int main(int argc, char\* argv[])

{

    string PORT, IP\_ADDRESS;

    try {

        po::options\_description desc("Options");

        desc.add\_options()

            ("help,h", "Produce a help message.")

            ("server,S", "The program will start as a server.")

            ("client,C", "The program will start as a client.")

            ("ip,I", po::value< string >(&IP\_ADDRESS), "Provides an IP address.")

            ("port,P", po::value< string >(&PORT), "Provides a port.")

            ;

        po::variables\_map vm;

        po::store(po::parse\_command\_line(argc, argv, desc), vm);

        po::notify(vm);

        if (vm.count("help")) {

            cout << desc << "\n";

            return 0;

        }

        cout << "IP: " << IP\_ADDRESS << endl

             << "Port: " << PORT << endl << endl;

        //SetConsoleCtrlHandler(CtrlHandler, true);

        if (vm.count("server")) {

            cout << "The program was started as a server." << "\n";

            new Server(PORT);

        }

        else if (vm.count("client"))

        {

            cout << "The program was started as a client." << "\n";

            new Client(PORT, IP\_ADDRESS);

        }

    }

    catch (std::exception& e) {

        cerr << "error: " << e.what() << "\n";

        return 1;

    }

    catch (...) {

        cerr << "Exception of unknown type!\n";

    }

    return 0;

}

Client.cpp

#undef UNICODE

#define WIN32\_LEAN\_AND\_MEAN

#include <iostream>

#include <fstream>

#include <WinSock2.h>

#include <ws2tcpip.h>

#include <thread>

#include "Client.h"

#include "Helpers.h"

#pragma comment(lib, "Ws2\_32.lib")

#pragma comment (lib, "Mswsock.lib")

#pragma comment (lib, "AdvApi32.lib")

using namespace std;

constexpr auto DEFAULT\_BUFLEN = 512;

Client::Client(string& PORT, string& IP\_ADDRESS)

{

    WSADATA wsaData;

    int iResult;

    SOCKET ConnectSocket = INVALID\_SOCKET;

    struct addrinfo\* result = NULL,

                   \* ptr = NULL,

                     hints;

    // Initializes Winsock.

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup failed with error: %d\n", iResult);

        return;

    }

    ZeroMemory(&hints, sizeof(hints));

    hints.ai\_family = AF\_UNSPEC;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_protocol = IPPROTO\_TCP;

    // Resolves a server address and port.

    iResult = getaddrinfo(IP\_ADDRESS.c\_str(), PORT.c\_str(), &hints, &result);

    if (iResult != 0)

    {

        printf("getaddrinfo failed with error: %d\n", iResult);

        WSACleanup();

        return;

    }

    // Attempt to connect to an address until one succeeds

    for (ptr = result; ptr != NULL; ptr = ptr->ai\_next)

    {

        // Creates a SOCKET for connecting to a server.

        ConnectSocket = socket(ptr->ai\_family,

                               ptr->ai\_socktype,

                               ptr->ai\_protocol);

        if (ConnectSocket == INVALID\_SOCKET)

        {

            printf("socket failed with error: %ld\n", WSAGetLastError());

            WSACleanup();

            return;

        }

        // Connects to a server.

        iResult = connect(ConnectSocket, ptr->ai\_addr, (int)ptr->ai\_addrlen);

        if (iResult == SOCKET\_ERROR)

        {

            closesocket(ConnectSocket);

            ConnectSocket = INVALID\_SOCKET;

            continue;

        }

        break;

    }

    freeaddrinfo(result);

    if (ConnectSocket == INVALID\_SOCKET)

    {

        printf("Unable to connect to server!\n");

        WSACleanup();

        return;

    }

    string input;

    char recvbuf[DEFAULT\_BUFLEN];

    int recvbuflen = DEFAULT\_BUFLEN;

    while (true)

    {

        string path;

        cout << "Type the path to a file." << endl;

        cin >> path;

        // Reads contents of the file.

        ifstream in(path);

        string contents((istreambuf\_iterator<char>(in)),

            istreambuf\_iterator<char>());

        // Sends the size of the file contents and then the contents itself.

        string contentsSize = to\_string(contents.length());

        send(ConnectSocket, contentsSize.c\_str(), contentsSize.length(), 0);

        iResult = send(ConnectSocket, contents.c\_str(), contents.length(), 0);

        if (iResult == SOCKET\_ERROR)

        {

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED);

            printf("The server was shutdown.");

            CorrectClosing(ConnectSocket);

            return;

        }

        else

        {

            // Receives the answer from a server.

            iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);

            string actualData(recvbuf, recvbuf + iResult);

            // Nicely displays the answer.

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_BLUE);

            cout << "The num of words 5 letters or more in the file: ";

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), BACKGROUND\_BLUE);

            cout << actualData << endl << endl;

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

        }

    }

}

Helpers.cpp

#include <vector>

#include <iterator>

#include <string>

#include <ws2tcpip.h>

using namespace std;

struct SocketData

{

    SOCKET socket;

    string ip;

    string port;

};

// A helper function to simplify the main part.

template<class T>

ostream& operator<<(ostream& os, const vector<T>& v)

{

    copy(v.begin(), v.end(), ostream\_iterator<T>(os, " "));

    return os;

}

void CorrectClosing(SOCKET socket)

{

    // Sets the console attributes to default values.

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

    closesocket(socket);

    // Should be called at the end of the program and be paired with WSAStartup.

    WSACleanup();

}

void getIP(sockaddr socketInfo, char\* ip)

{

    sockaddr\_in\* sin = reinterpret\_cast<sockaddr\_in\*>(&socketInfo);

    inet\_ntop(AF\_INET, &sin->sin\_addr, ip, INET\_ADDRSTRLEN);

}

Скриншоты выполнения программы



