#include <iostream>

#include "Winsock2.h" // Заголовок WS2\_32.dll

#pragma comment(lib, "ws2\_32.lib") // экспорт WS2\_32.dll

#pragma warning(suppress : 4996)

#pragma warning(disable : 4996)

#pragma warning(disable : 4996)

using namespace std;

string GetErrorMsgText(int code); // обработчик стандартных ошибок WS2\_32.LIB

string SetErrorMsgText(string msgText, int code);

int main()

{

setlocale(LC\_ALL, "rus");

int i;

int t;

int t2;

try {

WSADATA wsaData; //Версия WS

SOCKET serverSocket; // серверный сокет

if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0) // инициализация библиотеки

throw SetErrorMsgText("Startup:", WSAGetLastError());

//ориентирован на передачу потоком

if ((serverSocket = socket(AF\_INET, SOCK\_STREAM, NULL)) == INVALID\_SOCKET) // создать сокет

throw SetErrorMsgText("socket:", WSAGetLastError());

SOCKADDR\_IN serv; // параметры сокета sS

serv.sin\_family = AF\_INET; // используется IP-адресация

serv.sin\_port = htons(2000); // порт 2000 (Для преобразования номера порта в формат TCP/IP)

serv.sin\_addr.s\_addr = INADDR\_ANY; // любой собственный IP - адрес

if (bind(serverSocket, (LPSOCKADDR)&serv, sizeof(serv)) == SOCKET\_ERROR) // установка параметров serv для сокета serverSocket

/\*Функция связывает дескриптор сокета и структуру SOCKADDR\_IN,

которая предназначена для хранения параметров сокета\*/

throw SetErrorMsgText("bind:", WSAGetLastError());

Данный код выполняет привязку сокета к локальному адресу и порту с помощью функции bind(). Аргумент serverSocket - это дескриптор сокета, который был создан и инициализирован ранее с помощью функции socket(). Аргумент serv - это указатель на структуру SOCKADDR\_IN, которая содержит локальный адрес и порт, к которому нужно привязать сокет.

Если функция bind() возвращает SOCKET\_ERROR, то значит произошла ошибка при выполнении операции привязки сокета к адресу и порту. В этом случае, с помощью функции SetErrorMsgText() создается строка с сообщением об ошибке, которая будет брошена как исключение.

Таким образом, данный код проверяет результат выполнения операции привязки сокета к адресу и порту, и в случае ошибки выбрасывает исключение с сообщением об ошибке.

if (SOCKET\_ERROR == listen(serverSocket, 2)) // установка сокета сервера в режим прослушивания (2 параметр - макс. длина очереди)

{

cout << "Listen: " << WSAGetLastError << endl;

}

Данный код выполняет вызов функции listen(), которая устанавливает серверный сокет в режим прослушивания входящих соединений.

Первый аргумент функции listen() - это дескриптор серверного сокета, который был создан и инициализирован ранее с помощью функции socket(). Второй аргумент функции listen() - это размер очереди ожидающих соединений.

Если функция listen() возвращает SOCKET\_ERROR, то это означает, что произошла ошибка при вызове функции. В данном случае, код проверяет, что функция listen() вернула SOCKET\_ERROR с помощью проверки if (SOCKET\_ERROR == listen(serverSocket, 2)). Если это условие истинно, то выводится сообщение об ошибке с помощью функции WSAGetLastError(), которая возвращает код последней ошибки Winsock.

Таким образом, данный код проверяет результат выполнения функции listen() и выводит сообщение об ошибке в случае неудачи.

cout << "Сервер начал прослушку!" << endl;

SOCKET clientSocket; // сокет для обмена данными с клиентом

SOCKADDR\_IN clnt; // параметры сокета клиента

memset(&clnt, 0, sizeof(clnt)); // обнулить память

int lclnt = sizeof(clnt); // размер SOCKADDR\_IN

while (true) {

i = 1;

// Функция accept создает канал на стороне сервера и иоздает сокет для обмена данными по этому каналу

if ((clientSocket = accept(serverSocket, (sockaddr\*)&clnt, &lclnt)) == INVALID\_SOCKET)

throw SetErrorMsgText("accept:", WSAGetLastError());

Данный код выполняет вызов функции accept(), которая принимает входящее соединение на серверном сокете, установленном в режим прослушивания с помощью функции listen().

Первый аргумент функции accept() - это дескриптор серверного сокета, который был создан и инициализирован ранее с помощью функции socket() и установлен в режим прослушивания с помощью функции listen(). Второй аргумент - это указатель на структуру sockaddr, в которую будет сохранен адрес клиента, подключившегося к серверу. Третий аргумент - это указатель на переменную int, которая содержит размер структуры sockaddr.

Функция accept() возвращает новый сокет (клиентский сокет), который будет использоваться для обмена данными с клиентом. Если функция accept() возвращает INVALID\_SOCKET, то это означает, что произошла ошибка при вызове функции. В данном случае, код проверяет, что функция accept() вернула INVALID\_SOCKET с помощью проверки if ((clientSocket = accept(serverSocket, (sockaddr\*)&clnt, &lclnt)) == INVALID\_SOCKET). Если это условие истинно, то выводится сообщение об ошибке с помощью функции WSAGetLastError(), которая возвращает код последней ошибки Winsock.

Таким образом, данный код проверяет результат выполнения функции accept() и выводит сообщение об ошибке в случае неудачи. Если функция accept() успешно выполнена, то в переменную clientSocket сохраняется новый клиентский сокет, который будет использоваться для обмена данными с клиентом.

// Начал замер!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

t = clock();

cout << "Соединение с клиентом успешно!" << endl;

cout << "Адрес клиента: " << inet\_ntoa(clnt.sin\_addr) << ":" << htons(clnt.sin\_port) << "\n\n\n\n\n";

char ibuf[50], //буфер ввода

obuf[50] = "sever: принято "; //буфер вывода

int libuf = 0, //количество принятых байт

lobuf = 0;

while (true) {

//остановка до вызова connect, результат - сокет для связи

if ((libuf = recv(clientSocket, ibuf, sizeof(ibuf), NULL)) == SOCKET\_ERROR)

{

cout << "Recv: " << WSAGetLastError() << endl;

break;

}

Данный код является частью бесконечного цикла, который ожидает получения данных от клиента через сокет clientSocket.

Функция recv используется для приема данных из сокета, она возвращает количество байт, которые были фактически прочитаны, и сохраняет их в буфер ibuf.

Если при вызове recv произошла ошибка, то функция WSAGetLastError вернет код ошибки, который будет выведен на экран вместе с сообщением "Recv:".

В данном случае, если прием данных завершается с ошибкой, то выполнение цикла прерывается с помощью break.

\_itoa\_s(lobuf, obuf + sizeof("sever: принято ") - 1, 10, 10);

cout << "Принято сообщение " << i << ": " << ibuf << endl;

i++;

if ((lobuf = send(clientSocket, ibuf, strlen(ibuf) + 1, NULL)) == SOCKET\_ERROR)

{

cout << "Send: " << WSAGetLastError() << endl;

break;

}

Этот код отправляет сообщение через сокет clientSocket на сервер. Функция send() возвращает количество байт, отправленных успешно, или значение SOCKET\_ERROR в случае ошибки.

В данном коде, если функция send() вернула SOCKET\_ERROR, то она не смогла успешно отправить сообщение, и программа переходит в блок catch с помощью вызова функции SetErrorMsgText(). Затем выводится сообщение об ошибке в консоль. Если send() возвращает количество отправленных байт, программа продолжает работу.

}

t2 = clock();

cout << "\n\nКлиент отключился" << endl;

cout << "Затраченное время: " << (((float)t2) - t) /\*/ CLOCKS\_PER\_SEC\*/ << " миллисекунд";

}

if (closesocket(serverSocket) == SOCKET\_ERROR) // закрыть сокет

throw SetErrorMsgText("closesocket:", WSAGetLastError());

Данный код используется для закрытия серверного сокета. Функция closesocket() закрывает дескриптор сокета и освобождает все ресурсы, связанные с этим сокетом. Если выполнение функции завершается со значением SOCKET\_ERROR, то вызывается функция SetErrorMsgText(), которая генерирует сообщение об ошибке с помощью функции WSAGetLastError().

if (WSACleanup() == SOCKET\_ERROR) // завершить работу с библиотекой

throw SetErrorMsgText("Cleanup:", WSAGetLastError());

}

catch (string errorMsgText) {

{ cout << endl << "WSAGetLastError: " << errorMsgText; }

}

return 0;

}

string GetErrorMsgText(int code) // Функция позволяет получить сообщение ошибки

{

string msgText;

switch (code)

{

case WSAEINTR: return "WSAEINTR: Работа функции прервана ";

case WSAEACCES: return "WSAEACCES: Разрешение отвергнуто";

case WSAEFAULT: return "WSAEFAULT: Ошибочный адрес";

case WSAEINVAL: return "WSAEINVAL: Ошибка в аргументе";

case WSAEMFILE: return "WSAEMFILE: Слишком много файлов открыто";

case WSAEWOULDBLOCK: return "WSAEWOULDBLOCK: Ресурс временно недоступен";

case WSAEINPROGRESS: return "WSAEINPROGRESS: Операция в процессе развития";

case WSAEALREADY: return "WSAEALREADY: Операция уже выполняется";

case WSAENOTSOCK: return "WSAENOTSOCK: Сокет задан неправильно";

case WSAEDESTADDRREQ: return "WSAEDESTADDRREQ: Требуется адрес расположения";

case WSAEMSGSIZE: return "WSAEMSGSIZE: Сообщение слишком длинное";

case WSAEPROTOTYPE: return "WSAEPROTOTYPE: Неправильный тип протокола для сокета";

case WSAENOPROTOOPT: return "WSAENOPROTOOPT: Ошибка в опции протокола";

case WSAEPROTONOSUPPORT: return "WSAEPROTONOSUPPORT: Протокол не поддерживается";

case WSAESOCKTNOSUPPORT: return "WSAESOCKTNOSUPPORT: Тип сокета не поддерживается";

case WSAEOPNOTSUPP: return "WSAEOPNOTSUPP: Операция не поддерживается";

case WSAEPFNOSUPPORT: return "WSAEPFNOSUPPORT: Тип протоколов не поддерживается";

case WSAEAFNOSUPPORT: return "WSAEAFNOSUPPORT: Тип адресов не поддерживается протоколом";

case WSAEADDRINUSE: return "WSAEADDRINUSE: Адрес уже используется";

case WSAEADDRNOTAVAIL: return "WSAEADDRNOTAVAIL: Запрошенный адрес не может быть использован";

case WSAENETDOWN: return "WSAENETDOWN: Сеть отключена";

case WSAENETUNREACH: return "WSAENETUNREACH: Сеть не достижима";

case WSAENETRESET: return "WSAENETRESET: Сеть разорвала соединение";

case WSAECONNABORTED: return "WSAECONNABORTED: Программный отказ связи";

case WSAECONNRESET: return "WSAECONNRESET: Связь восстановлена";

case WSAENOBUFS: return "WSAENOBUFS: Не хватает памяти для буферов";

case WSAEISCONN: return "WSAEISCONN: Сокет уже подключен";

case WSAENOTCONN: return "WSAENOTCONN: Сокет не подключен";

case WSAESHUTDOWN: return "WSAESHUTDOWN: Нельзя выполнить send : сокет завершил работу";

case WSAETIMEDOUT: return "WSAETIMEDOUT: Закончился отведенный интервал времени";

case WSAECONNREFUSED: return "WSAECONNREFUSED: Соединение отклонено";

case WSAEHOSTDOWN: return "WSAEHOSTDOWN: Хост в неработоспособном состоянии";

case WSAEHOSTUNREACH: return "WSAEHOSTUNREACH: Нет маршрута для хоста";

case WSAEPROCLIM: return "WSAEPROCLIM: Слишком много процессов";

case WSASYSNOTREADY: return "WSASYSNOTREADY: Сеть не доступна";

case WSAVERNOTSUPPORTED: return "WSAVERNOTSUPPORTED: Данная версия недоступна";

case WSANOTINITIALISED: return "WSANOTINITIALISED: Не выполнена инициализация WS2\_32.DLL";

case WSAEDISCON: return "WSAEDISCON: Выполняется отключение";

case WSATYPE\_NOT\_FOUND: return "WSATYPE\_NOT\_FOUND: Класс не найден";

case WSAHOST\_NOT\_FOUND: return "WSAHOST\_NOT\_FOUND: Хост не найден";

case WSATRY\_AGAIN: return "WSATRY\_AGAIN: Неавторизированный хост не найден";

case WSANO\_RECOVERY: return "WSANO\_RECOVERY: Неопределенная ошибка";

case WSANO\_DATA: return "WSANO\_DATA: Нет записи запрошенного типа";

case WSA\_INVALID\_HANDLE: return "WSA\_INVALID\_HANDLE: Указанный дескриптор события с ошибкой";

case WSA\_INVALID\_PARAMETER: return "WSA\_INVALID\_PARAMETER: Один или более параметров с ошибкой";

case WSA\_IO\_INCOMPLETE: return "WSA\_IO\_INCOMPLETE: Объект ввода - вывода не в сигнальном состоянии";

case WSA\_IO\_PENDING: return "WSA\_IO\_PENDING: Операция завершится позже";

case WSA\_NOT\_ENOUGH\_MEMORY: return "WSA\_NOT\_ENOUGH\_MEMORY: Не достаточно памяти";

case WSA\_OPERATION\_ABORTED: return "WSA\_OPERATION\_ABORTED: Операция отвергнута";

case WSASYSCALLFAILURE: return "WSASYSCALLFAILURE: Аварийное завершение системного вызова";

default: return "\*\*ERROR\*\*";

}

//TODO: добавить коды

//return msgText;

}

string SetErrorMsgText(string msgText, int code) // Функция возвращает сообщение ошибки

{

return msgText + GetErrorMsgText(code);

}