Network Multiplayer Games

# Contents

[Contents 2](#_Toc5175159)

[Introduction 3](#_Toc5175160)

[1. TCP 4](#_Toc5175161)

[2. UDP 5](#_Toc5175162)

[3. Running the application 6](#_Toc5175163)

[3.1 TCP Functionality 7](#_Toc5175164)

[3.1.1 Client Input (TCP): 7](#_Toc5175165)

[3.1.2 Server Received (TCP): 9](#_Toc5175166)

[3.2 UDP Functionality 11](#_Toc5175167)

[3.2.1 Receiving on UDP 11](#_Toc5175168)

[3.2.2 Sending on UDP 12](#_Toc5175169)

[4. Socket Class 13](#_Toc5175170)

[5. Byte Order Packing Classes 14](#_Toc5175171)

[References 15](#_Toc5175172)

# Introduction

This report will look at the applications made as part of the Network Multiplayer Games module assignment: ‘TCPServer’ and ‘TCPClient’. Detailed explanations of each the source code files will be looked at, including implementation captures and improvements for the future of each class. Tests will be performed to ensure expected outcomes are correct. This application is designed to pack, unpack, receive and send messages on both the UDP and TCP protocols using the winsock2 library for windows. This would enable the creation of many types of games using networking features.

# 1. TCP

TCP is used to ensure a reliable connection between hosts. This requires a larger header file compared to UDP and the connection to be setup first before any data is sent by an application [1].

The process of setting up a connection is called a three-way handshake. The process involves [1]:

1. Host A allocates resources to hold connection state
2. Host A sends a packet with SYN flag set to Host B, requesting a connection be established
   1. A random sequence number is chosen by Host A and is sent in the same packet
3. Host B to allocate resources to hold connection state
4. Host B to reply with a packet that has both SYN and ACK flags set
   1. Host A sequence number set to received sequence number + 1
   2. A random sequence number is chosen by Host B and is sent in the same packet
5. Host A responds with a packet that has it ACK flag set
   1. Host B sequence number set to received sequence number + 1

If any packet is not received after a timeout length, the packet is sent again until an acknowledgement packet has been received [1].

TCP allows appending of a payload to a packet; the packets sequence number is increased by the number of bytes of the payload [1].

TCP ensures packets are received in the correct order. When a packet is received before it was expected, the connection (transport layer) can either drop the packet or store in its local buffer. There will be no ACK packet sent until all packets have been received in order. This is due to the buffer having a limited size and the packet possibly being dropped later if this buffer is used up, requiring the packet to be sent again [1].

To limit packets having to be sent multiple times, TCP has a flow control feature that allows a host to state how much buffer space is left. This allows the connecting host to only send that amount of data or less before expecting an acknowledgement of data being received [1].

TCP also makes use of the Nagle’s Algorithm, which enables the transport layer to accumulate data in a buffer before sending to a host. This ensures efficient bandwidth control. Data can be force sent by closing the connection, but in turn this means the connection will need to be setup again when new data is needing to be sent [1].

Closing a connection on TCP sets a packets Fin flag. The FIN flag notifies the destination host all required data has been sent, and all pending data in the send buffer will be sent. The connection will remain open until a packet is received with the ACK has been received from the destination host [1].

# 2. UDP

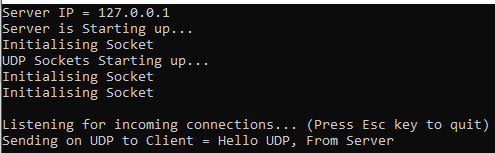
UDP is the opposite to the TCP connection type. UDP gives no support for reliability and does not verify packets have been received by the destination host [1]. When a UDP packet is sent, it is forgotten about by the host. If a packet must be successfully received it can either be sent across a TCP connection or the programmer would need to implement a system to monitor packets at the application layer level [1]. It is also good practice to keep UDP packets smaller than the MTU to allow single packet transmission on this protocol. UDP packets that are larger than the MTU need to be ordered and pieced together again at the application level, implemented by the programmer [1].

# 3. Running the application

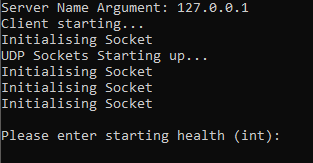
TCPServer Application needs to be launched before TCPClient is launched. Both applications launch in a similar way by initialising both a client and listen socket on both TCP and UDP:

Server:

The server blocks while listening for connections on TCP, this is intentional as a turn-based game is the suggested target. Multithreading with a queuing system of connection buffers would need to be implemented if the game is required to be action / real-time based mechanics.



Client:



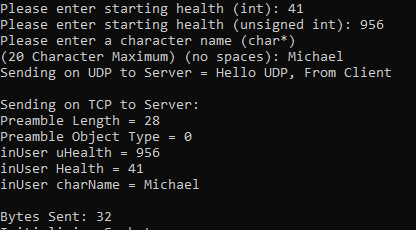
When both server and client have started up, the next stage is to follow the instructions on the client application to input each data type in the ‘User’ class and see the messages reflected on the server application.

## 3.1 TCP Functionality

This section will take a brief high-level overview of what steps are taken through the applications TCP functionality. TCP allows for user input to be dynamically accepted during runtime.

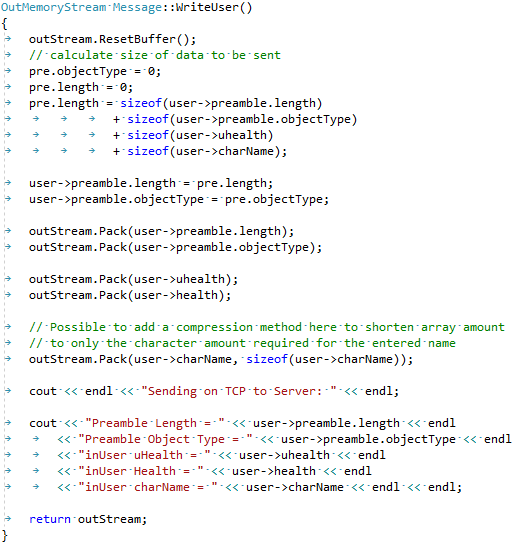
### 3.1.1 Client Input (TCP):

1. Enter Values using guidelines in brackets for each type required.



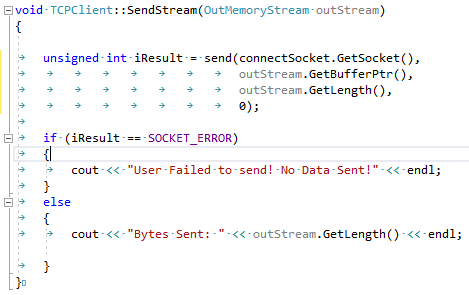
1. Client then Packs ‘User’ object using ‘Message::WriteUser’ Function to invoke ‘OutMemoryStream’ packing class.

Message::WriteUser() Function:



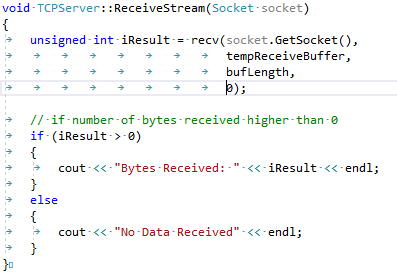
1. TCPClient class sends stream of data to TCPServer.

TCPClient::SendStream()



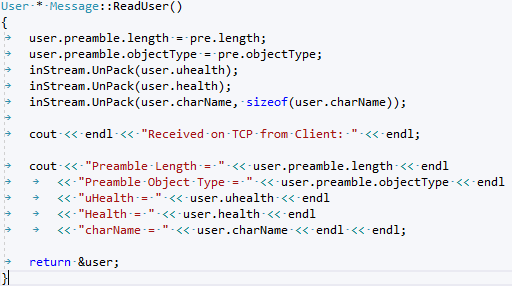
### 3.1.2 Server Received (TCP):

1. Server receives from listen socket.
2. TCPServer receives stream of data from TCPClient using TCPServer::RecieveStream():

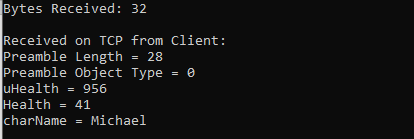


1. Server then Unpacks ‘User’ object using ‘Message::ReadUser’ Function to invoke ‘InMemoryStream’ unpacking class and displays result to console.

Message::ReadUser() Function:

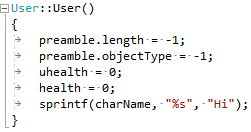


Server User object after steps above:



Preamble length is for the ‘User’ Object specifically. ‘User’ object has a size of 28 Bytes (int, unsigned int, char[20]) and ‘PreAmble’ object has a size of 4 bytes (unsigned short x2). Therefore, 32 Bytes of data was sent across from TCPClient to TCPServer.

The default constructor for the ‘User’ object on the server is set as uHealth = -1, Health = -1 and charName = Hi. The above steps show that the correct data has been received and applied on the server side correctly



## UDP Functionality

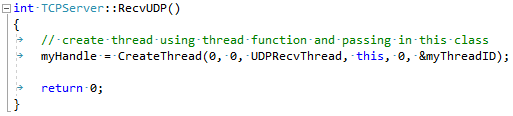
This section will take a brief high-level overview of what steps are taken through the applications UDP functionality. UDP is currently hardcoded and does not allow user manipulation from console, all edits must be made within the code files themselves currently. Both server and client have functionality enabling the sending and receiving of messages from each other across the UDP protocol.

### Receiving on UDP

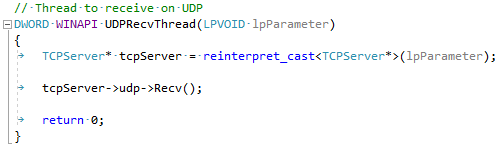
The recvfrom function for UDP is a blocking function and is therefore implemented in a thread of its own to enable TCP turn-based connections to continue without UDP blocking the whole application until a message is received.

Multithreaded of recvfrom function on UDP:

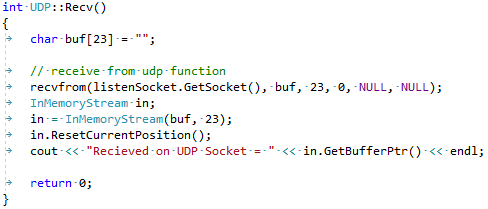
1. Call RecvUDP() function in server loop
2. RecvUDP() function creates thread and passes in current class ‘this’ (TCPServer / TCPClient)



1. Run thread Function body



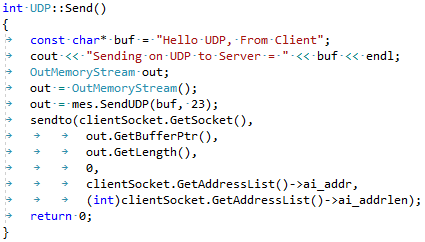
1. Run UDP::Recv()



### Sending on UDP

The sendto function is not a blocking function so does not include any multithreading functionality.

1. Call UDP::Send() function in server loop
2. UDP::Send() packs up message using OutMemoryStream and sends using sendto UDP function



# 4. Socket Class

The Socket class handles all functionality to do with both UDP and TCP socket creation. Multiple constructors are available to create a socket using this wrapper class. Once parameters are passed into the socket class via its constructor, a local object socket is initialised and addrinfo is populated and stored within the class. This socket can be accessed easily outside of its object safely with the GetSocket() function. Both TCP and UDP use the functionality of this socket wrapper class to create and or bind its listen and client sockets.

Improvements

To improve on the socket class, the send and recv functions should be included as easy to access functions that accepts multiple types of data and provides type safety automatically.

Socket could also be broken down into multiple objects such as: TCP Server, TCP Client, UDP Server, UDP Client with a higher parent virtual class for shared functions which would hold the name Socket.

# 5. Byte Order Packing Classes

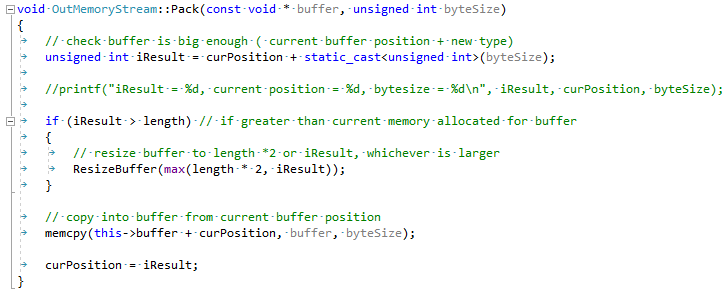
There are two packing classes used in the applications OutMemoryStream and InMemoryStream. Both classes are similar and could be made more streamlined as explained in the improvements section below. OutMemoryStream is used to convert data from host byte order to network byte order and then store in a buffer ready for transmission. InMemoryStream class does the opposite and converts the data received from network byte order to host byte order.

Buffer management is controlled by a check to ensure data received is not bigger than the current buffer allocated size. When the case arises that the data is larger than the current buffer size, the buffer is extended using memcpy (either by 2\* current buffer size, or to current data + new data size, whichever is larger) [1].

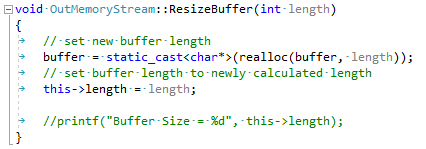
Improvements

Both Byte Order classes currently have multiple parts that are the same, these parts can be placed in a separate parent class and shared between OutMemoryStream and InMemoryStream.

Pack with buffer size check:



Resize buffer function:



# 6. Message Class

The Message class controls calling of packing streams and allows a place to reconstruct incoming data. A Preamble is used to determine object type and object length expected.

Improvements

Message currently only works with data from the TCP Socket, this could be expanded to expect data from UDP also. A queue of incoming data could also be implemented for data that could be sent in multiple parts through UDP and TCP. Multithreading could also mean a queue system would need to be controlled and monitored to ensure no shared data is being overwritten and corrupted.

# APPENDIX A – TCPServer Code

## TCPServer.h

#pragma once

// disable CRT Secure Warning Messages

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <iostream>

#include <WinSock2.h> // winsock2 library

#include <string>

#include "Socket.h"

#include "User.h"

#include "InMemoryStream.h"

#include "Message.h"

#include "UDP.h"

using std::cout; using std::endl; using std::string;

const unsigned int bufLength = 1470;

class TCPServer

{

public:

TCPServer();

TCPServer(int);

TCPServer(int, int, int, int);

~TCPServer();

int InitServer();

void CreateClientSocket();

int GetIntialisationError();

int RecvUDP();

int Listen();

int ListenLoop();

UDP\* udp;

private:

int clientWin = 0;

int port;

bool exit = false;

int portNumber;

int addressFamily;

int packetType;

int protocolType;

WSADATA wsaData;

string recvbuf;

char recvbufchar[32];

int iResult;

int recvbuflen;

int iRcvdBytes;

int intialisationError;

char tempReceiveBuffer[bufLength];

User\* one;

// Sockets

Socket listenSocket;

Socket clientSocket;

Message mes;

DWORD myThreadID;

HANDLE myHandle;

CRITICAL\_SECTION g\_cs;

void PrintError(string);

void ReceiveStream(Socket socket);

};

## TCPServer.cpp

////////////////////////////////////////////////////////////

//// Program Main Starting Point ////

////////////////////////////////////////////////////////////

#include "TCPServer.h"

// Thread to receive on UDP

DWORD WINAPI UDPRecvThread(LPVOID lpParameter)

{

TCPServer\* tcpServer = reinterpret\_cast<TCPServer\*>(lpParameter);

tcpServer->udp->Recv();

return 0;

}

// TCPServer Constructor

TCPServer::TCPServer()

{

myHandle = nullptr;

port = -1;

addressFamily = -1;

packetType = -1;

protocolType = -1;

InitServer();

}

// TCPServer overloaded constructor (Port Number)

TCPServer::TCPServer(int portNumber)

{

myHandle = nullptr;

port = portNumber;

addressFamily = -1;

packetType = -1;

protocolType = -1;

InitServer();

}

// TCPServer overloaded constructor (Port Number, Address Family, Packet Type, Protocol Type)

TCPServer::TCPServer(int portNumber, int addressFamily, int packetType, int protocolType)

{

myHandle = nullptr;

port = portNumber;

this->addressFamily = addressFamily;

this->packetType = packetType;

this->protocolType = protocolType;

InitServer();

}

TCPServer::~TCPServer()

{

if (myHandle != nullptr)

{

CloseHandle(myHandle);

}

// cleanup resources on server termination

WSACleanup();

}

// Sever initialisation function

int TCPServer::InitServer()

{

// initialise starting variables

recvbuflen = bufLength;

iRcvdBytes = 0;

// (MAKEWORD(Major version, Minor version), LPWSADATA = setup information from windows)

// WSA function return zero if working correctly

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

// check result of WSAStartup function

if (iResult == 0)

{

cout << "Server is Starting up..." << endl;

listenSocket = Socket(port, addressFamily, packetType, protocolType);

if (listenSocket.GetSocket() == INVALID\_SOCKET)

{

PrintError("server socket invalid");

return -1;

}

else

{

// setup listen socket

udp = new UDP(27019, AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

one = new User();

// set socket options, bind socket and freeaddressinfo

listenSocket.InitialiseListenSocketBind();

}

// Start server loop

ListenLoop();

return 0;

}

else

{

// Output last error encountered

PrintError("Server unable to start.");

return -1;

}

}

void TCPServer::CreateClientSocket()

{

clientSocket = Socket(listenSocket);

}

int TCPServer::GetIntialisationError()

{

return intialisationError;

}

int TCPServer::RecvUDP()

{

// create thread using thread function and passing in this class

myHandle = CreateThread(0, 0, UDPRecvThread, this, 0, &myThreadID);

return 0;

}

int TCPServer::Listen()

{

return 0;

}

int TCPServer::ListenLoop()

{

cout << endl << "Listening for incoming connections... (Press Esc key to quit)" << endl;

int i = 0;

// Loop while escape key has not been pressed

while (i < 10)

{

i++;

// Server must listen on listensocket for incoming connections

// (created listen socket, maximum connections allowed)

if (listen(listenSocket.GetSocket(), SOMAXCONN) == SOCKET\_ERROR)

{

PrintError("Listening failed");

}

// Create temporary socket to accept connections

CreateClientSocket();

// Accept client connection

// blocking happens here

// should be on a thread of its own if not a turn based game

clientSocket = accept(listenSocket.GetSocket(), NULL, NULL);

// check if accepting client connection was successful

if (clientSocket.GetSocket() == INVALID\_SOCKET)

{

PrintError("Accepting client connection failed");

}

// Send UDP data

udp->Send();

ReceiveStream(clientSocket.GetSocket());

mes = Message(tempReceiveBuffer, bufLength);

mes.ObjectReadSelector();

one = mes.GetUser();

//printf("buffer contents = %s", tempReceiveBuffer[mes.GetNewBufferPosition()]);

// Shutdown connection

clientSocket.ShutdownSocket();

// Free up socket - Close Socket

clientSocket.CloseSocket();

// Function to spawn thread for UDP receive

RecvUDP();

// if escape key is hit, set loop bool to true

if (GetAsyncKeyState(VK\_ESCAPE))

{

exit = true;

}

}

return 0;

}

// WSA Error printing function

void TCPServer::PrintError(string input)

{

// Output Server Port number and last error encountered

cout << input << endl

<< "Server Port Number: " << port << endl

<< "WSA Error Code: " << WSAGetLastError() << endl;

intialisationError = -1;

listenSocket.CloseSocket();

WSACleanup();

}

void TCPServer::ReceiveStream(Socket socket)

{

unsigned int iResult = recv(socket.GetSocket(), tempReceiveBuffer, bufLength, 0);

// if number of bytes received higher than 0

if (iResult > 0)

{

cout << "Bytes Received: " << iResult << endl;

}

else

{

cout << "No Data Received" << endl;

}

}

## Main.cpp

#pragma once

#include <iostream>

#include "TCPServer.h"

using std::cout; using std::endl;

int main(int argc, char\* argv[])

{

cout << "Server IP = 127.0.0.1" << endl;

TCPServer tcpServer = TCPServer(27015, AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

// Terminate Server if unable to intialise

if (tcpServer.GetIntialisationError() == -1)

{

return 1;

}

return 0;

}

# APPENDIX B – TCPClient Code

## TCPClient.h

#pragma once

// disable CRT Secure Warning Messages

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <iostream>

#include <WinSock2.h> // winsock2 library

#include <string>

#include <time.h>

#include "Socket.h"

#include "User.h"

#include "OutMemoryStream.h"

#include "Message.h"

#include "UDP.h"

using std::cout; using std::endl; using std::cin;

using std::string;

const unsigned int bufLength = 512;

class TCPClient

{

public:

TCPClient();

TCPClient(const char\*, int, int, int, int);

~TCPClient();

void InitClient();

void CreateConnectSocket(int addressFamily, int packetType, int protocolType);

void SetServerName(const char\* serverName);

string GetServerName();

int CheckIntialisationError();

int RecvUDP();

void Loop();

void PrintError(string input);

DWORD myThreadID;

HANDLE myHandle;

void SendStream(OutMemoryStream outStream);

UDP\* udp;

private:

int port;

bool exit = false;

WSADATA wsaData;

string recvbuf;

string sendbuf;

char sendbufchar[bufLength];

int iResult;

int recvbuflen;

int intialisationError;

string serverName;

Socket connectSocket;

int addressFamily;

int packetType;

int protocolType;

Message mes;

};

## TCPClient.cpp

#include "TCPClient.h"

DWORD WINAPI UDPRecvThread(LPVOID lpParameter)

{

TCPClient\* tcpClient = reinterpret\_cast<TCPClient\*>(lpParameter);

tcpClient->udp->Recv();

return 0;

}

TCPClient::TCPClient()

{

port = -1;

this->addressFamily = -1;

this->packetType = -1;

this->protocolType = -1;

serverName = "127.0.0.1";

}

TCPClient::TCPClient(const char\* serverName, int portNumber, int addressFamily, int packetType, int protocolType)

{

port = portNumber;

SetServerName(serverName);

this->addressFamily = addressFamily;

this->packetType = packetType;

this->protocolType = protocolType;

InitClient();

}

TCPClient::~TCPClient()

{

}

void TCPClient::InitClient()

{

// initialise starting variables

recvbuflen = bufLength;

// (MAKEWORD(Major version, Minor version), LPWSADATA = setup information from windows)

// WSA function return zero if working correctly

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

// check result of WSAStartup function

if (iResult == 0)

{

cout << "Client starting..." << endl;

CreateConnectSocket(addressFamily, packetType, protocolType);

// setup listen socket

udp = new UDP(27019, AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

Loop();

}

else

{

// Output last error encountered

PrintError("Client unable to start.");

}

}

void TCPClient::CreateConnectSocket(int addressFamily, int packetType, int protocolType)

{

// Create a listening socket to be able to connect to server

connectSocket = Socket(serverName, port, addressFamily, packetType, protocolType);

if (connectSocket.GetSocket() == INVALID\_SOCKET)

{

PrintError("connecting socket invalid");

}

}

void TCPClient::SetServerName(const char\* input)

{

if (input != NULL)

{

string tempString(input);

serverName = tempString;

}

}

string TCPClient::GetServerName()

{

return serverName;

}

int TCPClient::CheckIntialisationError()

{

return intialisationError;

}

int TCPClient::RecvUDP()

{

// create thread using thread function and passing in this class

myHandle = CreateThread(0, 0, UDPRecvThread, this, 0, &myThreadID);

return 0;

}

void TCPClient::Loop()

{

User\* one;

one = new User();

mes.SetUser(one);

int i = 0;

// Loop while escape key has not been pressed

while (i < 10)

{

i++;

// Create a socket to connect with

CreateConnectSocket(addressFamily, packetType, protocolType);

// Attempt to connect to an address until one succeeds

for (connectSocket.SetPtrToAddressList(); connectSocket.GetPtr() != NULL; connectSocket.SetPtr(connectSocket.GetPtr()->ai\_next))

{

// Connect to server.

iResult = connect(connectSocket.GetSocket(), connectSocket.GetPtr()->ai\_addr, (int)connectSocket.GetPtr()->ai\_addrlen);

if (iResult == SOCKET\_ERROR)

{

cout << "Unable to Connect to Server" << endl;

connectSocket.SetSocketInvalid();

connectSocket.CloseSocket();

exit = true;

continue;

}

break;

}

one->UserConsoleEntry();

// send udp message

udp->Send();

SendStream(mes.WriteUser());

// Function to spawn thread for UDP receive

RecvUDP();

// if escape key is hit, set loop bool to true

if (GetAsyncKeyState(VK\_ESCAPE))

{

exit = true;

}

connectSocket.FreeAddressInfo();

// free up resources

connectSocket.ShutdownSocket();

connectSocket.CloseSocket();

}

delete one;

}

// WSA Error printing function

void TCPClient::PrintError(string input)

{

// Output Server Port number and last error encountered

cout << input << endl

<< "Unable to connect to server: " << serverName << " " << endl

<< "WSA Error Code: " << WSAGetLastError() << endl;

intialisationError = -1;

WSACleanup();

}

void TCPClient::SendStream(OutMemoryStream outStream)

{

unsigned int iResult = send(connectSocket.GetSocket(), outStream.GetBufferPtr(), outStream.GetLength(), 0);

if (iResult == SOCKET\_ERROR)

{

cout << "User Failed to send! No Data Sent!" << endl;

}

else

{

cout << "Bytes Sent: " << outStream.GetLength() << endl;

}

}

## Main.cpp

#pragma once

#include <iostream>

#include "TCPClient.h"

#define \_CRT\_SECURE\_NO\_WARNINGS 1

using std::cout; using std::endl;

int main(int argc, char\* argv[])

{

TCPClient tcpClient;

if (argc < 2) // if user has not passed in an argument to application

{

cout << endl << "Usage: " << argv[0] << endl << "Server Name" << endl

<< "Using loopback address 127.0.0.1" << endl;

// set server name to loopback address

tcpClient = TCPClient("127.0.0.1", 27015, AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

}

else

{

// set server name to argument passed in by user

cout << "Server Name Argument: " << argv[1] << endl;

// Create Client

tcpClient = TCPClient(argv[1], 27015, AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

cout << "Server Name Set to: " << tcpClient.GetServerName() << endl;

}

// Terminate Server if unable to intialise

if (tcpClient.CheckIntialisationError() == -1)

{

return 1;

}

return 0;

}

# APPENDIX C – Shared

## InMemoryStream.h

#pragma once

#include <iostream>

#include <cstdlib>

#include <algorithm>

#include <string>

#include <WinSock2.h> // winsock2 library

using std::free; using std::memcpy;

using std::cout; using std::endl;

using std::string;

class InMemoryStream

{

public:

InMemoryStream();

InMemoryStream(char\* buffer, unsigned int byteSize);

~InMemoryStream();

unsigned int GetRemainingDataSize();

const unsigned int GetLength();

const char\* GetBufferPtr();

void ResetCurrentPosition();

void UnPack(void\* buffer, unsigned int byteSize);

// multiple type packs using main pack function

void UnPack(int &buffer);

void UnPack(unsigned int &buffer);

void UnPack(unsigned long &buffer);

void UnPack(unsigned short &buffer);

private:

char\* buffer;

unsigned int length;

unsigned int curPosition;

};

## InMemoryStream.cpp

// Memory stream to unpack data from network byte order to host byte order

#include "InMemoryStream.h"

InMemoryStream::InMemoryStream()

{

this->buffer = nullptr;

length = 0;

curPosition = 0;

}

InMemoryStream::InMemoryStream(char\* buffer, unsigned int byteSize)

{

this->buffer = buffer;

length = byteSize;

curPosition = 0;

}

InMemoryStream::~InMemoryStream()

{

}

unsigned int InMemoryStream::GetRemainingDataSize()

{

return length - curPosition;

}

// Return length of buffer

const unsigned int InMemoryStream::GetLength()

{

int temp = length - curPosition;

temp = length - temp;

return temp;

}

// Return pointer to buffer

const char \* InMemoryStream::GetBufferPtr()

{

return this->buffer;

}

// reset buffer current position pointer to start of buffer

void InMemoryStream::ResetCurrentPosition()

{

curPosition = 0;

}

// Default unpack function

void InMemoryStream::UnPack(void \* buffer, unsigned int byteSize)

{

// new position is current plus input bytesize

unsigned int iResult = curPosition + byteSize;

if (iResult > length)

{

cout << "Buffer Empty" << endl;

}

// memcpy old data and new data

memcpy(buffer, this->buffer + curPosition, byteSize);

curPosition = iResult;

}

// Unpack int using default unpack function

void InMemoryStream::UnPack(int &buffer)

{

UnPack(&buffer, sizeof(buffer));

buffer = (int)ntohl((unsigned long)buffer);

}

// Unpack unsigned int using default unpack function

void InMemoryStream::UnPack(unsigned int &buffer)

{

UnPack(&buffer, sizeof(buffer));

buffer = (unsigned int)ntohl((unsigned long)buffer);

}

// Unpack unsigned long using default unpack function

void InMemoryStream::UnPack(unsigned long &buffer)

{

UnPack(&buffer, sizeof(buffer));

buffer = ntohl(buffer);

}

// Unpack short using default unpack function

void InMemoryStream::UnPack(unsigned short &buffer)

{

UnPack(&buffer, sizeof(buffer));

buffer = ntohs(buffer);

}

## Message.h

#pragma once

#include "PreAmble.h"

#include "OutMemoryStream.h"

#include "InMemoryStream.h"

#include "User.h"

#include "Socket.h"

class Message

{

public:

Message();

Message(char \* buffer, unsigned int length);

void ObjectReadSelector();

void ObjectWriteSelector();

~Message();

OutMemoryStream GetOutStream();

InMemoryStream GetInStream();

User\* GetUser();

void SetUser(User\*);

unsigned int GetNewBufferPosition();

void ReadPreamble();

OutMemoryStream WriteUser();

User\* ReadUser();

OutMemoryStream SendUDP(const char \* send, unsigned int size);

InMemoryStream ReceiveUDP(char \* send, unsigned int size);

private:

unsigned int newBufferPosition;

char\* buffer;

unsigned int bufferLength;

PreAmble pre;

User \*user;

OutMemoryStream outStream;

InMemoryStream inStream;

};

## Message.cpp

#include "Message.h"

Message::Message()

{

this->buffer = buffer;

bufferLength = 0;

user = nullptr;

}

Message::Message(char \* buffer, unsigned int length)

{

this->buffer = buffer;

bufferLength = length;

inStream = InMemoryStream(buffer, length);

user = nullptr;

}

// Select which object to write or read from using object type in preamble

void Message::ObjectReadSelector()

{

// Check object type and length

ReadPreamble();

switch (pre.objectType)

{

case 0: // Is User object

ReadUser();

if (bufferLength > 28)

{

newBufferPosition = 28;

}

break;

default:

break;

}

}

void Message::ObjectWriteSelector()

{

// Check object type and length

ReadPreamble();

switch (pre.objectType)

{

case 0: // Is User object

WriteUser();

break;

default:

break;

}

}

Message::~Message()

{

}

OutMemoryStream Message::GetOutStream()

{

return outStream;

}

InMemoryStream Message::GetInStream()

{

return inStream;

}

User\* Message::GetUser()

{

return user;

}

void Message::SetUser(User \*user)

{

this->user = user;

}

unsigned int Message::GetNewBufferPosition()

{

return newBufferPosition;

}

void Message::ReadPreamble()

{

inStream.UnPack(pre.length);

inStream.UnPack(pre.objectType);

}

OutMemoryStream Message::WriteUser()

{

outStream.ResetBuffer();

// calculate size of data to be sent

pre.objectType = 0;

pre.length = 0;

pre.length = sizeof(user->preamble.length)

+ sizeof(user->preamble.objectType)

+ sizeof(user->uhealth)

+ sizeof(user->charName);

user->preamble.length = pre.length;

user->preamble.objectType = pre.objectType;

outStream.Pack(user->preamble.length);

outStream.Pack(user->preamble.objectType);

outStream.Pack(user->uhealth);

outStream.Pack(user->health);

// Possible to add a compression method here to shorten array amount

// to only the character amount required for the entered name

outStream.Pack(user->charName, sizeof(user->charName));

cout << endl << "Sending on TCP to Server: " << endl;

cout << "Preamble Length = " << user->preamble.length << endl

<< "Preamble Object Type = " << user->preamble.objectType << endl

<< "inUser uHealth = " << user->uhealth << endl

<< "inUser Health = " << user->health << endl

<< "inUser charName = " << user->charName << endl << endl;

return outStream;

}

User \* Message::ReadUser()

{

user->preamble.length = pre.length;

user->preamble.objectType = pre.objectType;

inStream.UnPack(user->uhealth);

inStream.UnPack(user->health);

inStream.UnPack(user->charName, sizeof(user->charName));

cout << endl << "Received on TCP from Server: " << endl;

cout << "Preamble Length = " << user->preamble.length << endl

<< "Preamble Object Type = " << user->preamble.objectType << endl

<< "uHealth = " << user->uhealth << endl

<< "Health = " << user->health << endl

<< "charName = " << user->charName << endl << endl;

return user;

}

OutMemoryStream Message::SendUDP(const char\* send, unsigned int size)

{

outStream = OutMemoryStream();

outStream.Pack(send, size);

outStream.ResetBuffer();

// cout << "Outstream UDP = " << outStream.GetBufferPtr() << endl;

return outStream;

}

InMemoryStream Message::ReceiveUDP(char\* send, unsigned int size)

{

// pack message

inStream = InMemoryStream(send, size);

// cout << "Instream UDP = " << send << endl;

return inStream;

}

## OutMemoryStream.h

#pragma once

// static\_cast<>, reference from - J. Glazer and S. Madhav, Multiplayer game programming. Pearson Education, Inc, 2016.

#include <cstdlib>

#include <algorithm>

#include <string>

#include <WinSock2.h> // winsock2 library

#include "PreAmble.h"

using std::free; using std::memcpy; using std::realloc;

using std::max;

using std::string;

class OutMemoryStream

{

public:

OutMemoryStream();

~OutMemoryStream();

// construct OutMemoryStream to allow c++ primitive data types without size needing to be passed

// Main pack function

// also facilitates char\* directly

void Pack(const void\* buffer, unsigned int byteSize);

// multiple type packs using main pack function

void Pack(int buffer);

void Pack(unsigned int buffer);

void Pack(unsigned long buffer);

void Pack(unsigned short buffer);

unsigned int size;

// get buffer length

const unsigned int GetLength();

char\* GetBuffer();

// Get pointer to buffer

const char\* GetBufferPtr();

void Trim();

void ResetBuffer();

private:

char\* buffer;

// buffer length

unsigned int length;

// current position in buffer

unsigned curPosition;

// increase buffer size if not enough space remaining in current buffer

void ResizeBuffer(int length);

};

## OutMemoryStream.cpp

// Memory stream to pack data from host byte order to network byte order

#include "OutMemoryStream.h"

OutMemoryStream::OutMemoryStream()

{

buffer = nullptr;

length = 0;

curPosition = 0;

ResizeBuffer(32);

}

OutMemoryStream::~OutMemoryStream()

{

//free(buffer);

}

// use this for char / char\*

void OutMemoryStream::Pack(const void \* buffer, unsigned int byteSize)

{

// check buffer is big enough ( current buffer position + new type)

unsigned int iResult = curPosition + static\_cast<unsigned int>(byteSize);

// printf("iResult = %d, current position = %d, bytesize = %d\n", iResult, curPosition, byteSize);

if (iResult > length) // if greater than current memory allocated for buffer

{

// resize buffer to length \*2 or iResult, whichever is larger

ResizeBuffer(max(length \* 2, iResult));

}

// copy into buffer from current buffer position

memcpy(this->buffer + curPosition, buffer, byteSize);

curPosition = iResult;

}

// Type int

void OutMemoryStream::Pack(int buffer)

{

buffer = (int)htonl(buffer);

Pack(&buffer, sizeof(buffer));

}

// Type unsigned int

void OutMemoryStream::Pack(unsigned int buffer)

{

buffer = (unsigned int)htonl(buffer);

Pack(&buffer, sizeof(buffer));

}

// Type long

void OutMemoryStream::Pack(unsigned long buffer)

{

buffer = htonl(buffer);

Pack(&buffer, sizeof(buffer));

}

// Type short

void OutMemoryStream::Pack(unsigned short buffer)

{

buffer = htons(buffer);

Pack(&buffer, sizeof(buffer));

}

// Return length of buffer

const unsigned int OutMemoryStream::GetLength()

{

return length;

}

// Return pointer to buffer

char \* OutMemoryStream::GetBuffer()

{

return this->buffer;

}

const char \* OutMemoryStream::GetBufferPtr()

{

return this->buffer;

}

// resize buffer to only send required byte amount

void OutMemoryStream::Trim()

{

ResizeBuffer(curPosition);

}

// reset buffer current position pointer to start of buffer

void OutMemoryStream::ResetBuffer()

{

curPosition = 0;

}

void OutMemoryStream::ResizeBuffer(int length)

{

// set new buffer length

buffer = static\_cast<char\*>(realloc(buffer, length));

// set buffer length to newly calculated length

this->length = length;

//printf("Buffer Size = %d", this->length);

}

## Socket.h

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <iostream>

#include <string>

#include <WinSock2.h> // winsock2 library

#include <ws2tcpip.h> // Used for address translation

using std::cout; using std::endl; using std::string;

class Socket

{

public:

Socket();

Socket(string, int, int, int, int);

Socket(int, int, int, int);

Socket(SOCKET);

~Socket();

// Socket Initialisation Function

int InitialiseSocket();

void InitialiseListenSocketBind();

int ResolveAddress();

int ResolveAddress(const char \* serverName);

int ResolveAddress(string serverName);

void SetSocket(SOCKET);

void SetSocketInvalid();

void SetSocketOptions();

void BindSocket();

void Connect();

SOCKET GetSocket();

addrinfo\* GetAddressList();

addrinfo\* GetPtr();

void SetPtrToAddressList();

void SetPtr(addrinfo \* inPtr);

addrinfo\* GetHints();

void FreeAddressInfo();

void ShutdownSocket();

void CloseSocket();

private:

int port = -1;

string serverName;

int addressFamily;

int packetType;

int protocolType;

struct addrinfo\* addressList;

struct addrinfo\* ptr;

struct addrinfo hints;

SOCKET currentSocket;

void PrintError(string input);

};

## Socket.cpp

#include "Socket.h"

Socket::Socket()

{

// set to Invalid socket, enables error checking against socket

SetSocketInvalid();

addressList = nullptr;

serverName= "";

addressFamily = -1;

packetType = -1;

protocolType = -1;

}

Socket::~Socket()

{

}

Socket::Socket(string serverName, int portNumber, int addressFamily, int packetType, int protocolType)

{

// set to Invalid socket, enables error checking against socket

SetSocketInvalid();

addressList = nullptr;

port = portNumber;

ptr = nullptr;

this->serverName = serverName;

this->addressFamily = addressFamily;

this->packetType = packetType;

this->protocolType = protocolType;

InitialiseSocket();

}

Socket::Socket(int portNumber, int addressFamily, int packetType, int protocolType)

{

// set to Invalid socket, enables error checking against socket

SetSocketInvalid();

addressList = nullptr;

port = portNumber;

ptr = nullptr;

this->addressFamily = addressFamily;

this->packetType = packetType;

this->protocolType = protocolType;

InitialiseSocket();

}

Socket::Socket(SOCKET input)

{

SetSocket(input);

}

// Initialise using a string type server name - client use

int Socket::InitialiseSocket()

{

cout << "Initialising Socket" << endl;

ZeroMemory(&hints, sizeof(hints));

// Address Family, AF-UNSPEC, AF\_INET = IPv4, AF\_INET6 = IPv6

hints.ai\_family = addressFamily;

// Packet Type, SOCK\_Stream = TCP, SOCK\_DGRAM = UDP,

// SOCK\_RAW = application layer crafted packet,

// SOCK\_SEQPACKET = packets might be required to be read in full upon receipt

hints.ai\_socktype = packetType;

// Protocol, Transport or Network layer protocols

// IPPROTO\_UDP - requires SOCK\_DGRAM

// IPPROTO\_TCP - requires SOCK\_STREAM

// IPPROTO\_IP / 0 - uses default protocol for type set

hints.ai\_protocol = protocolType;

// AI\_PASSIVE flag - address structure intended to be binded

hints.ai\_flags = AI\_PASSIVE;

if (serverName != "")

{

ResolveAddress(serverName);

}

else

{

ResolveAddress();

}

if (addressList->ai\_protocol != NULL)

{

currentSocket = socket(addressList->ai\_family, addressList->ai\_socktype, addressList->ai\_protocol);

}

if (currentSocket == INVALID\_SOCKET)

{

PrintError("Socket creation failed - Closing Application");

SetSocketInvalid();

closesocket(currentSocket);

return -1;

}

else

{

return 0;

}

}

void Socket::InitialiseListenSocketBind()

{

SetSocketOptions();

BindSocket();

FreeAddressInfo();

}

// Resolve address - use this for listen sockets

int Socket::ResolveAddress()

{

if (getaddrinfo(NULL, std::to\_string(port).c\_str(), &hints, &addressList) != 0)

{

return -1;

}

else

{

return 0;

}

}

// Resolve Address - use this for client sockets

int Socket::ResolveAddress(const char\* serverName)

{

if (getaddrinfo(serverName, std::to\_string(port).c\_str(), &hints, &addressList) != 0)

{

return -1;

}

else

{

return 0;

}

}

// Resolve Address - use this for client sockets

int Socket::ResolveAddress(string serverName)

{

if (getaddrinfo(serverName.c\_str(), std::to\_string(port).c\_str(), &hints, &addressList) != 0)

{

return -1;

}

else

{

return 0;

}

}

void Socket::SetSocket(SOCKET input)

{

this->currentSocket = input;

}

void Socket::SetSocketInvalid()

{

// set to Invalid socket, enables error checking against socket

currentSocket = INVALID\_SOCKET;

}

void Socket::SetSocketOptions()

{

BOOL bOptVal = TRUE;

int bOptLen = sizeof(BOOL);

if (setsockopt(currentSocket, SOL\_SOCKET, SO\_REUSEADDR, (char\*)&bOptVal, bOptLen) == SOCKET\_ERROR)

{

PrintError("Socket creation failed");

CloseSocket();

WSACleanup();

}

}

void Socket::BindSocket()

{

// Check binding of socket was completed correctly

if (bind(currentSocket, addressList->ai\_addr, (int)addressList->ai\_addrlen) == SOCKET\_ERROR)

{

PrintError("Socket Binding Failed");

}

}

void Socket::Connect()

{

if (connect(currentSocket, ptr->ai\_addr, (int)ptr->ai\_addrlen) == SOCKET\_ERROR)

{

PrintError("Unable to Connect to Socket!");

//cout << "Unable to Connect to Server" << endl;

}

}

SOCKET Socket::GetSocket()

{

return currentSocket;

}

addrinfo\* Socket::GetAddressList()

{

return addressList;

}

addrinfo\* Socket::GetPtr()

{

return ptr;

}

void Socket::SetPtrToAddressList()

{

ptr = addressList;

}

void Socket::SetPtr(addrinfo\* inPtr)

{

ptr = inPtr;

}

addrinfo\* Socket::GetHints()

{

return &hints;

}

void Socket::FreeAddressInfo()

{

if (addressList != nullptr)

{

// clear addressList memory

freeaddrinfo(addressList);

}

}

// Shutdown and wait for ACK from client

void Socket::ShutdownSocket()

{

// Shutdown connection, SD\_SEND sets no more sending allowed

if (shutdown(currentSocket, SD\_SEND) == SOCKET\_ERROR)

{

PrintError("Shutdown Failed");

}

}

void Socket::CloseSocket()

{

closesocket(currentSocket);

}

void Socket::PrintError(string input)

{

cout << input << endl

<< "WSA Error Code: " << WSAGetLastError() << endl;

CloseSocket();

SetSocketInvalid();

WSACleanup();

}

## UDP.h

#pragma once

// disable CRT Secure Warning Messages

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <iostream>

#include <WinSock2.h> // winsock2 library

#include <string>

#include "Socket.h"

#include "InMemoryStream.h"

#include "OutMemoryStream.h"

#include "Message.h"

class UDP

{

public:

UDP();

UDP(int portNumber, int addressFamily, int packetType, int protocolType);

~UDP();

const int bufLength = 1470;

int InitUDP();

int Send();

char buffer[1470];

int Recv();

int iResult;

WSADATA wsaData;

void PrintError(string input);

int i = 0;

int port;

int portNumber;

int addressFamily;

int packetType;

int protocolType;

Socket listenSocket;

Socket clientSocket;

Message mes;

};

## UDP.cpp

/// Controls UDP Functions for sending and receiving

#include "UDP.h"

UDP::UDP()

{

port = -1;

addressFamily = -1;

packetType = -1;

protocolType = -1;

}

UDP::UDP(int portNumber, int addressFamily, int packetType, int protocolType)

{

port = portNumber;

this->addressFamily = addressFamily;

this->packetType = packetType;

this->protocolType = protocolType;

InitUDP();

}

UDP::~UDP()

{

}

int UDP::InitUDP()

{

// (MAKEWORD(Major version, Minor version), LPWSADATA = setup information from windows)

// WSA function return zero if working correctly

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

// check result of WSAStartup function

if (iResult == 0)

{

cout << "UDP Sockets Starting up..." << endl;

listenSocket = Socket("127.0.0.1", 27018, addressFamily, packetType, protocolType);

clientSocket = Socket("127.0.0.1", 27019, addressFamily, packetType, protocolType);

if (listenSocket.GetSocket() == INVALID\_SOCKET)

{

PrintError("server socket invalid");

return -1;

}

else

{

// set socket options, bind socket and freeaddressinfo

listenSocket.InitialiseListenSocketBind();

}

}

return 0;

}

// Send on UDP

int UDP::Send()

{

const char\* buf = "Hello UDP, From Client";

cout << "Sending on UDP to Server = " << buf << endl;

OutMemoryStream out;

out = OutMemoryStream();

out = mes.SendUDP(buf, 23);

sendto(clientSocket.GetSocket(), out.GetBufferPtr(), out.GetLength(), 0, clientSocket.GetAddressList()->ai\_addr, (int)clientSocket.GetAddressList()->ai\_addrlen);

return 0;

}

// Receive on UDP

int UDP::Recv()

{

char buf[23] = "";

recvfrom(listenSocket.GetSocket(), buf, 23, 0, NULL, NULL);

InMemoryStream in;

in = InMemoryStream(buf, 23);

in.ResetCurrentPosition();

cout << "Recieved on UDP Socket = " << in.GetBufferPtr() << endl;

return 0;

}

// WSA Error printing function

void UDP::PrintError(string input)

{

// Output Server Port number and last error encountered

cout << input << endl

<< "Server Port Number: " << port << endl

<< "WSA Error Code: " << WSAGetLastError() << endl;

//intialisationError = -1;

listenSocket.CloseSocket();

WSACleanup();

}

## User.h

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <iostream>

#include <string>

#include "PreAmble.h"

using std::cout; using std::endl; using std::cin;

class User

{

public:

User();

User(int health, char charName);

~User();

PreAmble GetPreAmble();

unsigned int GetuHealth();

int GetHealth();

char\* GetCharName();

void SetPreAmble(PreAmble pre);

void SetuHealth(unsigned int uHealth);

void SetHealth(int health);

void SetCharName(char\* name);

void UserConsoleEntry();

// Should be private below, for future revision

PreAmble preamble;

unsigned int uhealth;

int health;

char charName[20];

};

## User.cpp

// Example user class

#include "User.h"

User::User()

{

preamble.length = -1;

preamble.objectType = -1;

uhealth = 10;

health = 23;

sprintf(charName, "%s", "Client Testing...");

}

User::User(int health, char charName)

{

preamble.length = -1;

preamble.objectType = -1;

uhealth = -1;

this->health = health;

sprintf(this->charName, "%s", charName);

}

User::~User()

{

}

PreAmble User::GetPreAmble()

{

return preamble;

}

unsigned int User::GetuHealth()

{

return uhealth;

}

int User::GetHealth()

{

return health;

}

char \* User::GetCharName()

{

return charName;

}

void User::SetPreAmble(PreAmble pre)

{

preamble = pre;

}

void User::SetuHealth(unsigned int uHealth)

{

this->uhealth = uhealth;

}

void User::SetHealth(int health)

{

this->health = health;

}

void User::SetCharName(char \* name)

{

sprintf(this->charName, "%s", name);

}

void User::UserConsoleEntry()

{

cout << endl << "Please enter starting health (int): ";

cin >> health;

cout << "Please enter starting health (unsigned int): ";

cin >> uhealth;

cout << "Please enter a character name (char\*)" << endl

<< "(20 Character Maximum) (no spaces): ";

cin >> charName;

}

## PreAmble.h

#pragma once

// Type:

// 0 = User

struct PreAmble

{

unsigned short length;

unsigned short objectType;

};

## Position.h

#pragma once

class Position

{

public:

Position();

Position(int x, int y);

~Position();

private:

int x;

int y;

};

## Position.cpp

#include "Position.h"

Position::Position()

{

x = -1;

y = -1;

}

Position::Position(int x, int y)

{

x = this->x;

y = this->y;

}

Position::~Position()

{

}

# References

[1]J. Glazer and S. Madhav, *Multiplayer game programming*. Crawfordsville, Indiana: Addison-Wesley, 2016, pp. 41-52.