Documentation for C++ chat

author: Eroshin Vitaliy

Content

1 Overview

1.1 Sockets Idea

These are just Adapters over standard **sys/socket.h** and other **C** libraries. Their interface is more **C++** friendly, it does not import such functions as **read**, **send** in global scope. However, these classes have features that enough for my project, and may be expanded if needed.

1.2 Objects Idea

Object is a key element, that is being transmitted through the socket. Of cause, C++ structure cannot be sent, so it is being encoded to the byte array via **Encoder** interface. On the other hand, **Encoder** also implements interface that decodes bytes and returns C++ structure.

1.3 Client Idea

All the main logic is stored in the **Client** class. It manages **Socket** (networking, information exchange), **Encoder** (Object interaction), **UserInterface** (*Facade* over standard C++ IO functions) and **ObjectTree** (Client object storage system)

```
// Example of usage

StrEncoder encoder;
Client client(encoder);
client.session();
```

1.4 Storage Idea

Storage is an interface for storing, adding and modifying **Object**s, user and chat information.

1.5 Server Idea

All the main logic is stored in the **Server** class. It manages **Sockets** (networking, information exchange), **Encoder** (Object interaction) and **Storage** (Object, chat and user data storing).

1.6 Exception policy

Exceptions are prohibited, mainly because they negatively affects performance and code readability. For more information, you can refer to the **Google C++ Style Guide**.

2 Socket

2.1 Socket()

It is default constructor, that does not initialize anything. Can be used in case if you want to setup all the class fields by yourself.

2.2 Socket(int port)

This constructor creates socket on given port and initializes descriptor field. Also sets up socket address, which is just typedef of **C** type. Other options such as *domain* or *stream* are hard-coded, mostly because I do not need to change anything for my usage. YAGNI.

2.3 ~ Socket()

Destructor, that only closes socket descriptor.

2.4 int bind()

Implements default **C** bind function. Used to bind server port.

2.5 int setSocketOption(int option, int value)

Implements default **C** setsockopt function, but in easier **C++** style way.

```
option  - option name
value   - option value
```

See setsockopt documentation for more details.

2.6 int listen(int backlog)

Implements default **C** listen function. Used in server port.

```
backlog - descriptor for backlogging.
```

2.7 void setAddress(int port)

Sets up socket address with port and hard-coded constants. Used in constructor.

```
port - port you want to set to the address.
```

2.8 Socket accept()

Accepts connection and returns new peer Socket.

2.9 string getIpAddress()

Returns IP address of socket. Mainly used for server logging.

2.10 void send(string message)

Sends message through the socket. Classic overloads such as char* instead of string are also available.

```
message - information you want to send through the socket.
```

2.11 string read()

Reads from the socket and returns string.

2.12 void getPeerName()

Refreshes socket address. Used in *getIpAddress*.

2.13 int getPort()

Returns port of the socket.

3 DescriptorSet

3.1 void set(int descriptor)

Implements standard **C** *FD_SET* macro. Adds descriptor to the set.

```
descriptor - file descriptor number
```

3.2 void clear()

Implements standard **C** *FD_ZERO* macro. Removes all the descriptors from the set.

3.3 bool count(int descriptor)

Implements standard **C** *FD_ISSET* macro. Returns TRUE if descriptor is in set, and FALSE if it is not.

```
descriptor - file descriptor number
```

3.4 fd_set* reference()

Returns pointer to *fd_set* (Which is DescriptorSet is based on)

4 Object

It does not have any methods - just fields. All the operations are being made by encoders or by user. Fields have their special types, that are defined in **src/include/types.hpp** and can be easily changed. Most of the fields can be uninitialized and would not be transmitted over the net (That makes chat to be extremely effective in terms of exchange speed and traffic consumption).

```
objectid_t id
```

Each object to *be stored* in the storage has to have its id to be easily referred in the future (message editing, user replies, etc...). Server commands or some other types of objects that does not need it can have their field uninitialized.

```
int timestamp
```

Another unnecessary field, that contains time related to object.

```
objectid_t parentId
```

Contains id of the parent.

```
userid_t author
```

Contains userid of the author.

```
code_t ret
```

Contains return code. Used for server callback on some users actions.

```
Type type
```

Enum type, that contains type of the object. It is necessary field, that contains information of how to decode the object. It is based on char, so it contains as little as 1 byte.

```
object_message_t message
```

Contains text of the message.

5 Encoder

Interface for interaction with Objects. Firstly, it defines **bytes** type as string (by default, because it is pretty comfortable to use). Each class that implements Encoder should contain two methods:

5.1 bytes encode(Object object)

Returns bytes, containing information about object.

5.2 Object decode(bytes bytes)

Returns object, decoded from bytes. Basically opposite method to the encode function.

6 StrEncoder

It is a very basic Encoder is used in the chat (easy to implement and debug).

6.1 char getTypeId(Object object)

Returns char value of the type of the object.

6.2 Object::Type fromTypeId(char id)

Opposite function to the **getTypeId**. Returns type enum from the it's char value.

6.3 bytes toBytes(T id)

Template function, that returns byte representation of the T object. Based on reinterpret_cast to the char*.

6.4 T fromBytes(bytes b)

Returns object of type T reinterpret_casted from the char*.

7 UserInterface

Facade class over standard **C/C++** input/output functions. It consists of 4 main substructures: *Output*, *Cursor*, *Keyboard* and *Input*.

7.1 Output

It contains **Window** substructure, that describes user window characteristics:

```
struct Window {
  size_t height;
  size_t width;
};
```

and the reference to the **std::ostream - out** (output stream, where app is to be displayed)

7.1.1 Output()

Default constructor. Just updates **Window** characteristics.

7.1.2 void flush()

Flushes the **out**.

7.2 Cursor

Contains cursor info

```
struct Position {
  size_t x = 0;
  size_t y = 0;
};
```

and **operator**== for comparing it.
It also has **Output** reference inside for cursor movements.

Each movement can be described by **Cursor::Direction**:

```
enum Direction {left, right, up, down};
```

7.2.1 Cursor(Output& output)

Constructor from the Output reference. Initializes Output& field and cursor position at (0, 0).

7.2.2 void move(Direction direction)

Move cursor by 1 unit in the given direction. Implemented by printing arrow symbol into **output.out**

7.2.3 Direction getDirection(output_char_t c)

Returns **Direction** corresponding to the given arrow symbol.

7.2.4 void moveTo(Position pos)

Higher level implementation of *void move(Direction)* function. Takes **Position** - coordinates of destination point and moves cursor to that point by sequential *void move(Direction)* calls.

7.3 Keyboard

Contains static key detecting predicates:

```
bool isTab(char x)
bool isEnter(char x)
bool isBackspace(char x)
bool isArrow(char x)
```

7.4 Input

Includes **Output** and **Keyboard** references and **Buffer** substructure.

Buffer consists of the left buffer (string) and right buffer (deque<char>). It also contains methods of interaction between right and left buffer, such as:

```
void moveLeft() - move one char from right to the left buffer.

void moveRight() - move one char from left to the right buffer.

void moveLeftAll() - move all the chars to the left buffer.

output_t getRight() - get representation of concatenation of buffers.
```

The only constructor is a **Input(Output& output, Keyboard& keyboard)** which initializes corresponding fields.

7.5 UserInterface()

Default constructor. Firstly initializes corresponding references and fields. After that, changes stty to raw mode to access extended IO abilities like so:

```
stty raw -echo
```

When stty is in raw mode, it allocates output space with **allocateSpace()**

7.6 ∼ UserInterface()

Desturctor. Sets stty mode to default - cooked echo.

```
stty cooked echo
```

7.7 void allocateSpace(size_t n)

Allocates **n** empty lines by printing "\n\r" symbols and moving cursor down. After that, calls *flush* and moves cursor to initial position.

7.8 void print(char c)

Prints the char and updates cursor position.

7.9 void print(string c)

Optional arguments:

type	argument	default value
Position	pivot	cursor.position
Position	size	Position(1, c.size())

Creates frame - rectangle with corner coordinates of **pivot** and **pivot** + **size**. After that puts the **c** into this frame.

7.10 string input(Position pivot, Position size)

Creates text input frame in rectangle with corner coordinates of **pivot** and **pivot** + **size**. Implemented as by-char input, with special key handling. By now it supports only ASCII symbols input (issue may be fixed by using std::wstring instead of std::string). Special keys are being handled by such functions as:

```
void processInputTab(Position pivot, Position end)
```

Basically, it does nothing for now. TODO: conditional space print.

```
void processInputArrow(Position pivot, Position end)
```

When left or right arrow is pressed, **Buffer** moves char between left and right buffer. If up or down arrow is pressed, function calls **void scrollChatUp()** / **void scrollChatDown()**, that are not yet implemented.

```
void processInputBackspace(Position pivot, Position end, Position size)
```

If backspace is pressed, while right buffer is empty and left is not, it just prints "\b" and pop_back() the left buffer. And if user decide to remove character from the middle, printed characters are need to be refreshed with
void refreshInputBuffer(Position pivot, Position size)

7.11 askForm(Position pivot, Position size, string text)

Just another form of **string input(Position pivot, Position size)**, but prints pretty invite (text) at the left of input frame.

7.12 void clearWindow()

Clears the window by printing empty symbols.

8 ObjectTree

Structure, with the list of the Objects and head iterator to the pivot object. For now, it does not erase old Objects and just inserts them to the list.

8.1 ObjectTree()

Default constructor. Puts empty fake Object to the list and sets up iterator to it.

8.2 void insert(string text)

Inserts Object with given text.

9 Client

Class, where all the main logic is concentrated. Contains references to or entities of all the necessary objects. Also has a **Status** field, which is enum:

```
enum Status {
    offline, online, connecting, authentification, failed
};
```

9.1 Client(Encoder& encoder)

Sets up references and entities. By default, status if offline.

9.2 void setupAddress()

Asks user to input IP address and port. Correct IP address is "localhost" or "X.Y.Z.W" , where X, Y, Z, W - are numbers between 0 and 255. Correct port is a number. If address:port is incorrect, it will ask a user to enter them again.
When address and port are correct, it will change the **Status** to the *connecting*.

9.3 int connectToHost()

Implements standard **C** connect(descriptor, address) function.

9.4 int auth()

Asks user to input login and password. Then sends it to the server. After that displays and returns server callback. If authentication is successful, it returns **0** else **-1**

9.5 void initializeGUI()

Clears the window and prints input invite.

9.6 void refreshMessages()

Clears the output and print the messages from the **ObjectTree**.

9.7 void sendText(string text)

Creates **Object** of type **text** with the given text and encodes it to the string. After that, sends it to the server.

9.8 void sendCommand(string text)

Creates **Object** of type **command** with the given text and encodes it to the string. After that, sends it to the server.

9.9 int connect()

Sets up address with **setAddress()**, then starts **showBackground** thread while connecting. If 2 seconds passed and server was not found, it prints "Connection failed" and returns **-1**. If server was found, it prints "Connected!" , clears the window and returns **0**;

9.10 void showBackground(std::atomic<bool> connecting)

While connecting is true, it prints fancy "Connecting..."message.

9.11 void listen()

Contains **atomic<bool> run** which is true while program is running
and **atomic<bool> update**, which is true if **refreshMessages()** is requested.
Starts 3 threads: **readUserInput**, **readServer**, **refreshOutput**.

9.12 void parseMessage(string message)

Splits given message by separate lines.

9.13 void readServer(atomic update, atomic run)

Thread, that listens to the server receives the messages. Then it decodes message to the **Object** calls **parseMessage()** function and requests **refreshMessages()**, by storing TRUE to update atomic.

9.14 void readUserInput(atomic update, atomic run)

Thread, that listens to users input and then handles it. If input is "/quit" it stores FALSE to the **run** and stops the program. If input starts with "/" it creates and sends *command* Object to the server. Else, it creates *text* Object and sends to the server.

9.15 void refreshOutput(atomic update, atomic run)

Thread, that calls *refreshMessages()* if needed by loading update atomic. If *refreshMessages()* is recently called, it waits for some time before calling it again. This helps to avoid glitches when refreshMessages() are called too frequently.

9.16 int session()

Firstly calls **connect()**, then **auth()** and then **initializeGUI()** and **listen()**.

10 Storage

10.1 int getUser(login_t login, password_t password)

Function that returns:

-1, if there is no user with given login.

-2, if there is such user, but password is wrong.
0, if login and password are correct.

10.2 addUser(login_t login, password_t password)

Function that creates new **User** and returns:
-1, if there is a user with such login.
userid, if user has been successfully created.

10.3 int createChat(userid_t creator)

Function that creates new **Chat** and returns:
-1, if creator does not exist.
chatid, if chat has been successfully created.

10.4 int inviteToChat(selfId, target, chat)

Where

argument	**type**	**description**
selfId	userid_t	id of user who invites.
target	userid_t	id of user who is being invited
chat	chatid_t	id of the chat where target is being invited

Function tries to invite a target to the chat and returns:
-1, if there is no such chat.
-2, if selfId is not a member of the chat.
-3, if selfId is not a user.
-4, if target is not a user.
0, if target has been successfully invited to the chat.

10.5 chatid_t getChat(userid_t selfId)

Function that returns:
0, if user is not in the chat.
chatid, if user is in chat with chatid.

10.6 User& getUserReference(userid_t id)

Returns reference to the **User** object for given userid.

10.7 setUserChat(userid_t id, chatid_t chat)

Function that tries to set current chatid to the user and returns:
-1, if there is no given chat.
-2, if there is no given user in the chat members.
0, if the chat is successfully set.

10.8 vector<userid_t>& getUserFriends(userid_t id)

Function, that returns vector of userids, who are the friends of **id**.

10.9 vector<chatid_t>& getUserChats(userid_t id)

Function, that returns vector of chatids, where **id** is in members list.

10.10 int addFriend(userid_t selfId, userid_t target)

Function, that tries to add target to selfId friendlist and returns:

-1, if the target is not a user

0, if target has been added to user's friendlist.

11 Server

Contains general **Socket**, **DescriptorSet**, references to **Storage** and **Encoder**, list of **Connection**s and map of handlers.

11.1 Connection

Structure, that contains all the information about connection:

```
Socket* socket;     // pointer to peer socket
userid_t user{};    // user id of peer
chatid_t chat{};    // chat id of peer
```

It also contains **Status**, which is

```
enum Status {
  unauthorized,
  inmenu,
  inchat,
  inprofile
};
```

The only constructor is **Connection(Socket* socket)**, which opens connection on given socket.

11.2 Server(int port, Storage& storage, Encoder& encoder)

Constructor, which initializes given references, makes general **Socket**, sets socket options, binds the port and listens. If everything went well, it initializes handlers, but calling **initHandlers()**

11.3 void loop()

Endless loop, where the process of information exchange is happening. On every iteration, it:
manages **DescriptorSet**,
selects descriptor (**selectDescriptor()**),
accepts connections (**acceptConnection()**),
removes connections (**removeConnection()**),
calls **parseQuery()** for received queries.

11.4 void acceptConnection()

Creates peer **Socket**, displays information about received connection and adds **Connection** to the list.

11.5 void selectDescriptor()

Resets **DescriptorSet**, gets max descriptor from **Connection**s list and calls standard **C** *select* function.

11.6 void removeConnection(Connection peer)

Prints information about peer, calls its **Socket** destructor.

11.7 void parseQuery(string query, Connection& user)

Decodes received query into **Object**. Calls other parsers depend on Object type: *addMessage, parseAuthData* or *parseCommand*

11.8 void parseAuthData(Object object, Connection& user)

Separates login and password, asks the **Storage**.

If password is wrong, sends **2** returnCode to the client.

If there is no such user, it creates new user with given data, sends **1** returnCode to the peer and sets **Connection::Status** to *inmenu*.

If login and password are correct, it sends back **0** and sets **Connection::Status** to *inmenu*.

11.9 void parseCommand(Object object, Connection& user)

Parses the command and creates callback **Object**. If command is unknown, it does not do anything. And if command exists, it passes data and callback to corresponding handler from the map of handlers. After that, it sends callback to the user.

11.10 void addMessage(Object object, Connection& user)

Adds author nickname to the object text and sends it to other users, who are in the same chat.

12 Command Handlers

Server can parse a lot of various commands that have a various handlers. Making handlers look similar will give make it easy to extend command list. So, there are some rules how to add new handlers.

Firstly, handler signature should look like that:

```
void addFriendHandler(Object& callback, Connection& user, stringstream& ss)
```

where

argument	**description**
callback	is *Object*, where you should store your response
user	*Connection* reference, containing all the info about command caller.
ss	a stringstream, where you should take arguments of command from.

After that, you should add your handler to the map of handlers like so:

```
addHandler("/addfriend", &Server::addFriendHandler);
```

This method can be called in **initHandlers()**.