Greg Patterson (grpatter)
Eric Spencer (erspence)

General Architecture:
Try to bind the server to a socket and set it up to listen for commands.

Assuming the above, run a select call to monitor socket status. When select receives something, enter a loop checking each connected clients' status. If someone is sending data, receive it and process it (command or new client). If a new client, initialize some variables and set them up to be able to send commands. If a command, verify it as syntactically correct, and send it to another set of functions to be dealt with.

Client data is a structure containing information about a specific client (ip, nickname, fd, registration status, channels, and other addr information).
Channel data is a structure containing information about any existing channel on the server (name, active status, and member list).

Both of these structs should be kept in sync, and will hold the overall status of what is happening on the server at any given time.
The admin log provides a brief overview of activity and/or problems that occurred. It does NOT contain any specific user information other than connections and disconnection details. Messages are not logged due to privacy concerns that people may have.

Main
This is where the server "core" lives. Here, we take in the command line arguments and verify we have received the correct amount. If not, we ignore them and continue with defaults. We also set up our out_stream so that we can keep a running log for administrative purposes. Some variables are initialized, the important ones being a vector containing our ircClient data structs, a vector containing our ircChannels data structs, file descriptor sets to retain what connections are active, and socket information for server startup. After declaration, we begin to attempt to start the server. This process includes zero'ing out our fd_set's, setting some socket preferences (to allow multiple clients, not block, and connection allowances, for example), and then utilizing socket functions. These socket functions include getaddrinfo, bind, listen, select, read, and accept. The select call runs through the statuses on one (or more) of our sockets waiting to perform some kind of I/O action. When we do "select", we move on to the main for loop that runs through our file descriptor set looking for incoming data. When we find something usable (using FD_ISSET), we move on. We perform a check to determine if the server is at capacity, and if not we accept the data connection and analyze it. Here, we have two branches. One handles completely new connections (it reads information, adds them to the fd_sets, creates their clientData entry, and makes sure no errors occur) while the other branch handles data from already connected clients. This branch inspects the data to determine whether or not it is a valid command, and if so we throw it off to the handle_command function to be dealt with. If not (or if the user has not registered [/CONNECT] with us, we send a message telling them to of the issue). At the end of main, we close our out_stream (logging) and wave bye-bye to the world.

get_time function
The get_time function provides a time stamp for the log and the server. Information was gathered from the C++ library for this function. We attempted to break the time down in order to reformat it to the proper format but were unsuccessful in this implementation.

***get_in_addr**
The *get_in_addr function gets the socket address, either IPv4 or IPv6.

get_client_prefix
The get_client_prefix returns the nickname of the file descriptor in the data struct clientData.

Greg Patterson (grpatter)
Eric Spencer (erspence)

get_client_position
The get_client_position loops through the vectors of connected clients and returns the position in that vector. If the client that is being searched for is not present a -1 result is sent.

cConnect
The eConnect function is called from the handle command function in order to connect the client if they entered the correct information. Furthermore, it checks to see if the nickname requested is already in use before assigning them that particular nickname.

cMsg
The cMsg function is also called from the handle command function in order to send a message to a client or a channel. The buffered is parsed to check whether the message is to a channel (#) or to an individual. If the message is to a channel we browse the vector of channels to make sure the channel exists before sending the message, if it exists we scan the channel for all users present. The message is then sent to all users. If it is a private message we check to see if the user exist and then send the message.

cWho
The cWho function is also called from the handle command function in order to send a list of users in that channel. The vectors of channels are scanned to ensure the channel exists. If the channel exists the requester is sent everyone's nickname and IP address.

cJoin
The cJoin function is also called from the handle command function in order to join the requested channel. The vector of channels are scanned to determine whether it exists, if it exists the user is added. If the channel does not exist the channel is created and the user joins the channel.

cPart
The cPart function is also called from the handle command function in order to leave the specified channel. This function also helps when a user in a channel quits.

forcePart
The forcePart function helps the cQuit function. The forcePart is used when a member enters the command /QUIT. They are forced out of the channel and a message is sent to all users in that channel.

handle_command
The handle_command determines what command has been issued and where to send it among cConnect, cMsg, cWho, cJoin, cPart, and forcePart/cQuit.