Specifications

Main

Main class ended up being very simple. All it has as construction of ftp client and user prompts. The first two commands (username and password) start automatically and the rest of them will compare user's input (as a string) to command operations that are available.

Socket

Socket class contains all the necessary information to establish a socket as well as to communicate with the other end (send and receive command / write and read). You may notice that I have readBuffer, readData, and readFrom that are in fact similar in operation functions; yet, each one has different purpose in this program. Technically this is not something that I intended to do, but unfortunately if I would use only one of those function for all received messages from the server I would once in awhile get incorrect/incomplete message. Therefore, readBuffer() is used to read “Important Notice” (it polls while there is something to poll from. I could have used readData() for this purpose, but there is no way of checking code number prior to displaying the message), readFrom() function is used to read simple messages, such as confirmation of what is going to be performed or what was completed successfully / unsuccessfully (it polls only once, and the buffer size is limited to 1500 bytes; thus if message is larger than the buffer it wont read everything passed 1500 bytes), while  readData() is used to display “ls” command (it uses ostream reference input. We convert char buffer to string for every poll iteration and then we pass string to ostream instance; thus it will read as much information as server can offer us).

FTPClient

FTPClient is the heart of this program.

**clientSocket** object/instance of a socket object is a private member of FTPClient class that is used to connect / pass messages between client and server.

**dataSocket** is used to establish passive connection with the server.

This class also contains several methods that are similar in some way but they have different terminal commands in it.

* **FTPClient** constructor takes a URL passed as an argument from a client passes it to Socket constructor to create connection. It will try doing so until it gets message that starts with code # 220.

* **username** function takes a username from user input and passes it to FTP server. For as long as you get message that starts with 331 code, you good**.**
* **password** function takes a password from user and sends it to the server, if server

authenticates it send a message to user that starts with code 230; otherwise system prompts to enter a pass again. We loop till we see code 230.

* **sendMessage** function is used to communicate with socket’s writeTo() function. It takes two arguments: command and argument. if argument is empty, it sends command only, such as "ls".

Before we do that we create empty buffer (even though we can use cmd's buffer space) and copy cmd to that buffer space. Then we concatenate cmd and message (if message is empty we will end up adding just “\0”). Finally, we end our buffer data with "\r\n\0" to make sure that server sees end of a file.

* **syst** is used to send syst request / command.
* **getCode** is a helper function that gets first few bytes from a buffer (passed in) to get its code number and returns code as an int value, such as 250, 215, etc.
* **passiveOpen** function sends passive open request. We send message to server with PASV cmd and get new ip address from server that gets saved in buffer. We then send this buffer to calculatePASSV() method that takes buffer and converts what's inside to legitimate IP address.
* **calculatePASSV** function converts what's inside given buffer to an ip address. There are couple of ways to do that but I used vector to store individual numbers and then used stringstream to make an ip like char from it.
* **lsCMD** function is used to send LS command. Behaves similar to any other command, except that it uses readDate function from Socket class (recall that readDate function is used with this

command only). readData can read enormous amount of data. First, you call passiveOpen, then you send LIST command to the server. Right after you send message to the server you read from it and display message that was passed (normally it’s a message that starts with 150 code). I didn't check for code number though because if user enters different command (different than the one this program supports) it won't just work (nothing will happen compare to if you put wrong password or username).

* **cdCMD** function is similar to any other command. It prompts user to give directory address to move to and sends it along with CMD to the server. Going back would work too.
* **quitCMD** and **closeConnection** are very similar. The only difference is that for close command we send QUIT message using  dataSD, while for quit we actually send command using client’s socket descriptor.
* **getCMD** and **putCMD** functions are used to put file on FTP server or “download” from it. You first open a file descriptor and then you write to a file on the other end. The only “big” difference between the two is that one uses ofstream (:: out) while another uses ifstream (:: in). But overall you can see that first thing I do is setting a mode for a file and opening it. In case og GET command we open remore file, while in PUT we are opening local file. Then we use that file either to write its content to another file. When we done we close dataSocket and print message received from the server.

Assumptions/Limitations

It wasnt part of implementation requirements but professor Dimpsey asked to use fork() system call to create a thread child that would read messages from the server. I tried to use it with some of commands but ended up writing code without creating separate threads. First, once in awhile when you run “ls” command it would print “150 Opening ASCII mode data connection for LIST”

somewhere in a middle of a list, and second, I was able to get correct behavior without using it.

Also it wasn’t stated in function description, but you may notice that almost every function starts with declaring a buffer of some size and clearing it using bzero() command. Technically, I could have either declared more buffer as a class members or simple create one function that would allocate any number of buffers and clear it at the same time. Thus, I would save some lines of code as well as memory that is allocated in every function to those buffers.