Assignment 1: HTTP server - Design Document

Goal:
The goal of this program is to create a server that can send and receive files via http.
The server will have to parse through headers that are commands to either fetch a file
from the server to the client (GET) or or send a file from client to server (PUT).

Handling arguments:
Arguments will come in the form of headers. For example:

```
-------------------------------------------
| 1.                                      |
| PUT ABCDEFabcdef012345XYZxyz-mm HTTP/1.1 |
| Content-Length: 460                     |
| <------------Empty-Line---------------> |
|                                         |
| 2.                                      |
| GET ABCDEFarqdeXYZxyzf012345-ab HTTP/1.1 |
| <------------Empty-Line---------------> |
-------------------------------------------
```

-For each of the first lines consists of a command (PUT/GET)
 and the name of which the server will bind the data to (ABCDEF...).
 Note that the name the the data will be bound to must be 27 ASCII
        characters long and can only consist of [a-zA-Z-_]+
-Additionally certain headers will contain a 'Content-Length' which
 depicts how many bytes are to be read. In Example 1, read() will stop
 reading the data after 460 bytes has been read.
-Finally every header concludes with an empty line (\r\n).

-However before pasrsing header, we must set up the socket in order to
 receive the header.

```
-------------------------------------------------------------------------
| //First use gethostbyname() to sort out the server address as our  |
| //computers have multiple Internet addresses including a string:   |
| //"localhost"                                                      |
|                                                                    |
| //We then define sockaddr_in as the struct represents the address  |
| //format and have memcpy() input the server address handled by the |
| //prior method: gethostbyname() into the newly defined struct.     |
| //Additonally, we set the sin_family field to IPv4 (AF_INET) and   |
| //the sin_port field to its respected port number after converting |
| //it from host byte order to network byte order.                   |
| struct hostent *hent = gethostbyname(args[1]);                     |
| struct sockaddr_in address;                                        |
| memcpy(&address.sin_addr.s_addr, hent->h_addr, hent->h_length);    |
```

```
| address.sin_port = htons(PORT_NUMBER);                                  |
| address.sin_family = AF_INET;                                           |
|                                                                         |
| //Now we set up the socket:                                             |
| //First we create the socket with AF_INET (IPv4) as the domain,         |
| //SOCK_STREAM (TCP) as its type, and 0 (IP) for the protocol value.     |
| //Then we use setsockopt() to enable address reusing in the case of     |
| //the program crashing while the address was in use.                    |
| //Afterwards we will bind the socket to the address and port number     |
| //specified in the address variable and begin listening for any         |
| //requests with a backlog as 0 as the server will handle 1 client.      |
| int socket = socket(AF_INET, SOCK_STREAM, 0);                           |
| int optvl = 1;                                                          |
| setsockopt(socket, SOL_SOCKET, SO_REUSEADDR, &optvl, sizeof(optvl));    |
| bind(socket, (struct sockaddr *)&address, sizeof(address);              |
| listen(socket, 0);                                                      |
---------------------------------------------------------------------------
```

-Now after that we have set up our socket, we must begin to parse
 the header that the client will send once it is connected to the
 server.
-However, the header will be in chars instead of strings, so the
 solution will be to use strtok to seperate each line of chars and
 then store them in a vector so they can technically become "strings"

 E.g char[] = ['a', 'b', 'c'] but inserted into vector will be:
     [['a', 'b', 'c']] so vector[0] = ['a', 'b', 'c'] or 'abc'.

```
---------------------------------------------------------------------------
| bytes = read(fd, buff, buffsize);                                       |
| vector<char*> vect;                                                     |
| //Seperate by spaces and new lines                                      |
| char * delimited = strtok(buff, " \r\n");                               |
| while (delimited != NULL) {                                             |
|    vect.push_back(delimited);                                           |
|    delimited = strtok(NULL, " \r\n");                                   |
| }                                                                       |
| //Reset the buffer for good measure                                     |
| memset(buff, 0, buffsize);                                              |
---------------------------------------------------------------------------
```

-After parsing the header, we must determine the command, but luckily
 we have a vector to determine that for us.
-So we will have to see whether vector[0] will be a 'PUT' or 'Get',
 and if neither, will respond to the client with a 500 error.

```
---------------------------------------------------------------------------
| if(strcmp(vect[0] == 'PUT')) { Do a PUT response here}                  |
| else if(strcmp(vect[0] == 'GET')) { Do a GET responer here }            |
| else { write(fd, 500Error, strlen(500Error)); }                         |
```

-Additionally, we have to check if the file name is in the correct
 formatting (27 chars, only a-z, A-Z, 0-9, -, or _
 Else the program will have to respond with an Error 400 Bad Request.
-We will be using regex() and regex_match() to compare the filename
 with the regexes and see if the filename has any additional ones that
 are not specified.

```
| if(strlen(vect[1] != 27) {                                          |
|   write(fd, 400Error, strlen(400Error);                             |
| }                                                                   |
| if(!regex_match(vect[1], regex("[a-zA-Z0-9-_]+")) {                 |
|   write(fd, 400Error, strlen(400Error);                             |
| }                                                                   |
```

-Now we must create methods to repsond to the PUT and GET commands.
-Starting with PUT:
 - PUT will open a file and write new data into it
 - If the file does not exist, PUT will create the file and input data
   into the file.
 - Note that 200 OK means file edited and 201 OK means file created.
   So there must be a flag to keep in check if the file has been
   edited or created.

```
| //O_TRUNC is used to delete all previous data from the file so when|
| //rewriting the data, no old data is in the file                   |
| writeFD = open(trimFile, O_RDWR | O_TRUNC);                        |
| int flag = 0;                                                      |
| if(writeFD == -1) {                                                |
|   writeFD = open(trimFile, O_RDWR | O_TRUNC | O_CREAT);           |
|   flag = 1;                                                        |
| }                                                                  |
```

 _ Now we need to check if open() had returned an errno of 13 which
   is an error code for permission denied. If it did, then write to
   the client a 403 Forbidden and return.

```
| if(errno == 13) {                                                  |
|    write(fd, 403Forbidden, strlen(403Forbidden));                  |
|    return;                                                         |
```

 - Content length plays a role in this. If there is a content length
   specified in the header, then read up to the content length amount

of data. Else, read all of the data and continue reading from the client until the client manually closes.
- But first we must search through the header (stored in vector) to see if it contains a vector

```
| //int length is set to -1 to indicate no content-length.           |
| int length = -1;                                                   |
| for(iterator it = vect.begin(); it != vect.end(); ++it) {          |
|    //strcmp() method is used to compare the words or strings      |
|    if(strcmp(*it == "Content-Length:") == 0) {                     |
|       //If there exists a Content-Length, then set length to be the|
|       //int converted string right after "Content-Length:"        |
|       length = atoi(++it);                                        |
|    }                                                               |
| }                                                                  |
```

- Now we simply read the data and write to the file, but make sure to return a 200 OK or 201 OK depending if the file was created or not
- Also consider the content length

```
| if(length == -1) {                                                 |
|   do {                                                             |
|      //read from fd again to get data after getting the header     |
|      nbytes = read(fd, buff, buffsize);                            |
|      write(writeFD, buffer, nbytes);                               |
|      memset(buff, 0, buffsize);                                    |
|      //Checks for the created flag to determine a 200 or 201 reply |
|      if(flag == 0) { write(fd, 200OK, strlen(200OK); }             |
|      else if(flag == 1) { write(fd, 201OK, strlen(201OK); }        |
|   } while (nbytes > 0);                                            |
| }                                                                  |
| else {                                                             |
|   do {                                                             |
|      nbytes = read(fd, buff, length);                              |
|      write(writeFD, buff, nbytes);                                 |
|      memset(buff, 0 , buffsize);                                   |
|      length = length - nbytes;                                     |
|      if(length == 0) {                                             |
|         if(flag == 0) { write( fd, 200OK, strlen(200OK); }         |
|         else if(flag == 1) { write(fd, 201OK, strlen(201OK; }      |
|      }                                                             |
|   } while(length > 0);                                             |
| }                                                                  |
| //Dont forget to close the fd                                      |
| close(writeFD);                                                    |
```

-Now for the method of responding to the GET command:
 - GET would require the server to have the file, open the file, read the file, and then write to the client
 - As such, Error 404 must be accounted for, along with Error 403 if there is no permission to read the file, and 200 OK reply must be included as a success reply.
 - Get is similar to PUT, so it will copy the first few methods for verifying the name of the file and opening it.
 - For Error 403, it will use the same method PUT had used: comparing errno
 - For Error 404, we will have to compare errno to 2

```
| if(errno == 2) {                                                    |
|    write(fd, 404Error, strlen(404Error);                            |
| }                                                                   |
```

 - Additionally we have to account for the Content-Length of the file so that we may print it out to the client
 - In order to do this, we will have to use lseek() on the fd returned by opening the to-be-read file to get the integer, use to_string() to turn it into a string and add necesseary elements to the string ("Content-Length: "), write the string to the socket, then use lseek again reset the offset to the beginning of the file (as it was set to the end of the file from the first call).

```
| int size = lseek(readFD, 0, SEEK_END);                              |
| string cString = "Content-Length: " + size + "\r\n";                |
| write(fd, cString.c_str(), strlen(cString.c_str());                 |
| lseek(readFD, 0, SEEK_SET);                                         |
```

 - Now we can begin reading and writing from the file to client

```
| for(;;) {                                                           |
|   int nbytes = read(readFD, buffer, sizeof(buffer));                |
|   if(nbytes == 0) {                                                 |
|     write(fd, 200OK, strlen(200OK);                                 |
|     close(readFD);                                                  |
|     return;                                                         |
|   }                                                                 |
|   else {                                                            |
|     write(fd, buffer, nbytes);                                      |
|     memset(buffer, 0, sizeof(buffer));                              |
|   }                                                                 |
| }                                                                   |
```

-And with that, the program should be ready to handle GET and PUT
 commands from the client