Purpose:

This basic program is designed to serve as a basic HTTP or HTTPS client capable of making GET or HEAD requests to a functional web server.

Client:

The program begins by validating the input from the user by checking that they have passed in valid arguments in the correct places.

After this, the program examines the first argument to the program to determine whether the user is using HTTP or HTTPS. If no protocol is specified, we assume HTTP.

Next, parseArgs() extracts the target host name or IP, port, and resource to perform the request on. If no port is specified, we assume the default port of 80 for HTTP and 443 for HTTPS. If no target resource is specified, we assume "/", which will likely translate to index.html, but is dependent on the web server's implementation.

Next, we check whether the hostname specified is a hostname or an IP. If it is a hostname, we perform a DNS lookup using getaddrinfo() inside of the dnsLookup() function. If the dnsLookup() call is successful, it will return a valid socket with a connection to the specified server. If the specified hostname is an IP, we simply create a socket to connect to the ip and port specified.

Next, if the request is HTTP or HTTPS, we execute processHTTPRequest() or processHTTPSRequest() respectively. In both cases, the functions begin by building an HTTP request appropriate to the arguments of the program. Next, if we are performing an HTTPS request, we create an SSL connection with the server. Then, in either case, we read an initial response from the server.

If we are making a HEAD request, we will parse the response to see if it is the full length of the response. In either case, we will write the response to STDOUT. If the response is not the full response (we see no /r/n/r/n), then we will continually read from the socket and write to STDOUT until we see this response.

If we are making a GET request, we will parse the response for the Content-Length field. We extract this value with getContentLength(). Next, we create output.dat and then read from the socket and write to the file so long as there is content left to read. We decrement contentLength as we go in order to keep track of this.

Finally, when we finish in the HTTP case, we simply clean up all our memory and then quit. If we finish an HTTPS request, we execute an SSL graceful shutdown and then exit.

In any case in which something goes wrong (failing to read from a socket, failing to perform a DNS lookup, failing to make an SSL connection, etc.), the client will generate an informative message explaining the error. Then, the client will clean up all memory and close all connections and exit with EXIT_FAILURE.