12/10/2010

WEB SERVER

PROJECT REPORT

CIS 554 OBJECT ORIENTED PROGRAMMING IN C++

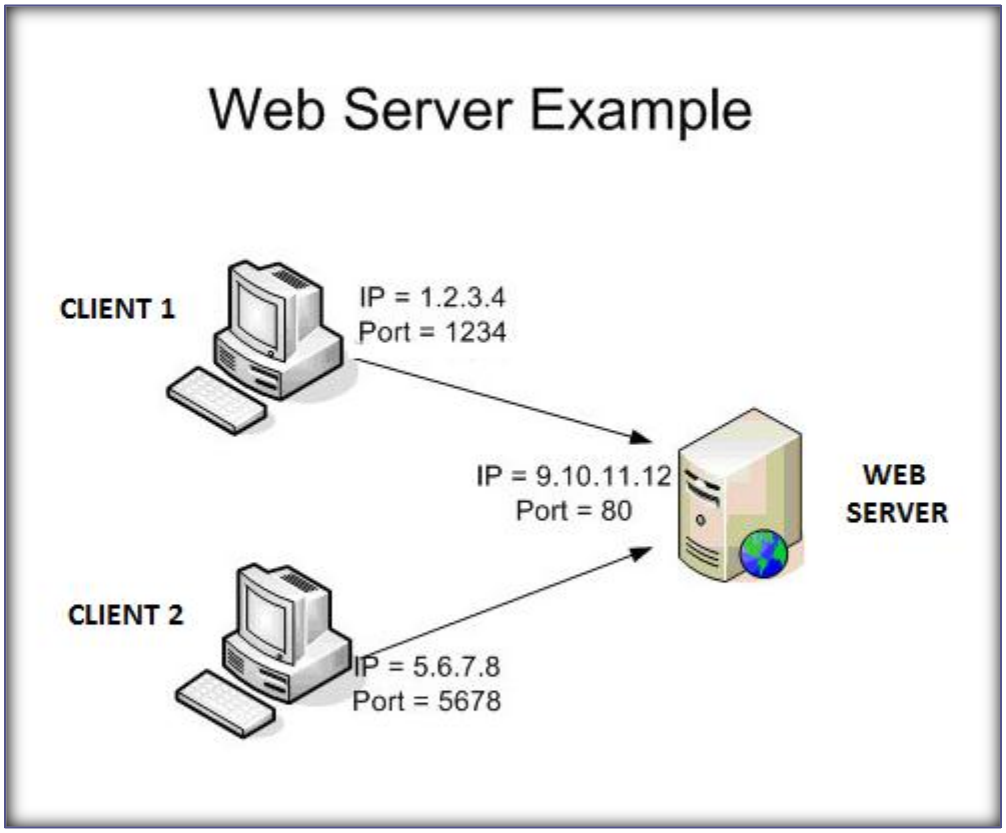
HARSHAL BHAKTA
SHAZIA BEE

Content

1.0 PROBLEM STATEMENT

Web Server Definition

- Computer program that runs on the server to deliver content to client machine.
- Content is basically served in the form of HTML documents and additional content that may be included as a document, such as images, style sheets and JavaScripts.
- Content is transmitted using the HTTP protocol.
- Additionally, an advanced web-server also handles receiving content from clients, processing the information and producing results back to the client.
- A client is typically a Web Browser that connects to the server to fetch the content.



The web-sever is basically an HTTP server that accepts HTTP connections from client browsers and passes on the content from the server to the client. Several client machines spread across the network can simultaneously establish connection and request content from the server. A basic web-server will deliver static web pages. Generally more advanced servers also handle dynamic pages which involves processing to be performed at the server side. The content is primararily deliverd in HTML format.

2.0 PROJECT SCOPE

Basic activities related to Web Server are : (GET Page implemented as mentioned below)

- Web-server Installer & Configuration manager.
- Starting the Web Server.
- Stopping the Web Server.
- Accept & handle HTTP Connections from client browsers.
- Implement HTTP Protocal for communication.
- Configure Port 80 on the server to provide web services.
- Client browser will establish connection to Server using IP Address + Port 80.
- Browser sends a GET request to the server requesting a file. (Ex. testPage.html)
- The server transmits the HTML text for the file to the browser.
- The browser reads the HTML tags received from the server and display the page.
- Configure default HTML page.
- Generate Real Time Server Logs for errors and warnings..
- Maintain statistics of the current connections.
- Limit the number of simultaneous connections.

Core essential network steps are :

- Create a listening socket
- Accept a connection with it
- Fork a child thread to service the connection, whilst the parent thread goes back to accept more connections.
- Read and accpept the HTTP request
- Send the HTTP response
- Send the entity requested (e.g. an HTML document)

3.0 HIGH LEVEL OVERVIEW OF BASIC FUNCTIONALITY

Below mentioned are a the high level steps to be handled for functional implementaion.

STEP 1 : WAIT FOR A NEW REQUEST

- Web server (http program) waits for a request to arrive from client over network.
- Program listens to a port until some client browser calls it.
- Server program will be dormant till a request is received.

STEP 2 : A REQUEST ARRIVES FROM A CLIENT

- Action begins when client browser program requests a document from server.
- Basically, a URL will trigger the action from the client to the server.
- Client network software connects to server using Internet Protocol.
- Client makes a request to server based on HTTP protocol over the network.

Example URL : http://www.myserver.org/test.html

Example Request Format : METHOD DOCUMENT PROTOCOL

Example Request : GET /test.html HTTP/1.0 (ASCII Format)

STEP 3 : THE SERVER PARSES THE REQUEST

- Server decodes the request based on HTTP protocol to determine the action required.
- The method in the request specifiec the precise action.
- Example : GET indicates that server should locate and read a file and return it to client.
- Information is sent back over the same connection that the request came from.

STEP 4 : DO THE METHOD REQUESTED

- Server fullfills the request by searching its file system for the requested file.
- Server send back a result code indicationg status of request and information to be sent.
- Example : Result code = 200 -> OK (Content-Type : text/html)
- Example : Result code = 403 -> NOT FOUND.
- Server sends information like document length, server information & content length.
- Server reads file from the disk and writes to the network port.

STEP 5 : FINISH UP – CLOSE FILE & CLOSE NETWORK CONNECTION

- Once file or error message is sent, file is closed.
- Server closes network ports, which terminates the network connection.
- Now it is upto the client to receive the data and format it for user.

STEP 6 : GO TO STEP 1

- Server is now ready to handle another request.
- On receiveing a new request from the network server repeats the mentioned steps.

4.0 IDENTIFIED REQUIRED COMPONENTS

- Web Server : Controls the execution of the web server.

- Socket : Socket from the header filr `WinSock2.h`

- Server Socket : Socket to act as server to listen for client request.

- File Logger : Logger to generate web server logs in a File.

- Console Logger : Logger to generate web server logs on the console.

- Error Logger : Logger to generate web server error logs in a File.

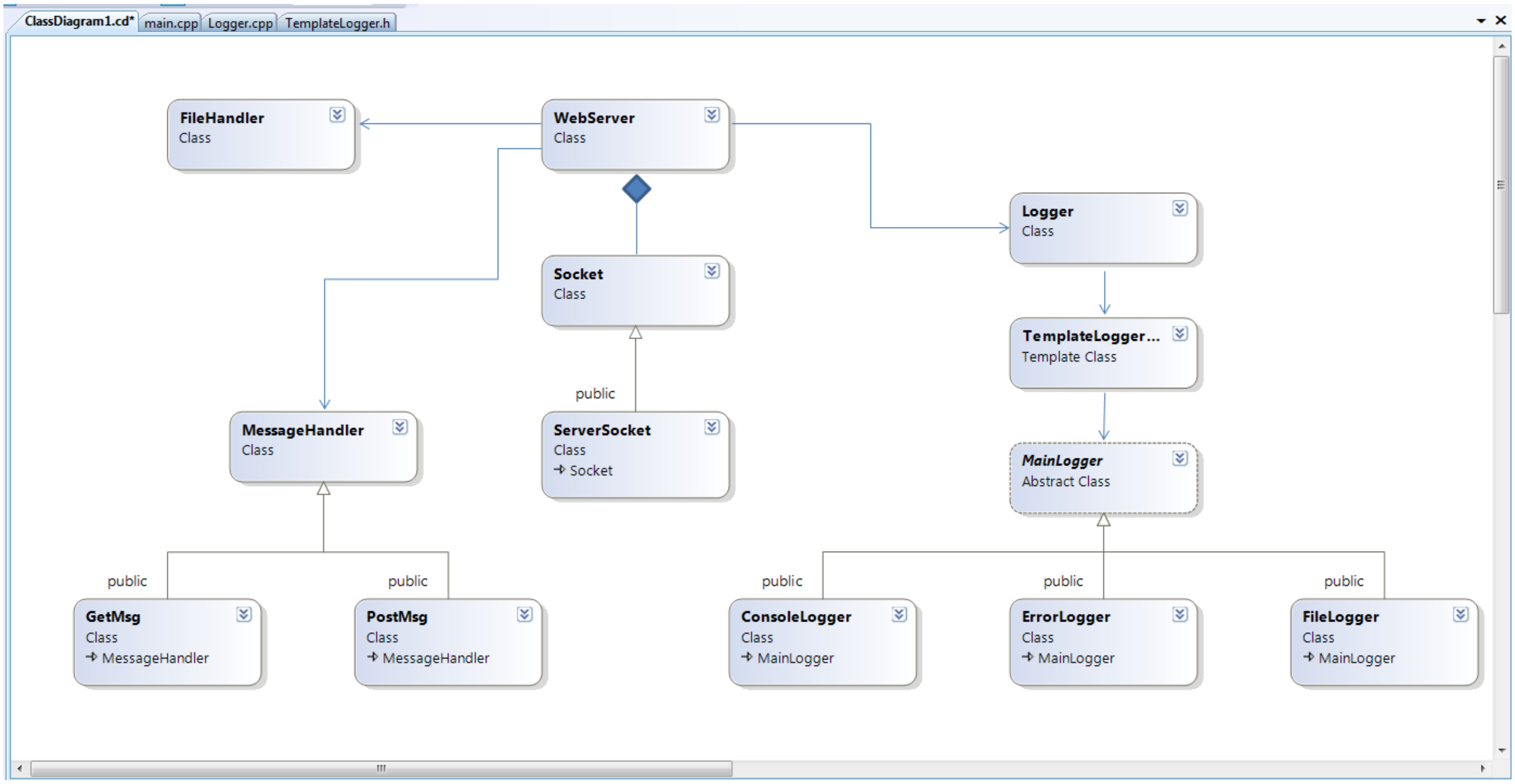
- Get Message Handler : Handler to parse and extract data from HTTP GET Message.

- Post Message Handler : Handler to generate data for HTTP POST Message.

- File Handler : Interact with the file directory to fetch required HTML files.

5.0 DESIGN OF THE PROGRAM (CHANGED DURING IMPLEMENTATION)

We describe various classes, methods and attributes required for the implementation. The proposed classes have been mentioned in the Proposal Document. However, there have been some changes made to match the behavior of the program and to work with the SOCKET provided by "WinSock2.h".



Class : WebServer

Webserver class will be responsible for controlling the execution of the progeram. The Listener of the server program starts from the Web Serverclass. The server will accept the GET request and will generate the response containing the HTML content of the file requested.

```
class WebServer{

int port ;

struct httpRequest
{
   Socket* HttpSocket;
};

public  :
      WebServer(int);    // Port as input for the constructor.
      void static setLogger();
      void StartWebServer();  // Function to Start the Web server.
private   :
      static unsigned __stdcall GenerateResponse(void*);
};
```

Class : Socket

Socket class will be responsbile for creating the Sockets from "WinSock2.h" provided by Microsoft for creating the Sockets. The various operations being performed on the socket will be included in this class. We derive the Server Socket class. We create base class Socket as we will in future be deriving the Client class for some HTTP server client applications.

```
class Socket
{
public :
      virtual ~Socket();
      Socket(const Socket&);
      Socket& operator=(Socket&);
      void SendLine(string);
      string GetRequestLine();
      void CloseSocket();
protected:
      friend class ServerSocket;
      Socket(SOCKET s);
      Socket();
      SOCKET DefaultSocket; // Default Socket for class Socket.
      int* refCounter_;
private :
      static int socketCount;       // Keep count of sockets.
      static void Begin();
      static void Finish();
};
```

Class : ServerSocket

ServerSocket class is derived from the Socket class. The socket class is a general class which we created to also act as a ServerClient to initially test the server-client functionality. ServerSocket's main task is going to be to create a new socket and listen for new client to connect.

```
class ServerSocket : public Socket
{
    public:
            // Wait for clients to connect.
            Socket* ListenForClient();
            // Port - Number of connections
            ServerSocket(int,int);
};
```

Class : MessageHandler

Message Handler is the base class for Get and Post Messages. There are a few attributes that we will require for both the Get and Post Messages. We derive the Get and Post Messages from a common base class as both the messages will be following HTTP Protocol.

```
class MessageHandler
{

public :
	string gethttpProtocol();
	string getserver();
	string getfilePath();
	MessageHandler();
protected :
	string httpProtocol;
	string server;
	string filePath;

private :

};
```

Class : GetMsg

GetMsg will be resposible to fetch the required information from the GET request and store it in relavent variables. We use these values further in the PostMsg to generate the appropriate Post messages for the requested file.

```
class GetMsg : public MessageHandler
{
public :
	void extractGetMessageInfo(vector<string>);
	GetMsg();
	string getAcceptType();
	string getAcceptLanguage();
	string getUserAgent();
	string getAcceptEncoding();
	string getconnection();
	string getAcceptTypeTemplate();
	string getAcceptLanguageTemplate();
	string getUserAgentTemplate();
	string gethostTemplate();
	string getconnectionTemplate();
private :
	bool msgValidity;
	string AcceptType;
	string AcceptLanguage;
	string UserAgent;
	string AcceptEncoding;
	string connection;
	string AcceptTypeTemplate;
	string AcceptLanguageTemplate;
	string UserAgentTemplate;
	string hostTemplate;
	string connectionTemplate;
};
```

Class : PostMsg

PostMsg class will be responsible for generating the post message by taking the GetMsg as a reference. The various attributes that form the part of the HTTP Response are generated in the PostMsg class and finally used from here to be sent to the client who requested the page.

```
class PostMsg : public MessageHandler
{
public :
      PostMsg();
      PostMsg(string);
      void generatePostMsg(GetMsg);
      string gethttpResponseHeader();
      string gethttpdateTime();
      string getconnection();
      string getcontentType();
      string getcontentLength();
      string getemptyLine();
      string getfileContent();

private :
      bool msgValidity;
      string httpResponseHeader;
      string dateTime;
      string connection;
      string contentType;
      int contentLength;
      string emptyLine;
      string fileContent;
};
```

Class : MainLogger

Main logger will act as a base class for three loggres we will have for our web server. We will have Console Logger, Error Logger and File Logger dervied from this base class and we will also implement polymorphism for these classes to choose the appropriate logger at run time.

```
class MainLogger
{

public :

      virtual void Print() = 0;

      void setLogMessage(string);

protected :

      string LogMessageMain;

};
```

Class : FileLogger

FileLogger will be responsible for implementing a logger which wrties the logs generated by our webserver to a file LogFile.txt in the sever directory.

```
class FileLogger : public MainLogger
{

private :

      string logFileLocation;

public :

      virtual void Print();

};
```

Class : ConsoleLogger

ConsoleLogger will be responsible for implementing a logger which wrties the logs generated by our webserver to the console.

```
class ConsoleLogger : public MainLogger
{

private :

      string Message;

public :

      virtual void Print();

};
```

Class : ErrorLogger

ErrorLogger will be responsible for implementing a logger which wrties the error logs generated by our webserver to the file ErrorLog.txt in the server directory.

```
class ErrorLogger : public MainLogger
{

public :

      static void LogError(string);

};
```

Class : FileHandler

FileHandler class will be responsible for the IO operations required for fetching the requested file from server directory. The class will also perform certain important function like checking of the file exists or not , and if not then fetch the defauls file .

```
class FileHandler
{

public :
        bool ifFileExists();
        void getHtmlFileContent();
        FileHandler(string);     // input as a filename.
        void setFileName(string);
        int getFileLength();
        string getFileContent();

private :
        string fileName;
        string fileContent;

};
```

Class : TemplateLogger

TemplateLogger class was created for letting the Web Server administrator choose the typr of logger at runtime. Based on the choice , LogTemplate will be either – ConsoleLogger or FileLogger or Both. We have the Virtual function in both the classes to further implement polymorphism.

```
template <class LogTemplate>
class TemplateLogger
{

public :

        void logMessage(LogTemplate);

        void setMessage(string);

private :

        string Message;

};
```

Class : Logger

Logger class will be providing us with the static method for writing the logs using a single line. This single line function call to write the log will at runtime implement TemplateLogger and choose the appropirat logger(s) and write the logs.

```
class Logger
{

public :

      Logger();

      void static LogMessage(string);

      void static setMode(int);

      int static getMode();

      // mode = 1 : Console Logger
      // mode = 2 : File Logger
      // mode = 3 : Both
       static int errorMode;

private :


};
```

6.0 IMPLEMENTATION OF THE REQUIRED COMPONENTS OF THE PROJECT

Required Components of Project

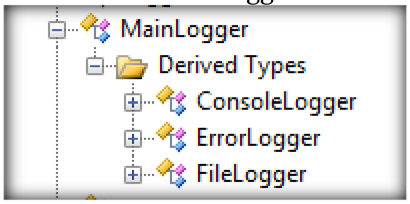
1. Multiple Classes (at least 10). You may re-use classes that you wrote during this course, but classes I provide or you get from another source DO NOT COUNT.

   Below is the list of the classes create for the project.

   1. ConsoleLogger
   2. ErrorLogger
   3. FileHandler
   4. FileLogger
   5. GetMsg
   6. Logger
   7. MainLogger
   8. MessageHandler
   9. PostMsg
   10. ServerSocket
   11. Socket
   12. TemplateLogger<LogTemplate>
   13. WebServer

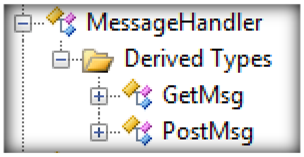
2. Inheritance

   Base Class : Main Logger



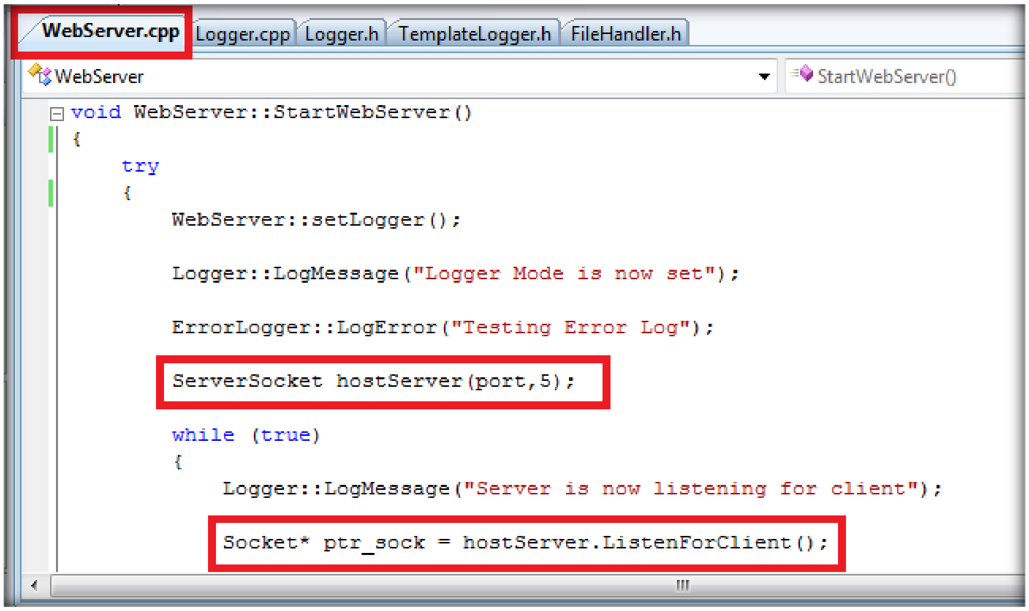
MainLogger
Derived Types
ConsoleLogger
ErrorLogger
FileLogger

   Base Class : Message Handler



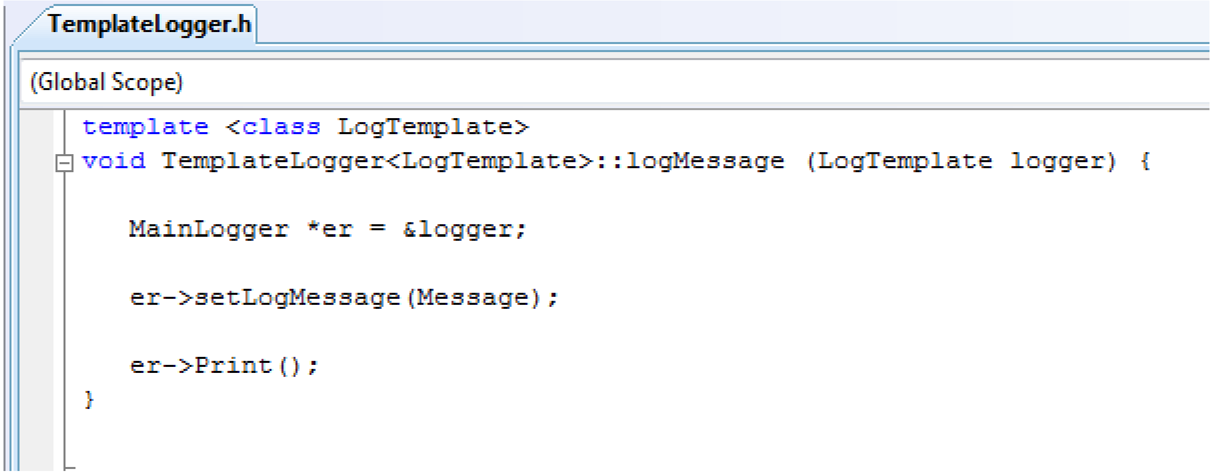
MessageHandler
Derived Types
GetMsg
PostMsg

3. Composition : (Class WebServer -> ServerSocket & Socket)



```
void WebServer::StartWebServer()
{
    try
    {
        WebServer::setLogger();

        Logger::LogMessage("Logger Mode is now set");

        ErrorLogger::LogError("Testing Error Log");

        ServerSocket hostServer(port,5);

        while (true)
        {
            Logger::LogMessage("Server is now listening for client");

            Socket* ptr_sock = hostServer.ListenForClient();
```

4. Polymorphism



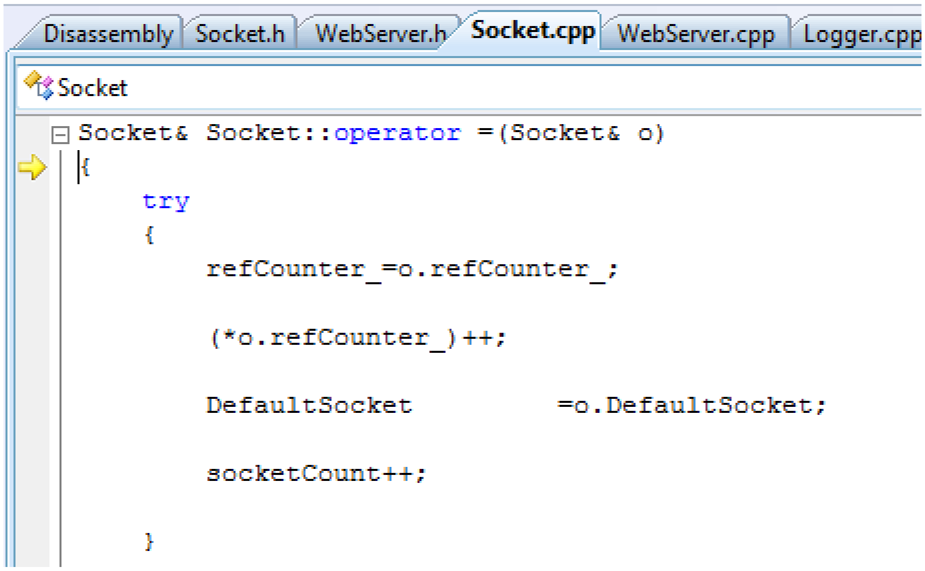
```
template <class LogTemplate>
void TemplateLogger<LogTemplate>::logMessage (LogTemplate logger) {

   MainLogger *er = &logger;

   er->setLogMessage(Message);

   er->Print();
}
```

***er** is the base class pointer and will appropriately point to one of the derived class object (Console Logger or File Logger) and call the **virtual function Print()**

```
class MainLogger
{

   public :

   virtual void Print() = 0;

}

class FileLogger : public MainLogger
{
   public :

   virtual void Print();
}

class ConsoleLogger : public MainLogger
{
   public :

   virtual void Print();
}
```

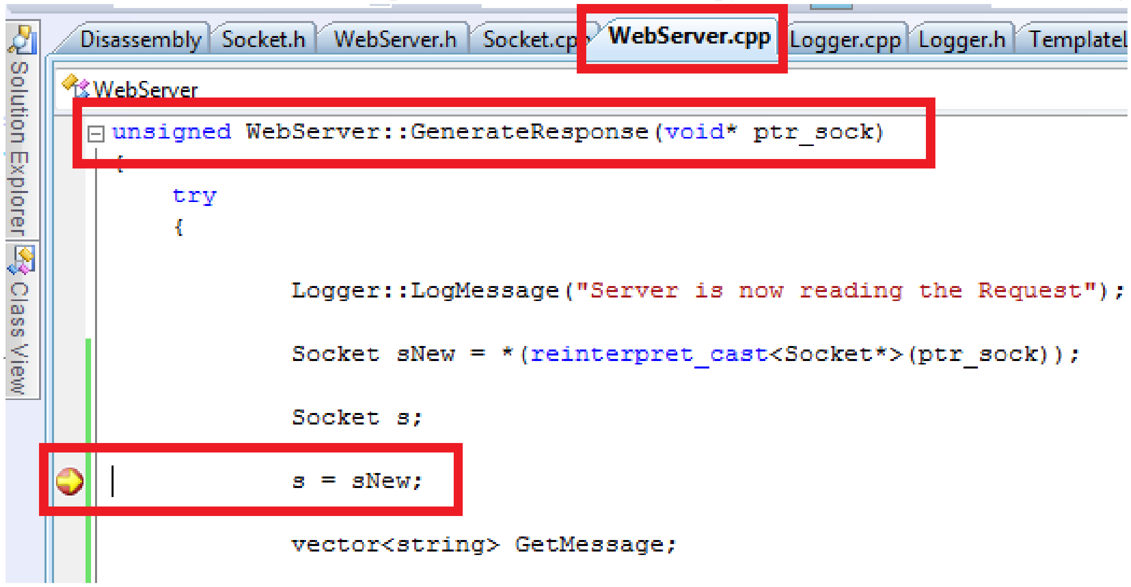
5. Operator overloading.

Overloading Operator “=” for our Socket class.



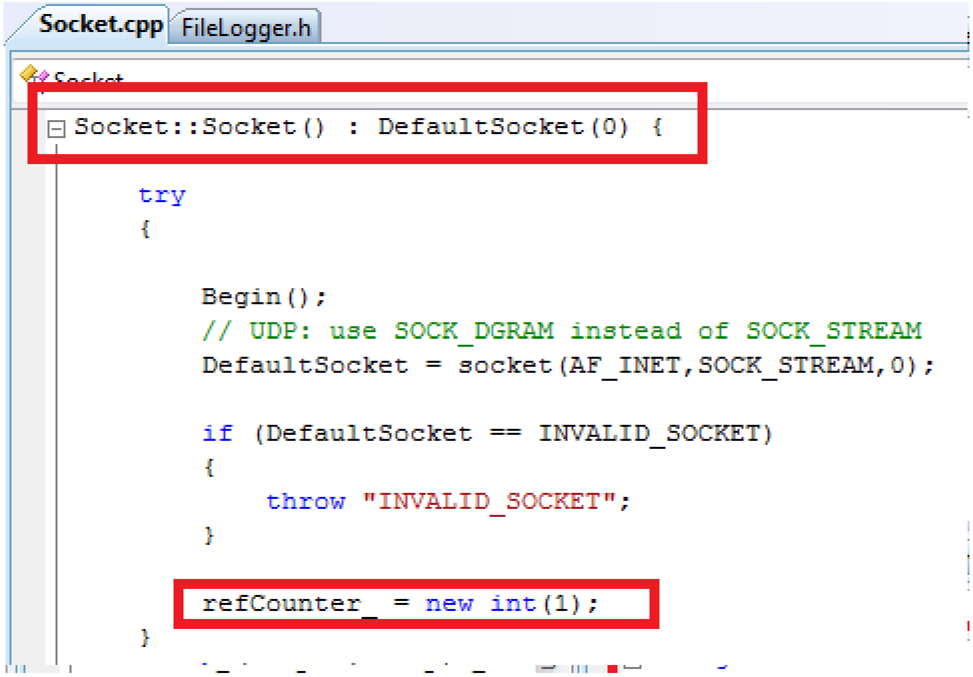
```
Socket& Socket::operator =(Socket& o)
{
    try
    {
        refCounter_=o.refCounter_;

        (*o.refCounter_)++;

        DefaultSocket           =o.DefaultSocket;

        socketCount++;

    }
```

Used in WebServer : GenerateRepsonse(void*) to create a copy and send to function GetMessage.

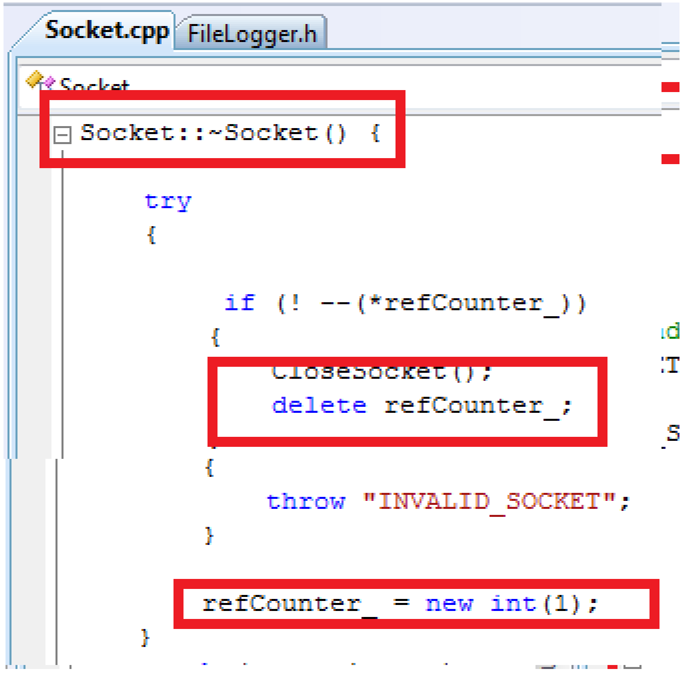


```
unsigned WebServer::GenerateResponse(void* ptr_sock)
{
    try
    {

            Logger::LogMessage("Server is now reading the Request");

            Socket sNew = *(reinterpret_cast<Socket*>(ptr_sock));

            Socket s;

            s = sNew;

            vector<string> GetMessage;
```

6. Dynamic memory allocation and deallocation.

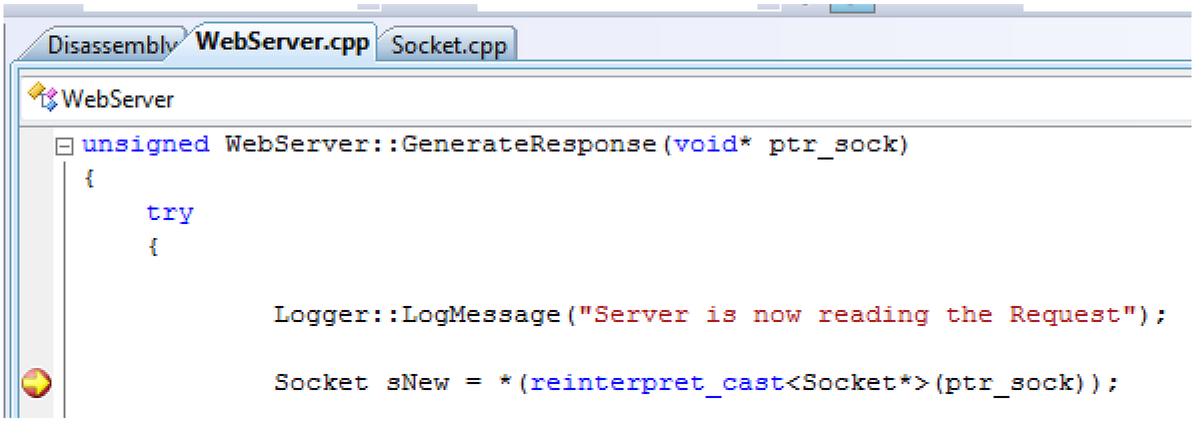


```
Socket::Socket() : DefaultSocket(0) {
    try
    {

        Begin();
        // UDP: use SOCK_DGRAM instead of SOCK_STREAM
        DefaultSocket = socket(AF_INET,SOCK_STREAM,0);

        if (DefaultSocket == INVALID_SOCKET)
        {
            throw "INVALID_SOCKET";
        }

        refCounter_ = new int(1);
    }
```

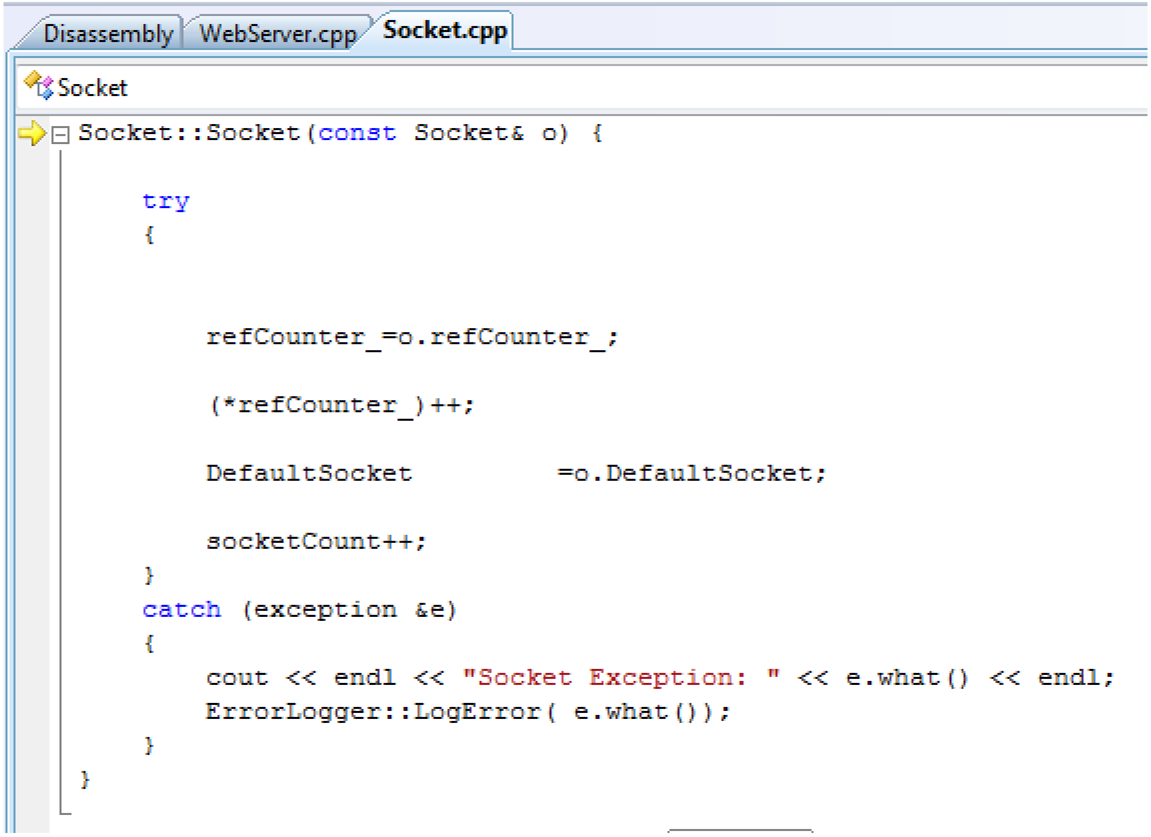


```
Socket::~Socket() {
    try
    {

        if (! --(*refCounter_))
        {
            CloseSocket();
            delete refCounter_;
        {
            throw "INVALID_SOCKET";
        }

        refCounter_ = new int(1);
    }
```

7. Proper copy constructors and assignment operators (where applicable).

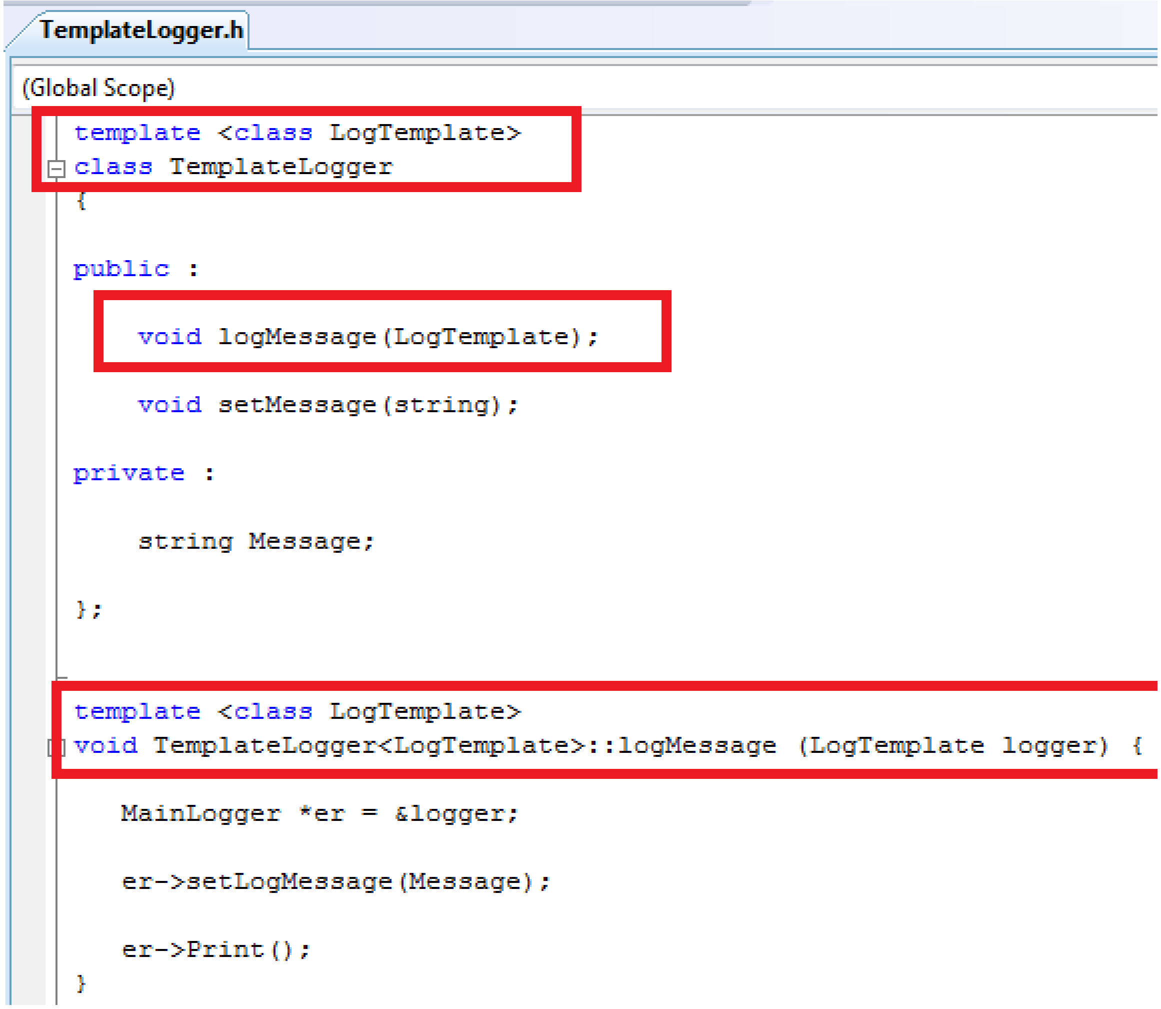


```
unsigned WebServer::GenerateResponse(void* ptr_sock)
{
    try
    {

            Logger::LogMessage("Server is now reading the Request");

            Socket sNew = *(reinterpret_cast<Socket*>(ptr_sock));
```

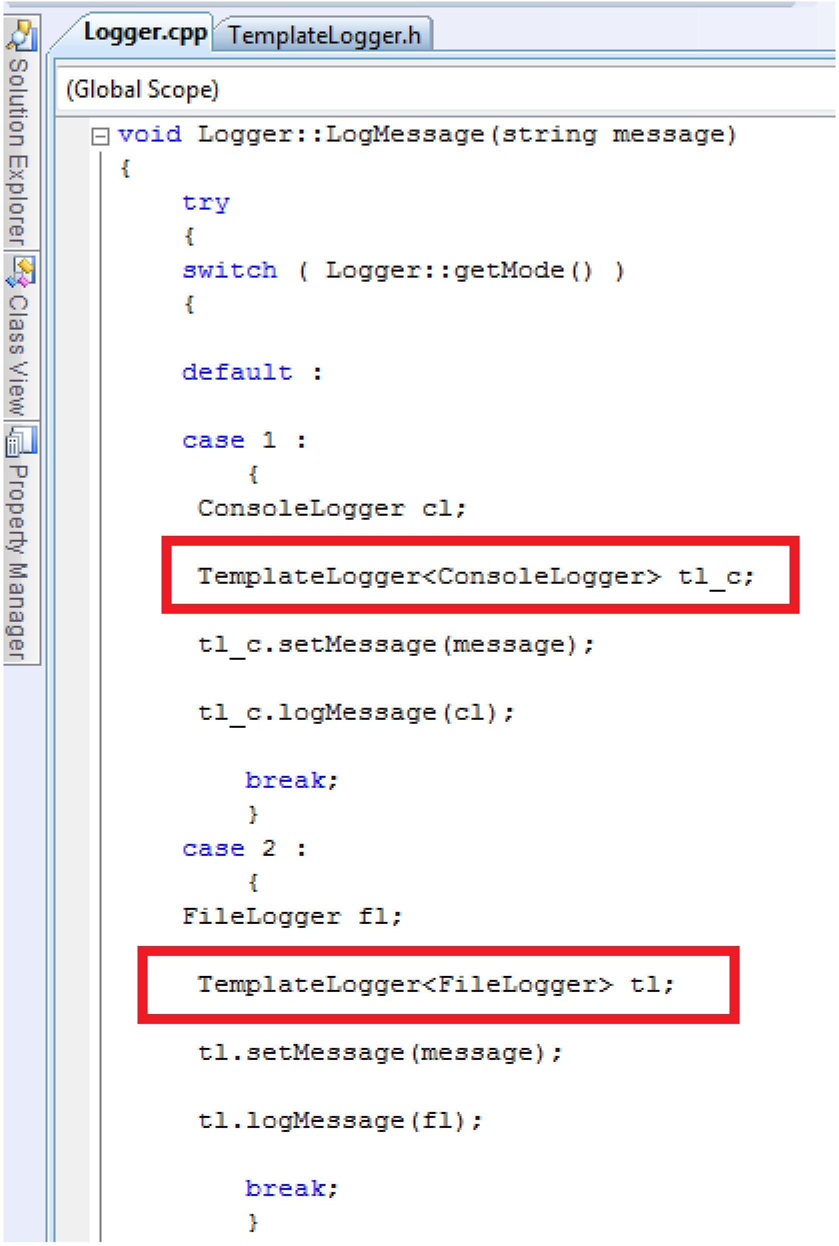


```
Socket::Socket(const Socket& o) {

    try
    {


        refCounter_=o.refCounter_;

        (*refCounter_)++;

        DefaultSocket         =o.DefaultSocket;

        socketCount++;
    }
    catch (exception &e)
    {
        cout << endl << "Socket Exception: " << e.what() << endl;
        ErrorLogger::LogError( e.what());
    }
}
```

8. Templates

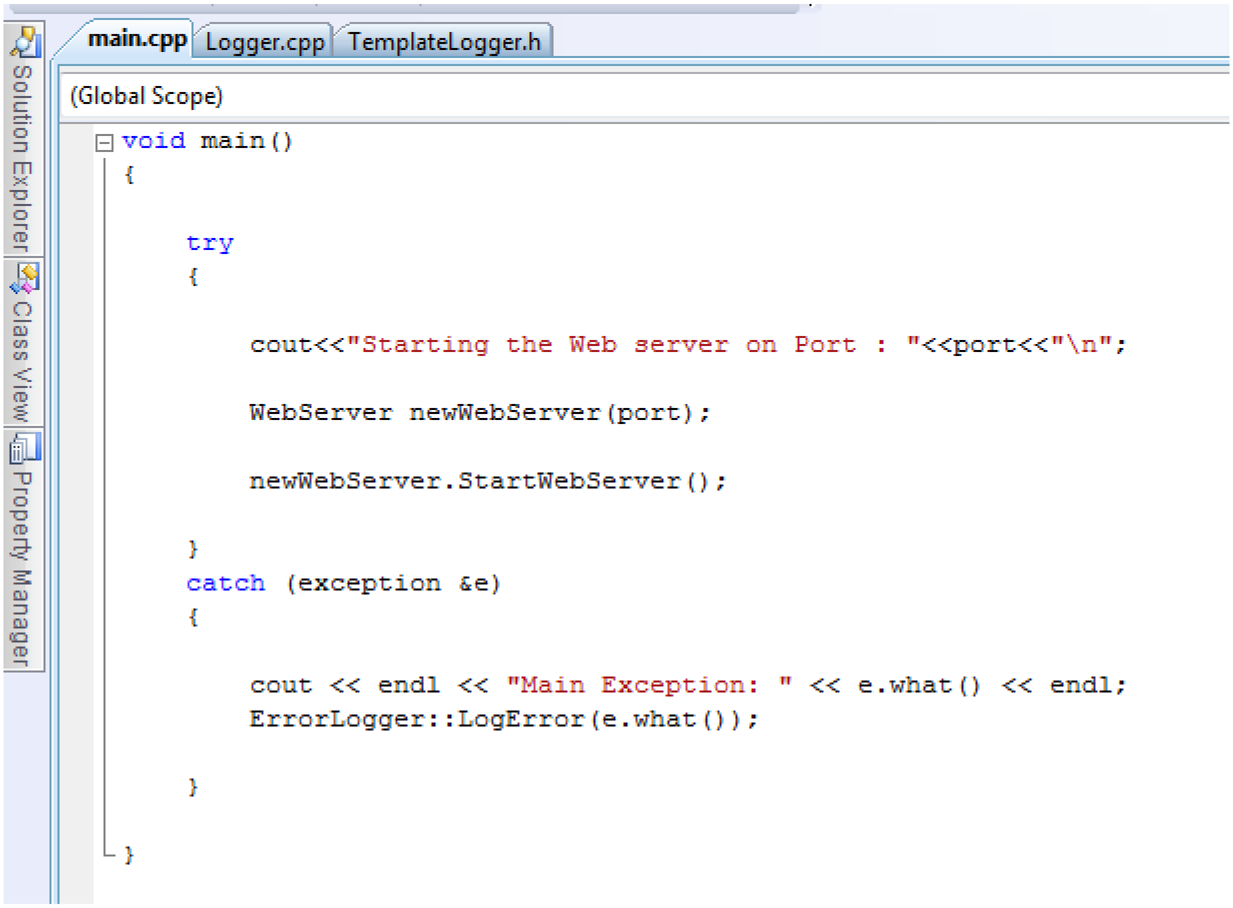


```
template <class LogTemplate>
class TemplateLogger
{

public :

    void logMessage(LogTemplate);

    void setMessage(string);

private :

    string Message;

};


template <class LogTemplate>
void TemplateLogger<LogTemplate>::logMessage (LogTemplate logger) {

   MainLogger *er = &logger;

   er->setLogMessage(Message);

   er->Print();
}
```



```
void Logger::LogMessage(string message)
{
    try
    {
    switch ( Logger::getMode() )
    {

    default :

    case 1 :
        {
     ConsoleLogger cl;

     TemplateLogger<ConsoleLogger> tl_c;

     tl_c.setMessage(message);

     tl_c.logMessage(cl);

        break;
        }
    case 2 :
        {
    FileLogger fl;

     TemplateLogger<FileLogger> tl;

     tl.setMessage(message);

     tl.logMessage(fl);

        break;
        }
```

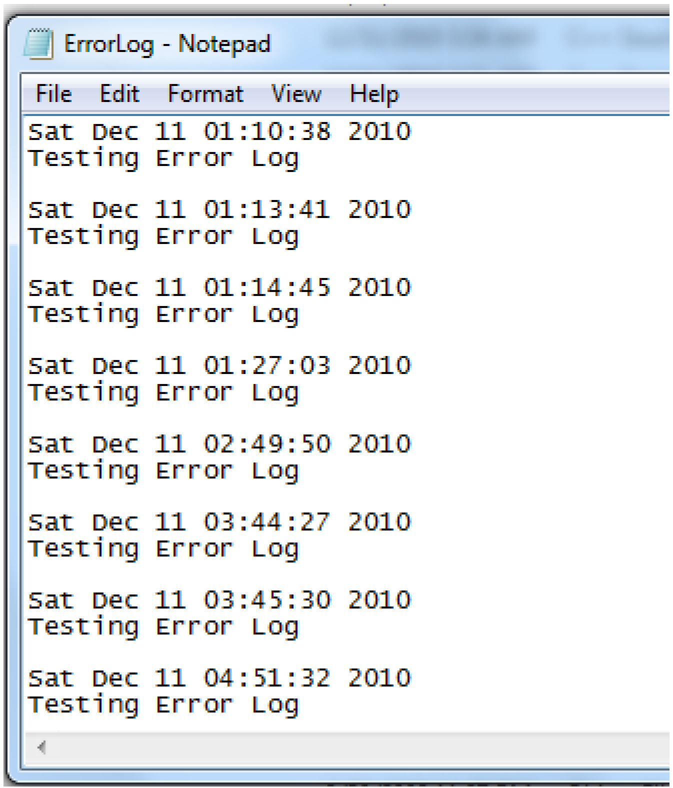
9. Exception Handling



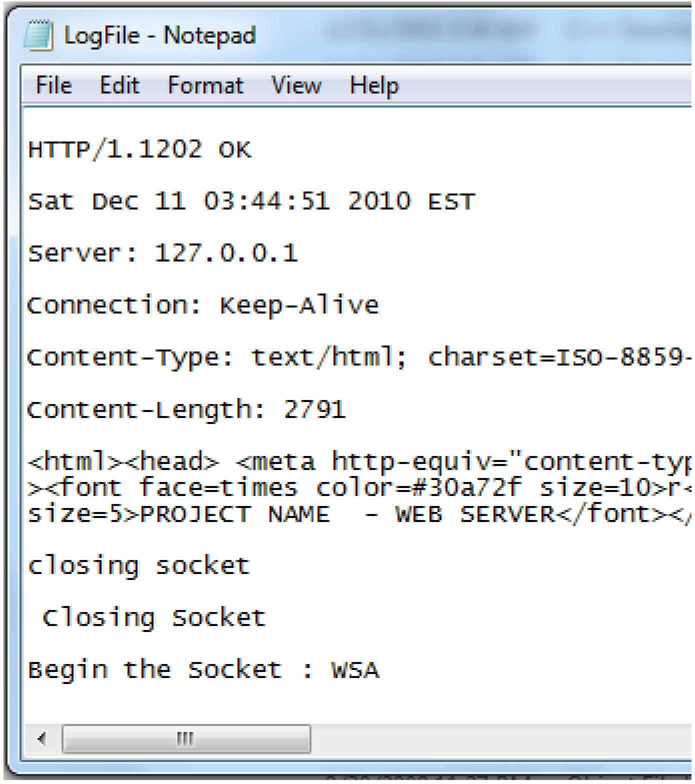
```
void main()
{

    try
    {

        cout<<"Starting the Web server on Port : "<<port<<"\n";

        WebServer newWebServer(port);

        newWebServer.StartWebServer();

    }
    catch (exception &e)
    {

        cout << endl << "Main Exception: " << e.what() << endl;
        ErrorLogger::LogError(e.what());

    }

}
```

7.0 EXTRA CREDIT COMPONENTS

- File Logging

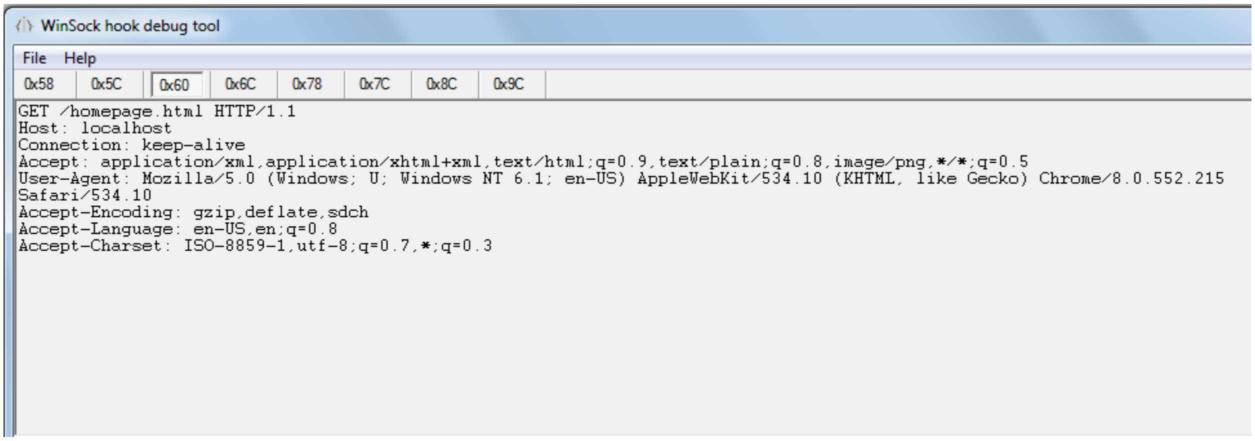


```
Sat Dec 11 01:10:38 2010
Testing Error Log

Sat Dec 11 01:13:41 2010
Testing Error Log

Sat Dec 11 01:14:45 2010
Testing Error Log

Sat Dec 11 01:27:03 2010
Testing Error Log

Sat Dec 11 02:49:50 2010
Testing Error Log

Sat Dec 11 03:44:27 2010
Testing Error Log

Sat Dec 11 03:45:30 2010
Testing Error Log

Sat Dec 11 04:51:32 2010
Testing Error Log
```



```

HTTP/1.1202 OK

Sat Dec 11 03:44:51 2010 EST

Server: 127.0.0.1

Connection: Keep-Alive

Content-Type: text/html; charset=ISO-8859-

Content-Length: 2791

<html><head> <meta http-equiv="content-typ
><font face=times color=#30a72f size=10>r
size=5>PROJECT NAME  - WEB SERVER</font></

closing socket

 Closing Socket

Begin the Socket : WSA
```

- WinSock Tool used to Monitor the GET Request & Response



```
GET /homepage.html HTTP/1.1
Host: localhost
Connection: keep-alive
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/534.10 (KHTML, like Gecko) Chrome/8.0.552.215
Safari/534.10
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

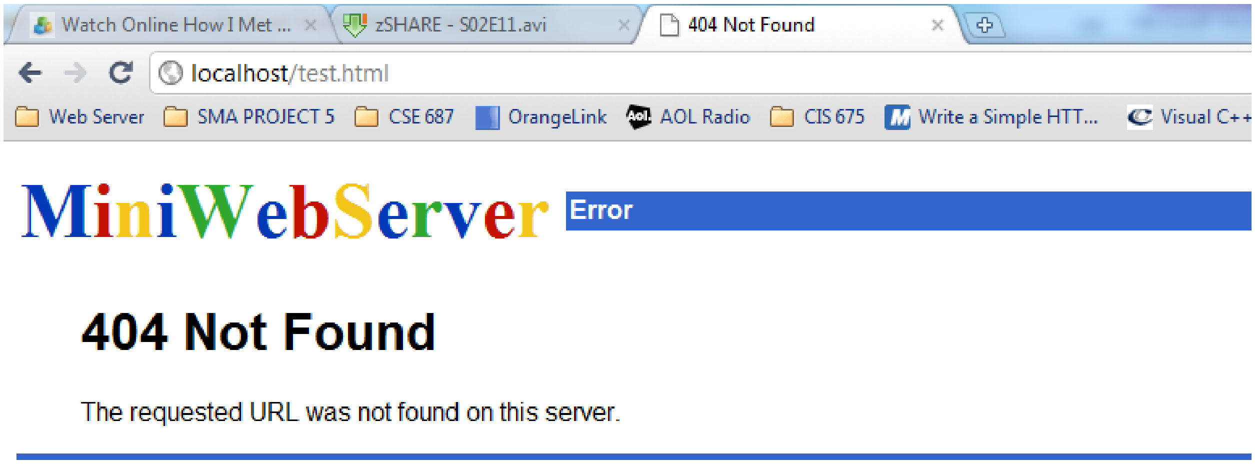
- Sample HTTP REQUEST : Extracted by using Winsock Tool

```
GET /test.html HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1;
Trident/5.0)
Accept-Encoding: gzip, deflate
Host: 127.0.0.1
Connection: Keep-Alive
```

- Sample HTTP RESPONSE : Extracted by using Winsock Tool

```
HTTP/1.1 202 OK
Date: Wed Dec 01 02:49:03 2010 GMT
Server: RenesWebserver (Windows)
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 394

<html><head><title>Web Server Example</title></head><body
bgcolor='#ffffff'><h1>Web Server Example</h1>I wonder what you're going
to click<p><a href='/red'>red</a> <br><a href='/blue'>blue</a> <br><a
href='/form'>form</a> <br><a href='/auth'>authentication example</a>
[use <b>adp</b> as username and <b>gmbh</b> as password<br><a
href='/header'>show some HTTP header details</a> </body></html>
```

- Implemented the 404 PAGE NOT FOUND ERROR.



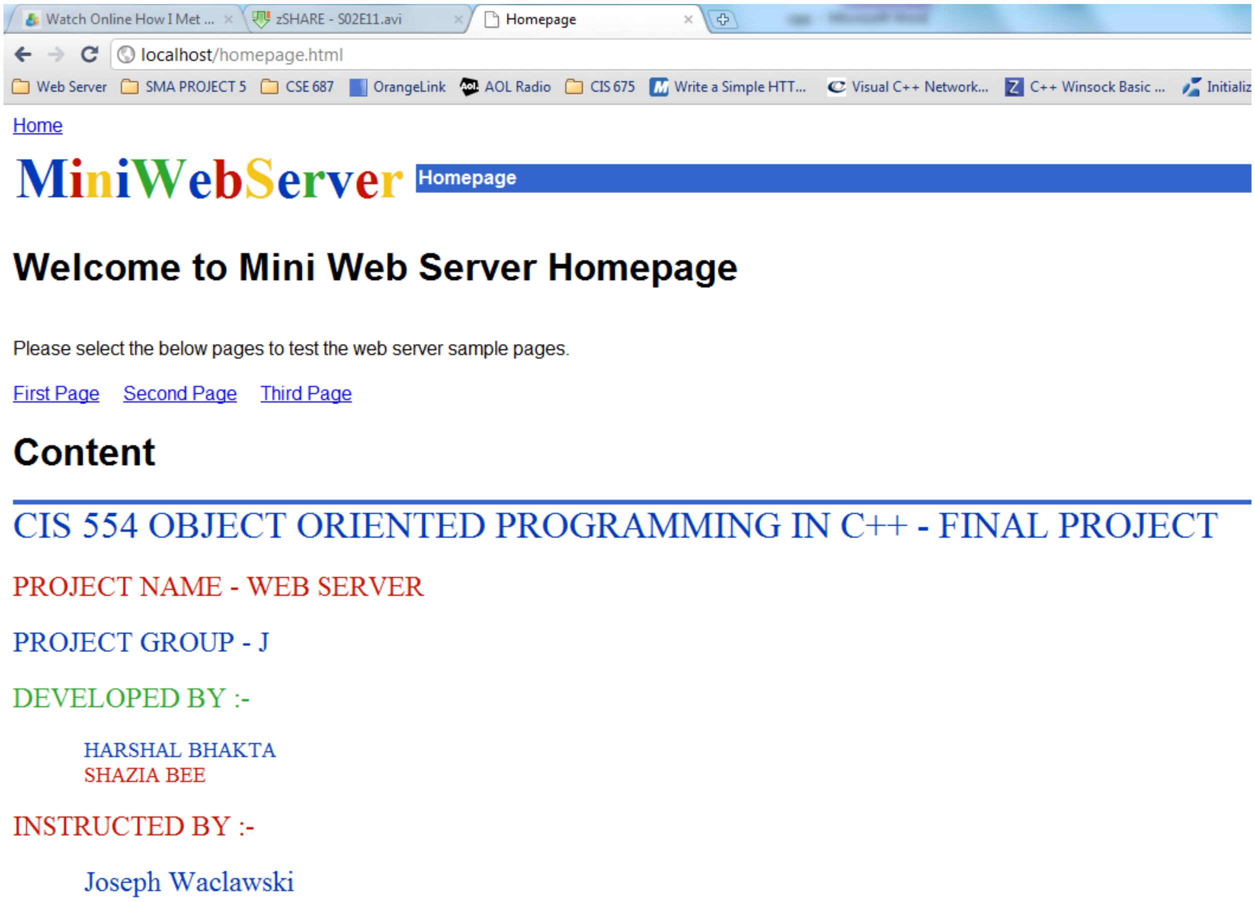
MiniWebServer Error

404 Not Found

The requested URL was not found on this server.

- Sample Pages Created to test Webserver.

  1. Homepage.html
  2. PageOne.html
  3. PageTwo.html
  4. PageThree.html



Home

MiniWebServer Homepage

Welcome to Mini Web Server Homepage

Please select the below pages to test the web server sample pages.

First Page Second Page Third Page

Content

CIS 554 OBJECT ORIENTED PROGRAMMING IN C++ - FINAL PROJECT

PROJECT NAME - WEB SERVER

PROJECT GROUP - J

DEVELOPED BY :-

HARSHAL BHAKTA
SHAZIA BEE

INSTRUCTED BY :-

Joseph Waclawski

8.0 REFRENCES

1) Web Server Technology : By Nancy J. Yeager & Robert E. McGrath
2) http://computer.howstuffworks.com/web-server2.htm
3) http://beej.us/guide/bgnet/
4) http://jmarshall.com/easy/http/
5) http://www.news.cs.nyu.edu/~jinyang/sp07/notes/webserver.pdf
6) http://www.news.cs.nyu.edu/~jinyang/sp07/notes/webclient.pdf
7) http://www.paulgriffiths.net/program/c/srcs/webservsrc.html
8) http://www.w3.org/Protocols/rfc2616/rfc2616.html
9) http://beej.us/guide/bgnet/output/html/multipage/index.html
10) http://www.adp-gmbh.ch/win/misc/webserver.html
11) http://www.ibm.com/developerworks/systems/library/es-nweb/index.html