```
#
# Makefile
#
# Computer Science 50
# Problem Set 6
#

server: server.c Makefile
    clang -ggdb3 -O0 -std=c11 -Wall -Werror -o server server.c -lm

clean:
    rm -f *.o core server

```

```
<!DOCTYPE html>

<html>
    <head>
        <title>Happy Cat</title>
    </head>
    <body>
        <img alt="Happy Cat" src="cat.jpg"/>
    </body>
</html>

```

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        <form action="hello.php" method="get">
            <input autocomplete="off" autofocus name="name" placeholder="Name" type="text"/>
            <input type="submit" value="Say Hello"/>
        </form>
    </body>
</html>

```

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
        <?php if (!empty($_GET["name"])): ?>
            hello, <?= htmlspecialchars($_GET["name"]) ?>
        <?php else: ?>
            hello, world
        <?php endif ?>
    </body>
</html>

```

```
<!DOCTYPE html>

<html>
    <head>
        <title>RickRoll'D</title>
    </head>
    <body>
        <iframe width="420" height="315" src="https://www.youtube.com/embed/oHg5SJYRHA0?autoplay=1&iv_load_policy=3" frameborder="0" allowfullscreen></iframe>
    </body>
</html>

```

```c
//
// server.c
//
// Computer Science 50
// Problem Set 6
//

// feature test macro requirements
#define _GNU_SOURCE
#define _XOPEN_SOURCE 700
#define _XOPEN_SOURCE_EXTENDED

// limits on an HTTP request's size, based on Apache's
// http://httpd.apache.org/docs/2.2/mod/core.html
#define LimitRequestFields 50
#define LimitRequestFieldSize 4094
#define LimitRequestLine 8190

// number of bytes for buffers
#define BYTES 512

// header files
#include <arpa/inet.h>
#include <dirent.h>
#include <errno.h>
#include <limits.h>
#include <math.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

// types
typedef char BYTE;

// prototypes
bool connected(void);
void error(unsigned short code);
void freedir(struct dirent** namelist, int n);
void handler(int signal);
char* htmlspecialchars(const char* s);
char* indexes(const char* path);
```

```
49. void interpret(const char* path, const char* query);
50. void list(const char* path);
51. bool load(FILE* file, BYTE** content, size_t* length);
52. const char* lookup(const char* path);
53. bool parse(const char* line, char* path, char* query);
54. const char* reason(unsigned short code);
55. void redirect(const char* uri);
56. bool request(char** message, size_t* length);
57. void respond(int code, const char* headers, const char* body, size_t length);
58. void start(short port, const char* path);
59. void stop(void);
60. void transfer(const char* path, const char* type);
61. char* urldecode(const char* s);
62.
63. // server's root
64. char* root = NULL;
65.
66. // file descriptor for sockets
67. int cfd = -1, sfd = -1;
68.
69. // flag indicating whether control-c has been heard
70. bool signaled = false;
71.
72. int main(int argc, char* argv[])
73. {
74.     // a global variable defined in errno.h that's "set by system
75.     // calls and some library functions [to a nonzero value]
76.     // in the event of an error to indicate what went wrong"
77.     errno = 0;
78.
79.     // default to port 8080
80.     int port = 8080;
81.
82.     // usage
83.     const char* usage = "Usage: server [-p port] /path/to/root";
84.
85.     // parse command-line arguments
86.     int opt;
87.     while ((opt = getopt(argc, argv, "hp:")) != -1)
88.     {
89.         switch (opt)
90.         {
91.             // -h
92.             case 'h':
93.                 printf("%s\n", usage);
94.                 return 0;
95.
96.             // -p port
```

```
 97.             case 'p':
 98.                 port = atoi(optarg);
 99.                 break;
100.         }
101.     }
102.
103.     // ensure port is a non-negative short and path to server's root is specified
104.     if (port < 0 || port > SHRT_MAX || argv[optind] == NULL || strlen(argv[optind]) == 0)
105.     {
106.         // announce usage
107.         printf("%s\n", usage);
108.
109.         // return 2 just like bash's builtins
110.         return 2;
111.     }
112.
113.     // start server
114.     start(port, argv[optind]);
115.
116.     // listen for SIGINT (aka control-c)
117.     struct sigaction act;
118.     act.sa_handler = handler;
119.     act.sa_flags = 0;
120.     sigemptyset(&act.sa_mask);
121.     sigaction(SIGINT, &act, NULL);
122.
123.     // a message and its length
124.     char* message = NULL;
125.     size_t length = 0;
126.
127.     // path requested
128.     char* path = NULL;
129.
130.     // accept connections one at a time
131.     while (true)
132.     {
133.         // free last path, if any
134.         if (path != NULL)
135.         {
136.             free(path);
137.             path = NULL;
138.         }
139.
140.         // free last message, if any
141.         if (message != NULL)
142.         {
143.             free(message);
144.             message = NULL;
```

```
145.         }
146.         length = 0;
147.
148.         // close last client's socket, if any
149.         if (cfd != -1)
150.         {
151.             close(cfd);
152.             cfd = -1;
153.         }
154.
155.         // check for control-c
156.         if (signaled)
157.         {
158.             stop();
159.         }
160.
161.         // check whether client has connected
162.         if (connected())
163.         {
164.             // check for request
165.             if (request(&message, &length))
166.             {
167.                 // extract message's request-line
168.                 // http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
169.                 const char* haystack = message;
170.                 const char* needle = strstr(haystack, "\r\n");
171.                 if (needle == NULL)
172.                 {
173.                     error(500);
174.                     continue;
175.                 }
176.                 char line[needle - haystack + 2 + 1];
177.                 strncpy(line, haystack, needle - haystack + 2);
178.                 line[needle - haystack + 2] = '\0';
179.
180.                 // log request-line
181.                 printf("%s", line);
182.
183.                 // parse request-line
184.                 char abs_path[LimitRequestLine + 1];
185.                 char query[LimitRequestLine + 1];
186.                 if (parse(line, abs_path, query))
187.                 {
188.                     // URL-decode absolute-path
189.                     char* p = urldecode(abs_path);
190.                     if (p == NULL)
191.                     {
192.                         error(500);
```

```
193.                     continue;
194.                 }
195.
196.                 // resolve absolute-path to local path
197.                 path = malloc(strlen(root) + strlen(p) + 1);
198.                 if (path == NULL)
199.                 {
200.                     error(500);
201.                     continue;
202.                 }
203.                 strcpy(path, root);
204.                 strcat(path, p);
205.                 free(p);
206.
207.                 // ensure path exists
208.                 if (access(path, F_OK) == -1)
209.                 {
210.                     printf("Acess -%s-",abs_path);
211.                     error(404);
212.                     continue;
213.                 }
214.
215.                 // if path to directory
216.                 struct stat sb;
217.                 if (stat(path, &sb) == 0 && S_ISDIR(sb.st_mode))
218.                 {
219.                     // redirect from absolute-path to absolute-path/
220.                     if (abs_path[strlen(abs_path) - 1] != '/')
221.                     {
222.                         char uri[strlen(abs_path) + 1 + 1];
223.                         strcpy(uri, abs_path);
224.                         strcat(uri, "/");
225.                         redirect(uri);
226.                         continue;
227.                     }
228.
229.                     // use path/index.php or path/index.html, if present, instead of directory's path
230.                     char* index = indexes(path);
231.                     if (index != NULL)
232.                     {
233.                         free(path);
234.                         path = index;
235.                     }
236.
237.                     // list contents of directory
238.                     else
239.                     {
240.                         list(path);
```

```
                            continue;
                        }
                    }

                    // look up MIME type for file at path
                    const char* type = lookup(path);
                    if (type == NULL)
                    {
                        error(501);
                        continue;
                    }

                    // interpret PHP script at path
                    if (strcasecmp("text/x-php", type) == 0)
                    {
                        interpret(path, query);
                    }

                    // transfer file at path
                    else
                    {
                        transfer(path, type);
                    }
                }
            }
        }
    }
}

/**
 * Checks (without blocking) whether a client has connected to server.
 * Returns true iff so.
 */
bool connected(void)
{
    struct sockaddr_in cli_addr;
    memset(&cli_addr, 0, sizeof(cli_addr));
    socklen_t cli_len = sizeof(cli_addr);
    cfd = accept(sfd, (struct sockaddr*) &cli_addr, &cli_len);
    if (cfd == -1)
    {
        return false;
    }
    return true;
}

/**
 * Responds to client with specified status code.
```

```
 */
void error(unsigned short code)
{
    // determine code's reason-phrase
    const char* phrase = reason(code);
    if (phrase == NULL)
    {
        return;
    }

    // template for response's content
    char* template = "<html><head><title>%i %s</title></head><body><h1>%i %s</h1></body></html>";

    // render template
    char body[(strlen(template) - 2 - ((int) log10(code) + 1) - 2 + strlen(phrase)) * 2 + 1];
    int length = sprintf(body, template, code, phrase, code, phrase);
    if (length < 0)
    {
        body[0] = '\0';
        length = 0;
    }

    // respond with error
    char* headers = "Content-Type: text/html\r\n";
    respond(code, headers, body, length);
}

/**
 * Frees memory allocated by scandir.
 */
void freedir(struct dirent** namelist, int n)
{
    if (namelist != NULL)
    {
        for (int i = 0; i < n; i++)
        {
            free(namelist[i]);
        }
        free(namelist);
    }
}

/**
 * Handles signals.
 */
void handler(int signal)
{
    // control-c
```

```
337.     if (signal == SIGINT)
338.     {
339.         signaled = true;
340.     }
341. }
342.
343. /**
344.  * Escapes string for HTML. Returns dynamically allocated memory for escaped
345.  * string that must be deallocated by caller.
346.  */
347. char* htmlspecialchars(const char* s)
348. {
349.     // ensure s is not NULL
350.     if (s == NULL)
351.     {
352.         return NULL;
353.     }
354.
355.     // allocate enough space for an unescaped copy of s
356.     char* t = malloc(strlen(s) + 1);
357.     if (t == NULL)
358.     {
359.         return NULL;
360.     }
361.     t[0] = '\0';
362.
363.     // iterate over characters in s, escaping as needed
364.     for (int i = 0, old = strlen(s), new = old; i < old; i++)
365.     {
366.         // escape &
367.         if (s[i] == '&')
368.         {
369.             const char* entity = "&amp;";
370.             new += strlen(entity);
371.             t = realloc(t, new);
372.             if (t == NULL)
373.             {
374.                 return NULL;
375.             }
376.             strcat(t, entity);
377.         }
378.
379.         // escape "
380.         else if (s[i] == '"')
381.         {
382.             const char* entity = "&quot;";
383.             new += strlen(entity);
384.             t = realloc(t, new);
```

```
385.             if (t == NULL)
386.             {
387.                 return NULL;
388.             }
389.             strcat(t, entity);
390.         }
391.
392.         // escape '
393.         else if (s[i] == '\'')
394.         {
395.             const char* entity = "&#039;";
396.             new += strlen(entity);
397.             t = realloc(t, new);
398.             if (t == NULL)
399.             {
400.                 return NULL;
401.             }
402.             strcat(t, entity);
403.         }
404.
405.         // escape <
406.         else if (s[i] == '<')
407.         {
408.             const char* entity = "&lt;";
409.             new += strlen(entity);
410.             t = realloc(t, new);
411.             if (t == NULL)
412.             {
413.                 return NULL;
414.             }
415.             strcat(t, entity);
416.         }
417.
418.         // escape >
419.         else if (s[i] == '>')
420.         {
421.             const char* entity = "&gt;";
422.             new += strlen(entity);
423.             t = realloc(t, new);
424.             if (t == NULL)
425.             {
426.                 return NULL;
427.             }
428.             strcat(t, entity);
429.         }
430.
431.         // don't escape
432.         else
```

```
        {
            strncat(t, s + i, 1);
        }
    }

    // escaped string
    return t;
}

/**
 * Checks, in order, whether index.php or index.html exists inside of path.
 * Returns path to first match if so, else NULL.
 */
char* indexes(const char* path)
{

    int path_length = strlen(path) + 1;
    char temp[path_length];
    strcpy(temp, path);
    // allocate memory on heap for new paths
    char* filepath = malloc(path_length + 11);
    if (filepath == NULL)
    {
        return NULL;
    }

    if ((strstr(temp, "index.php") != NULL) || (strstr(temp, "index.html") != NULL))
    {
        strcpy(filepath, temp);
        if (access(filepath, F_OK) != -1)
        {
            return filepath;
        }
    }
    else if (temp[path_length - 2]!='/')
    {
        strcpy(filepath, temp);
        strcat(filepath, "/");
    }
    else
    {
        strcpy(filepath, temp);

    }

    strcat(filepath, "index.php");
    if (access(filepath, F_OK) != -1)
    {
```

```
481.         return filepath;
482.     }
483.     else
484.     {
485.         free(filepath);
486.         filepath = malloc(path_length+12);
487.         strcpy(filepath,temp);
488.         strcat(filepath,"index.html");
489.         if (access(filepath, F_OK) != -1)
490.         {
491.             return filepath;
492.         }
493. 
494.     }
495.         free(filepath);
496. 
497.         return NULL;
498. }
499. 
500. /**
501.  * Interprets PHP file at path using query string.
502.  */
503. void interpret(const char* path, const char* query)
504. {
505.     // ensure path is readable
506.     if (access(path, R_OK) == -1)
507.     {
508.         error(403);
509.         return;
510.     }
511. 
512.     // open pipe to PHP interpreter
513.     char* format = "QUERY_STRING=\"%s\" REDIRECT_STATUS=200 SCRIPT_FILENAME=\"%s\" php-cgi";
514.     char command[strlen(format) + (strlen(path) - 2) + (strlen(query) - 2) + 1];
515.     if (sprintf(command, format, query, path) < 0)
516.     {
517.         error(500);
518.         return;
519.     }
520.     FILE* file = popen(command, "r");
521.     if (file == NULL)
522.     {
523.         error(500);
524.         return;
525.     }
526. 
527.     // load interpreter's content
528.     char* content;
```

```
529.     size_t length;
530.     if (load(file, &content, &length) == false)
531.     {
532.         error(500);
533.         return;
534.     }
535.
536.     // close pipe
537.     pclose(file);
538.
539.     // subtract php-cgi's headers from content's length to get body's length
540.     char* haystack = content;
541.     char* needle = strstr(haystack, "\r\n\r\n");
542.     if (needle == NULL)
543.     {
544.         free(content);
545.         error(500);
546.         return;
547.     }
548.
549.     // extract headers
550.     char headers[needle + 2 - haystack + 1];
551.     strncpy(headers, content, needle + 2 - haystack);
552.     headers[needle + 2 - haystack] = '\0';
553.
554.     // respond with interpreter's content
555.     respond(200, headers, needle + 4, length - (needle - haystack + 4));
556.
557.     // free interpreter's content
558.     free(content);
559. }
560.
561. /**
562.  * Responds to client with directory listing of path.
563.  */
564. void list(const char* path)
565. {
566.     // ensure path is readable and executable
567.     if (access(path, R_OK | X_OK) == -1)
568.     {
569.         error(403);
570.         return;
571.     }
572.
573.     // open directory
574.     DIR* dir = opendir(path);
575.     if (dir == NULL)
576.     {
```

```
577.         return;
578.     }
579.
580.     // buffer for list items
581.     char* list = malloc(1);
582.     list[0] = '\0';
583.
584.     // iterate over directory entries
585.     struct dirent** namelist = NULL;
586.     int n = scandir(path, &namelist, NULL, alphasort);
587.     for (int i = 0; i < n; i++)
588.     {
589.         // omit . from list
590.         if (strcmp(namelist[i]->d_name, ".") == 0)
591.         {
592.             continue;
593.         }
594.
595.         // escape entry's name
596.         char* name = htmlspecialchars(namelist[i]->d_name);
597.         if (name == NULL)
598.         {
599.             free(list);
600.             freedir(namelist, n);
601.             error(500);
602.             return;
603.         }
604.
605.         // append list item to buffer
606.         char* template = "<li><a href=\"%s\">%s</a></li>";
607.         list = realloc(list, strlen(list) + strlen(template) - 2 + strlen(name) - 2 + strlen(name) + 1);
608.         if (list == NULL)
609.         {
610.             free(name);
611.             freedir(namelist, n);
612.             error(500);
613.             return;
614.         }
615.         if (sprintf(list + strlen(list), template, name, name) < 0)
616.         {
617.             free(name);
618.             freedir(namelist, n);
619.             free(list);
620.             error(500);
621.             return;
622.         }
623.
624.         // free escaped name
```

```
625.         free(name);
626.     }
627.
628.     // free memory allocated by scandir
629.     freedir(namelist, n);
630.
631.     // prepare response
632.     const char* relative = path + strlen(root);
633.     char* template = "<html><head><title>%s</title></head><body><h1>%s</h1><ul>%s</ul></body></html>";
634.     char body[strlen(template) - 2 + strlen(relative) - 2 + strlen(relative) - 2 + strlen(list) + 1];
635.     int length = sprintf(body, template, relative, relative, list);
636.     if (length < 0)
637.     {
638.         free(list);
639.         closedir(dir);
640.         error(500);
641.         return;
642.     }
643.
644.     // free buffer
645.     free(list);
646.
647.     // close directory
648.     closedir(dir);
649.
650.     // respond with list
651.     char* headers = "Content-Type: text/html\r\n";
652.     respond(200, headers, body, length);
653. }
654.
655. /**
656.  * Loads a file into memory dynamically allocated on heap.
657.  * Stores address thereof in *content and length thereof in *length.
658.  */
659. bool load(FILE* file, BYTE** content, size_t* length)
660. {
661.     char* file_content = malloc(sizeof(char)*1000);
662.     if (file_content == NULL)
663.     {
664.         return false;
665.     }
666.
667.     int i = 0;
668.     for (int c = fgetc(file); c != EOF; c = fgetc(file))
669.     {
670.         if (i % 1000 == 0)
671.         {
672.             file_content = realloc(file_content, sizeof(char)*(1000 + i));
```

```
673.         }
674.
675.         file_content[i] = c;
676.         i++;
677.     }
678.     // add terminating string and increase the size by one
679.     //file_content[i] = '\0';
680.     i++;
681.
682.     // stores the address of the first byte of file contant in heap
683.     *content = file_content;
684.     *length = i;
685.
686.     return true;
687. }
688.
689. /**
690.  * Returns MIME type for supported extensions, else NULL.
691.  */
692. const char* lookup(const char* path)
693. {
694.     char* MIME = malloc(sizeof(char)*16);
695.     char* extension = strrchr(path,'.');
696.     if(strncasecmp(extension,".html",5)==0){
697.         strcpy(MIME,"text/html");
698.     }else if(strncasecmp(extension,".css",4)==0){
699.         strcpy(MIME,"text/css");
700.     }else if(strncasecmp(extension,".js",3)==0){
701.         strcpy(MIME,"text/javascript");
702.     }else if(strncasecmp(extension,".php",4)==0){
703.         strcpy(MIME,"text/x-php");
704.     }else if(strncasecmp(extension,".gif",4)==0){
705.         strcpy(MIME,"image/gif");
706.     }else if(strncasecmp(extension,".ico",4)==0){
707.         strcpy(MIME,"image/x-icon");
708.     }else if(strncasecmp(extension,".jpg",4)==0){
709.         strcpy(MIME,"image/jpeg");
710.     }else if(strncasecmp(extension,".png",4)==0){
711.         strcpy(MIME,"image/png");
712.     }else{
713.
714.         MIME = NULL;
715.     }
716.     return MIME;
717. }
718.
719. /**
720.  * Parses a request-line, storing its absolute-path at abs_path
```

```
 * and its query string at query, both of which are assumed
 * to be at least of length LimitRequestLine + 1.
 */
bool parse(const char* line, char* abs_path, char* query)
{
    char *path ;
    char* target;
    path = strchr(line,' ')+1;
    if(strncmp("GET ",line,4) != 0){
        error(405);
        return false;
    }
    if (path==NULL || (strstr(path,"\"")!=NULL)){
        error(400);
        return false;
    }
    if(path[0] !='/'){
        error(501);
        return false;
    }
    char *temp = strstr(path," ")+1;
    //printf("temp-%s-%i",temp,strcmp(temp,"HTTP/1.1\r\n"));
    if(temp==NULL || strcmp(temp,"HTTP/1.1\r\n")!=0){
        error(505);
        return false;
    }
    target = strtok(path," ");
    printf("Tar%s--",target);
    char* temp2 = strtok(target,"?");
    strcpy(abs_path,target);
    printf("%s",abs_path);
    temp2 = strtok(NULL,"?");
    if(temp2==NULL || strlen(temp2) < 3){
        query[0] = '\0';
    }else{
        strcpy(query,temp2);
        printf("quert%s-",query);
    }
    return true;
}
/**
 * Returns status code's reason phrase.
 *
 * http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6
 * https://tools.ietf.org/html/rfc2324
 */
const char* reason(unsigned short code)
{
```

```
769.     switch (code)
770.     {
771.         case 200: return "OK";
772.         case 301: return "Moved Permanently";
773.         case 400: return "Bad Request";
774.         case 403: return "Forbidden";
775.         case 404: return "Not Found";
776.         case 405: return "Method Not Allowed";
777.         case 414: return "Request-URI Too Long";
778.         case 418: return "I'm a teapot";
779.         case 500: return "Internal Server Error";
780.         case 501: return "Not Implemented";
781.         case 505: return "HTTP Version Not Supported";
782.         default: return NULL;
783.     }
784. }
785.
786. /**
787.  * Redirects client to uri.
788.  */
789. void redirect(const char* uri)
790. {
791.     char* template = "Location: %s\r\n";
792.     char headers[strlen(template) - 2 + strlen(uri) + 1];
793.     if (sprintf(headers, template, uri) < 0)
794.     {
795.         error(500);
796.         return;
797.     }
798.     respond(301, headers, NULL, 0);
799. }
800.
801. /**
802.  * Reads (without blocking) an HTTP request's headers into memory dynamically allocated on heap.
803.  * Stores address thereof in *message and length thereof in *length.
804.  */
805. bool request(char** message, size_t* length)
806. {
807.     // ensure socket is open
808.     if (cfd == -1)
809.     {
810.         return false;
811.     }
812.
813.     // initialize message and its length
814.     *message = NULL;
815.     *length = 0;
816.
```

```
817.     // read message
818.     while (*length < LimitRequestLine + LimitRequestFields * LimitRequestFieldSize + 4)
819.     {
820.         // read from socket
821.         BYTE buffer[BYTES];
822.         ssize_t bytes = read(cfd, buffer, BYTES);
823.         if (bytes < 0)
824.         {
825.             if (*message != NULL)
826.             {
827.                 free(*message);
828.                 *message = NULL;
829.             }
830.             *length = 0;
831.             break;
832.         }
833.
834.         // append bytes to message
835.         *message = realloc(*message, *length + bytes + 1);
836.         if (*message == NULL)
837.         {
838.             *length = 0;
839.             break;
840.         }
841.         memcpy(*message + *length, buffer, bytes);
842.         *length += bytes;
843.
844.         // null-terminate message thus far
845.         *(*message + *length) = '\0';
846.
847.         // search for CRLF CRLF
848.         int offset = (*length - bytes < 3) ? *length - bytes : 3;
849.         char* haystack = *message + *length - bytes - offset;
850.         char* needle = strstr(haystack, "\r\n\r\n");
851.         if (needle != NULL)
852.         {
853.             // trim to one CRLF and null-terminate
854.             *length = needle - *message + 2;
855.             *message = realloc(*message, *length + 1);
856.             if (*message == NULL)
857.             {
858.                 break;
859.             }
860.             *(*message + *length) = '\0';
861.
862.             // ensure request-line is no longer than LimitRequestLine
863.             haystack = *message;
864.             needle = strstr(haystack, "\r\n");
```

```
865.             if (needle == NULL || (needle - haystack + 2) > LimitRequestLine)
866.             {
867.                 break;
868.             }
869.
870.             // count fields in message
871.             int fields = 0;
872.             haystack = needle + 2;
873.             while (*haystack != '\0')
874.             {
875.                 // look for CRLF
876.                 needle = strstr(haystack, "\r\n");
877.                 if (needle == NULL)
878.                 {
879.                     break;
880.                 }
881.
882.
883.
884.                 // ensure field is no longer than LimitRequestFieldSize
885.                 if (needle - haystack + 2 > LimitRequestFieldSize)
886.                 {
887.                     break;
888.                 }
889.
890.                 // look beyond CRLF
891.                 haystack = needle + 2;
892.             }
893.
894.             // if we didn't get to end of message, we must have erred
895.             if (*haystack != '\0')
896.             {
897.                 break;
898.             }
899.
900.             // ensure message has no more than LimitRequestFields
901.             if (fields > LimitRequestFields)
902.             {
903.                 break;
904.             }
905.
906.             // valid
907.             return true;
908.         }
909.     }
910.
911.     // invalid
912.     if (*message != NULL)
```

```
    {
        free(*message);
    }
    *message = NULL;
    *length = 0;
    return false;
}

/**
 * Responds to a client with status code, headers, and body of specified length.
 */
void respond(int code, const char* headers, const char* body, size_t length)
{
    // determine Status-Line's phrase
    // http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1
    const char* phrase = reason(code);
    if (phrase == NULL)
    {
        return;
    }

    // respond with Status-Line
    if (dprintf(cfd, "HTTP/1.1 %i %s\r\n", code, phrase) < 0)
    {
        return;
    }

    // respond with headers
    if (dprintf(cfd, "%s", headers) < 0)
    {
        return;
    }

    // respond with CRLF
    if (dprintf(cfd, "\r\n") < 0)
    {
        return;
    }

    // respond with body
    if (write(cfd, body, length) == -1)
    {
        return;
    }

    // log response line
    if (code == 200)
    {
```

```
961.         // green
962.         printf("\033[32m");
963.     }
964.     else
965.     {
966.         // red
967.         printf("\033[33m");
968.     }
969.     printf("HTTP/1.1 %i %s", code, phrase);
970.     printf("\033[39m\n");
971. }
972.
973. /**
974.  * Starts server on specified port rooted at path.
975.  */
976. void start(short port, const char* path)
977. {
978.     // path to server's root
979.     root = realpath(path, NULL);
980.     if (root == NULL)
981.     {
982.         stop();
983.     }
984.
985.     // ensure root is executable
986.     if (access(root, X_OK) == -1)
987.     {
988.         stop();
989.     }
990.
991.     // announce root
992.     printf("\033[33m");
993.     printf("Using %s for server's root", root);
994.     printf("\033[39m\n");
995.
996.     // create a socket
997.     sfd = socket(AF_INET, SOCK_STREAM, 0);
998.     if (sfd == -1)
999.     {
1000.         stop();
1001.     }
1002.
1003.     // allow reuse of address (to avoid "Address already in use")
1004.     int optval = 1;
1005.     setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
1006.
1007.     // assign name to socket
1008.     struct sockaddr_in serv_addr;
```

```
1009.      memset(&serv_addr, 0, sizeof(serv_addr));
1010.      serv_addr.sin_family = AF_INET;
1011.      serv_addr.sin_port = htons(port);
1012.      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
1013.      if (bind(sfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) == -1)
1014.      {
1015.          printf("\033[33m");
1016.          printf("Port %i already in use", port);
1017.          printf("\033[39m\n");
1018.          stop();
1019.      }
1020.
1021.      // listen for connections
1022.      if (listen(sfd, SOMAXCONN) == -1)
1023.      {
1024.          stop();
1025.      }
1026.
1027.      // announce port in use
1028.      struct sockaddr_in addr;
1029.      socklen_t addrlen = sizeof(addr);
1030.      if (getsockname(sfd, (struct sockaddr*) &addr, &addrlen) == -1)
1031.      {
1032.          stop();
1033.      }
1034.      printf("\033[33m");
1035.      printf("Listening on port %i", ntohs(addr.sin_port));
1036.      printf("\033[39m\n");
1037.  }
1038.
1039.  /**
1040.   * Stop server, deallocating any resources.
1041.   */
1042.  void stop(void)
1043.  {
1044.      // preserve errno across this function's library calls
1045.      int errsv = errno;
1046.
1047.      // announce stop
1048.      printf("\033[33m");
1049.      printf("Stopping server\n");
1050.      printf("\033[39m");
1051.
1052.      // free root, which was allocated by realpath
1053.      if (root != NULL)
1054.      {
1055.          free(root);
1056.      }
```

```
1057.
1058.     // close server socket
1059.     if (sfd != -1)
1060.     {
1061.         close(sfd);
1062.     }
1063.
1064.     // stop server
1065.     exit(errsv);
1066. }
1067.
1068. /**
1069.  * Transfers file at path with specified type to client.
1070.  */
1071. void transfer(const char* path, const char* type)
1072. {
1073.     // ensure path is readable
1074.     if (access(path, R_OK) == -1)
1075.     {
1076.         error(403);
1077.         return;
1078.     }
1079.
1080.     // open file
1081.     FILE* file = fopen(path, "r");
1082.     if (file == NULL)
1083.     {
1084.         error(500);
1085.         return;
1086.     }
1087.
1088.     // load file's content
1089.     BYTE* content;
1090.     size_t length;
1091.     if (load(file, &content, &length) == false)
1092.     {
1093.         error(500);
1094.         return;
1095.     }
1096.
1097.     // close file
1098.     fclose(file);
1099.
1100.     // prepare response
1101.     char* template = "Content-Type: %s\r\n";
1102.     char headers[strlen(template) - 2 + strlen(type) + 1];
1103.     if (sprintf(headers, template, type) < 0)
1104.     {
```

```
1105.         error(500);
1106.         return;
1107.     }
1108.
1109.     // respond with file's content
1110.     respond(200, headers, content, length);
1111.
1112.     // free file's content
1113.     free(content);
1114. }
1115.
1116. /**
1117.  * URL-decodes string, returning dynamically allocated memory for decoded string
1118.  * that must be deallocated by caller.
1119.  */
1120. char* urldecode(const char* s)
1121. {
1122.     // check whether s is NULL
1123.     if (s == NULL)
1124.     {
1125.         return NULL;
1126.     }
1127.
1128.     // allocate enough (zeroed) memory for an undecoded copy of s
1129.     char* t = calloc(strlen(s) + 1, 1);
1130.     if (t == NULL)
1131.     {
1132.         return NULL;
1133.     }
1134.
1135.     // iterate over characters in s, decoding percent-encoded octets, per
1136.     // https://www.ietf.org/rfc/rfc3986.txt
1137.     for (int i = 0, j = 0, n = strlen(s); i < n; i++, j++)
1138.     {
1139.         if (s[i] == '%' && i < n - 2)
1140.         {
1141.             char octet[3];
1142.             octet[0] = s[i + 1];
1143.             octet[1] = s[i + 2];
1144.             octet[2] = '\0';
1145.             t[j] = (char) strtol(octet, NULL, 16);
1146.             i += 2;
1147.         }
1148.         else if (s[i] == '+')
1149.         {
1150.             t[j] = ' ';
1151.         }
1152.         else
```

```
1153.         {
1154.             t[j] = s[i];
1155.         }
1156.     }
1157.
1158.     // escaped string
1159.     return t;
1160. }
1161.
```

```c
//
// server.c
//
// Computer Science 50
// Problem Set 6
//

// feature test macro requirements
#define _GNU_SOURCE
#define _XOPEN_SOURCE 700
#define _XOPEN_SOURCE_EXTENDED

// limits on an HTTP request's size, based on Apache's
// http://httpd.apache.org/docs/2.2/mod/core.html
#define LimitRequestFields 50
#define LimitRequestFieldSize 4094
#define LimitRequestLine 8190

// number of bytes for buffers
#define BYTES 512

// header files
#include <arpa/inet.h>
#include <dirent.h>
#include <errno.h>
#include <limits.h>
#include <math.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

// types
typedef char BYTE;

// prototypes
bool connected(void);
void error(unsigned short code);
void freedir(struct dirent** namelist, int n);
void handler(int signal);
char* htmlspecialchars(const char* s);
char* indexes(const char* path);
```

```
49. void interpret(const char* path, const char* query);
50. void list(const char* path);
51. bool load(FILE* file, BYTE** content, size_t* length);
52. const char* lookup(const char* path);
53. bool parse(const char* line, char* path, char* query);
54. const char* reason(unsigned short code);
55. void redirect(const char* uri);
56. bool request(char** message, size_t* length);
57. void respond(int code, const char* headers, const char* body, size_t length);
58. void start(short port, const char* path);
59. void stop(void);
60. void transfer(const char* path, const char* type);
61. char* urldecode(const char* s);
62.
63. // server's root
64. char* root = NULL;
65.
66. // file descriptor for sockets
67. int cfd = -1, sfd = -1;
68.
69. // flag indicating whether control-c has been heard
70. bool signaled = false;
71.
72. int main(int argc, char* argv[])
73. {
74.     // a global variable defined in errno.h that's "set by system
75.     // calls and some library functions [to a nonzero value]
76.     // in the event of an error to indicate what went wrong"
77.     errno = 0;
78.
79.     // default to port 8080
80.     int port = 8080;
81.
82.     // usage
83.     const char* usage = "Usage: server [-p port] /path/to/root";
84.
85.     // parse command-line arguments
86.     int opt;
87.     while ((opt = getopt(argc, argv, "hp:")) != -1)
88.     {
89.         switch (opt)
90.         {
91.             // -h
92.             case 'h':
93.                 printf("%s\n", usage);
94.                 return 0;
95.
96.             // -p port
```

```
 97.             case 'p':
 98.                 port = atoi(optarg);
 99.                 break;
100.         }
101.     }
102.
103.     // ensure port is a non-negative short and path to server's root is specified
104.     if (port < 0 || port > SHRT_MAX || argv[optind] == NULL || strlen(argv[optind]) == 0)
105.     {
106.         // announce usage
107.         printf("%s\n", usage);
108.
109.         // return 2 just like bash's builtins
110.         return 2;
111.     }
112.
113.     // start server
114.     start(port, argv[optind]);
115.
116.     // listen for SIGINT (aka control-c)
117.     struct sigaction act;
118.     act.sa_handler = handler;
119.     act.sa_flags = 0;
120.     sigemptyset(&act.sa_mask);
121.     sigaction(SIGINT, &act, NULL);
122.
123.     // a message and its length
124.     char* message = NULL;
125.     size_t length = 0;
126.
127.     // path requested
128.     char* path = NULL;
129.
130.     // accept connections one at a time
131.     while (true)
132.     {
133.         // free last path, if any
134.         if (path != NULL)
135.         {
136.             free(path);
137.             path = NULL;
138.         }
139.
140.         // free last message, if any
141.         if (message != NULL)
142.         {
143.             free(message);
144.             message = NULL;
```

```
145.         }
146.         length = 0;
147.
148.         // close last client's socket, if any
149.         if (cfd != -1)
150.         {
151.             close(cfd);
152.             cfd = -1;
153.         }
154.
155.         // check for control-c
156.         if (signaled)
157.         {
158.             stop();
159.         }
160.
161.         // check whether client has connected
162.         if (connected())
163.         {
164.             // check for request
165.             if (request(&message, &length))
166.             {
167.                 // extract message's request-line
168.                 // http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
169.                 const char* haystack = message;
170.                 const char* needle = strstr(haystack, "\r\n");
171.                 if (needle == NULL)
172.                 {
173.                     error(500);
174.                     continue;
175.                 }
176.                 char line[needle - haystack + 2 + 1];
177.                 strncpy(line, haystack, needle - haystack + 2);
178.                 line[needle - haystack + 2] = '\0';
179.
180.                 // log request-line
181.                 printf("%s", line);
182.
183.                 // parse request-line
184.                 char abs_path[LimitRequestLine + 1];
185.                 char query[LimitRequestLine + 1];
186.                 if (parse(line, abs_path, query))
187.                 {
188.                     // URL-decode absolute-path
189.                     char* p = urldecode(abs_path);
190.                     if (p == NULL)
191.                     {
192.                         error(500);
```

```
193.                     continue;
194.                 }
195.
196.                 // resolve absolute-path to local path
197.                 path = malloc(strlen(root) + strlen(p) + 1);
198.                 if (path == NULL)
199.                 {
200.                     error(500);
201.                     continue;
202.                 }
203.                 strcpy(path, root);
204.                 strcat(path, p);
205.                 free(p);
206.
207.                 // ensure path exists
208.                 if (access(path, F_OK) == -1)
209.                 {
210.                     printf("Acess -%s-",abs_path);
211.                     error(404);
212.                     continue;
213.                 }
214.
215.                 // if path to directory
216.                 struct stat sb;
217.                 if (stat(path, &sb) == 0 && S_ISDIR(sb.st_mode))
218.                 {
219.                     // redirect from absolute-path to absolute-path/
220.                     if (abs_path[strlen(abs_path) - 1] != '/')
221.                     {
222.                         char uri[strlen(abs_path) + 1 + 1];
223.                         strcpy(uri, abs_path);
224.                         strcat(uri, "/");
225.                         redirect(uri);
226.                         continue;
227.                     }
228.
229.                     // use path/index.php or path/index.html, if present, instead of directory's path
230.                     char* index = indexes(path);
231.                     if (index != NULL)
232.                     {
233.                         free(path);
234.                         path = index;
235.                     }
236.
237.                     // list contents of directory
238.                     else
239.                     {
240.                         list(path);
```

```
                            continue;
                        }
                    }

                    // look up MIME type for file at path
                    const char* type = lookup(path);
                    if (type == NULL)
                    {
                        error(501);
                        continue;
                    }

                    // interpret PHP script at path
                    if (strcasecmp("text/x-php", type) == 0)
                    {
                        interpret(path, query);
                    }

                    // transfer file at path
                    else
                    {
                        transfer(path, type);
                    }
                }
            }
        }
    }
}

/**
 * Checks (without blocking) whether a client has connected to server.
 * Returns true iff so.
 */
bool connected(void)
{
    struct sockaddr_in cli_addr;
    memset(&cli_addr, 0, sizeof(cli_addr));
    socklen_t cli_len = sizeof(cli_addr);
    cfd = accept(sfd, (struct sockaddr*) &cli_addr, &cli_len);
    if (cfd == -1)
    {
        return false;
    }
    return true;
}

/**
 * Responds to client with specified status code.
```

```
 */
void error(unsigned short code)
{
    // determine code's reason-phrase
    const char* phrase = reason(code);
    if (phrase == NULL)
    {
        return;
    }

    // template for response's content
    char* template = "<html><head><title>%i %s</title></head><body><h1>%i %s</h1></body></html>";

    // render template
    char body[(strlen(template) - 2 - ((int) log10(code) + 1) - 2 + strlen(phrase)) * 2 + 1];
    int length = sprintf(body, template, code, phrase, code, phrase);
    if (length < 0)
    {
        body[0] = '\0';
        length = 0;
    }

    // respond with error
    char* headers = "Content-Type: text/html\r\n";
    respond(code, headers, body, length);
}

/**
 * Frees memory allocated by scandir.
 */
void freedir(struct dirent** namelist, int n)
{
    if (namelist != NULL)
    {
        for (int i = 0; i < n; i++)
        {
            free(namelist[i]);
        }
        free(namelist);
    }
}

/**
 * Handles signals.
 */
void handler(int signal)
{
    // control-c
```

```
    if (signal == SIGINT)
    {
        signaled = true;
    }
}

/**
 * Escapes string for HTML. Returns dynamically allocated memory for escaped
 * string that must be deallocated by caller.
 */
char* htmlspecialchars(const char* s)
{
    // ensure s is not NULL
    if (s == NULL)
    {
        return NULL;
    }

    // allocate enough space for an unescaped copy of s
    char* t = malloc(strlen(s) + 1);
    if (t == NULL)
    {
        return NULL;
    }
    t[0] = '\0';

    // iterate over characters in s, escaping as needed
    for (int i = 0, old = strlen(s), new = old; i < old; i++)
    {
        // escape &
        if (s[i] == '&')
        {
            const char* entity = "&amp;";
            new += strlen(entity);
            t = realloc(t, new);
            if (t == NULL)
            {
                return NULL;
            }
            strcat(t, entity);
        }

        // escape "
        else if (s[i] == '"')
        {
            const char* entity = "&quot;";
            new += strlen(entity);
            t = realloc(t, new);
```

```
385.             if (t == NULL)
386.             {
387.                 return NULL;
388.             }
389.             strcat(t, entity);
390.         }
391.
392.         // escape '
393.         else if (s[i] == '\'')
394.         {
395.             const char* entity = "&#039;";
396.             new += strlen(entity);
397.             t = realloc(t, new);
398.             if (t == NULL)
399.             {
400.                 return NULL;
401.             }
402.             strcat(t, entity);
403.         }
404.
405.         // escape <
406.         else if (s[i] == '<')
407.         {
408.             const char* entity = "&lt;";
409.             new += strlen(entity);
410.             t = realloc(t, new);
411.             if (t == NULL)
412.             {
413.                 return NULL;
414.             }
415.             strcat(t, entity);
416.         }
417.
418.         // escape >
419.         else if (s[i] == '>')
420.         {
421.             const char* entity = "&gt;";
422.             new += strlen(entity);
423.             t = realloc(t, new);
424.             if (t == NULL)
425.             {
426.                 return NULL;
427.             }
428.             strcat(t, entity);
429.         }
430.
431.         // don't escape
432.         else
```

```
433.         {
434.             strncat(t, s + i, 1);
435.         }
436.     }
437.
438.     // escaped string
439.     return t;
440. }
441.
442. /**
443.  * Checks, in order, whether index.php or index.html exists inside of path.
444.  * Returns path to first match if so, else NULL.
445.  */
446. char* indexes(const char* path)
447. {
448.
449.     int path_length = strlen(path) + 1;
450.     char temp[path_length];
451.     strcpy(temp, path);
452.     // allocate memory on heap for new paths
453.     char* filepath = malloc(path_length + 11);
454.     if (filepath == NULL)
455.     {
456.         return NULL;
457.     }
458.
459.     if ((strstr(temp, "index.php") != NULL) || (strstr(temp, "index.html") != NULL))
460.     {
461.         strcpy(filepath, temp);
462.         if (access(filepath, F_OK) != -1)
463.         {
464.             return filepath;
465.         }
466.     }
467.     else if (temp[path_length - 2]!='/')
468.     {
469.         strcpy(filepath, temp);
470.         strcat(filepath, "/");
471.     }
472.     else
473.     {
474.         strcpy(filepath, temp);
475.
476.     }
477.
478.     strcat(filepath, "index.php");
479.     if (access(filepath, F_OK) != -1)
480.     {
```

```
481.         return filepath;
482.     }
483.     else
484.     {
485.         free(filepath);
486.         filepath = malloc(path_length+12);
487.         strcpy(filepath,temp);
488.         strcat(filepath,"index.html");
489.         if (access(filepath, F_OK) != -1)
490.         {
491.             return filepath;
492.         }
493. 
494.     }
495.         free(filepath);
496. 
497.         return NULL;
498. }
499. 
500. /**
501.  * Interprets PHP file at path using query string.
502.  */
503. void interpret(const char* path, const char* query)
504. {
505.     // ensure path is readable
506.     if (access(path, R_OK) == -1)
507.     {
508.         error(403);
509.         return;
510.     }
511. 
512.     // open pipe to PHP interpreter
513.     char* format = "QUERY_STRING=\"%s\" REDIRECT_STATUS=200 SCRIPT_FILENAME=\"%s\" php-cgi";
514.     char command[strlen(format) + (strlen(path) - 2) + (strlen(query) - 2) + 1];
515.     if (sprintf(command, format, query, path) < 0)
516.     {
517.         error(500);
518.         return;
519.     }
520.     FILE* file = popen(command, "r");
521.     if (file == NULL)
522.     {
523.         error(500);
524.         return;
525.     }
526. 
527.     // load interpreter's content
528.     char* content;
```

```
529.     size_t length;
530.     if (load(file, &content, &length) == false)
531.     {
532.         error(500);
533.         return;
534.     }
535. 
536.     // close pipe
537.     pclose(file);
538. 
539.     // subtract php-cgi's headers from content's length to get body's length
540.     char* haystack = content;
541.     char* needle = strstr(haystack, "\r\n\r\n");
542.     if (needle == NULL)
543.     {
544.         free(content);
545.         error(500);
546.         return;
547.     }
548. 
549.     // extract headers
550.     char headers[needle + 2 - haystack + 1];
551.     strncpy(headers, content, needle + 2 - haystack);
552.     headers[needle + 2 - haystack] = '\0';
553. 
554.     // respond with interpreter's content
555.     respond(200, headers, needle + 4, length - (needle - haystack + 4));
556. 
557.     // free interpreter's content
558.     free(content);
559. }
560. 
561. /**
562.  * Responds to client with directory listing of path.
563.  */
564. void list(const char* path)
565. {
566.     // ensure path is readable and executable
567.     if (access(path, R_OK | X_OK) == -1)
568.     {
569.         error(403);
570.         return;
571.     }
572. 
573.     // open directory
574.     DIR* dir = opendir(path);
575.     if (dir == NULL)
576.     {
```

```
577.         return;
578.     }
579.
580.     // buffer for list items
581.     char* list = malloc(1);
582.     list[0] = '\0';
583.
584.     // iterate over directory entries
585.     struct dirent** namelist = NULL;
586.     int n = scandir(path, &namelist, NULL, alphasort);
587.     for (int i = 0; i < n; i++)
588.     {
589.         // omit . from list
590.         if (strcmp(namelist[i]->d_name, ".") == 0)
591.         {
592.             continue;
593.         }
594.
595.         // escape entry's name
596.         char* name = htmlspecialchars(namelist[i]->d_name);
597.         if (name == NULL)
598.         {
599.             free(list);
600.             freedir(namelist, n);
601.             error(500);
602.             return;
603.         }
604.
605.         // append list item to buffer
606.         char* template = "<li><a href=\"%s\">%s</a></li>";
607.         list = realloc(list, strlen(list) + strlen(template) - 2 + strlen(name) - 2 + strlen(name) + 1);
608.         if (list == NULL)
609.         {
610.             free(name);
611.             freedir(namelist, n);
612.             error(500);
613.             return;
614.         }
615.         if (sprintf(list + strlen(list), template, name, name) < 0)
616.         {
617.             free(name);
618.             freedir(namelist, n);
619.             free(list);
620.             error(500);
621.             return;
622.         }
623.
624.         // free escaped name
```

```
625.         free(name);
626.     }
627.
628.     // free memory allocated by scandir
629.     freedir(namelist, n);
630.
631.     // prepare response
632.     const char* relative = path + strlen(root);
633.     char* template = "<html><head><title>%s</title></head><body><h1>%s</h1><ul>%s</ul></body></html>";
634.     char body[strlen(template) - 2 + strlen(relative) - 2 + strlen(relative) - 2 + strlen(list) + 1];
635.     int length = sprintf(body, template, relative, relative, list);
636.     if (length < 0)
637.     {
638.         free(list);
639.         closedir(dir);
640.         error(500);
641.         return;
642.     }
643.
644.     // free buffer
645.     free(list);
646.
647.     // close directory
648.     closedir(dir);
649.
650.     // respond with list
651.     char* headers = "Content-Type: text/html\r\n";
652.     respond(200, headers, body, length);
653. }
654.
655. /**
656.  * Loads a file into memory dynamically allocated on heap.
657.  * Stores address thereof in *content and length thereof in *length.
658.  */
659. bool load(FILE* file, BYTE** content, size_t* length)
660. {
661.     unsigned int capacity = 0;
662.     char* buffer = NULL;
663.     int c;
664.     *length = 0;
665.     for (c = fgetc(file); c != EOF; c = fgetc(file))
666.     {
667.         // new capacity
668.         if (*length + 1 > capacity)
669.         {
670.             if (capacity == 0){
671.                 capacity = 2;
672.             }
```

```
673.             else{
674.                 capacity *= 2;
675.             }
676.             //reallocating memory to buffer
677.             buffer = realloc(buffer, capacity* sizeof(BYTE));
678.             if (buffer == NULL)
679.             {
680.                 free(buffer);
681.                 return false;
682.             }
683.         }
684.         buffer[*length] = c;
685.         *length=*length+1;
686.     }
687.     //allocating to new buffer to save space
688.     char* new_buffer = malloc((*length + 1) * sizeof(BYTE));
689.     memcpy(new_buffer, buffer, *length);
690.     free(buffer);
691. 
692.     *content = new_buffer;
693. 
694.     return true;
695. }
696. 
697. /**
698.  * Returns MIME type for supported extensions, else NULL.
699.  */
700. const char* lookup(const char* path)
701. {
702.     char* MIME = malloc(sizeof(char)*16);
703.     char* extension = strrchr(path,'.');
704.     if(strncasecmp(extension,".html",5)==0){
705.         strcpy(MIME,"text/html");
706.     }else if(strncasecmp(extension,".css",4)==0){
707.         strcpy(MIME,"text/css");
708.     }else if(strncasecmp(extension,".js",3)==0){
709.         strcpy(MIME,"text/javascript");
710.     }else if(strncasecmp(extension,".php",4)==0){
711.         strcpy(MIME,"text/x-php");
712.     }else if(strncasecmp(extension,".gif",4)==0){
713.         strcpy(MIME,"image/gif");
714.     }else if(strncasecmp(extension,".ico",4)==0){
715.         strcpy(MIME,"image/x-icon");
716.     }else if(strncasecmp(extension,".jpg",4)==0){
717.         strcpy(MIME,"image/jpeg");
718.     }else if(strncasecmp(extension,".png",4)==0){
719.         strcpy(MIME,"image/png");
720.     }else{
```

```
721.
722.         MIME = NULL;
723.     }
724.     return MIME;
725. }
726.
727. /**
728.  * Parses a request-line, storing its absolute-path at abs_path
729.  * and its query string at query, both of which are assumed
730.  * to be at least of length LimitRequestLine + 1.
731.  */
732. bool parse(const char* line, char* abs_path, char* query)
733. {
734.     char *path ;
735.     char* target;
736.     path = strchr(line,' ')+1;
737.     if(strncmp("GET ",line,4) != 0){
738.         error(405);
739.         return false;
740.     }
741.     if (path==NULL || (strstr(path,"\"")!=NULL)){
742.         error(400);
743.         return false;
744.     }
745.     if(path[0] !='/'){
746.         error(501);
747.         return false;
748.     }
749.     char *temp = strstr(path," ")+1;
750.     if(temp==NULL || strcmp(temp,"HTTP/1.1\r\n")!=0){
751.         error(505);
752.         return false;
753.     }
754.     target = strtok(path," ");
755.     char* temp2 = strtok(target,"?");
756.     strcpy(abs_path,target);
757.     temp2 = strtok(NULL,"?");
758.     if(temp2==NULL || strlen(temp2) < 3){
759.         query[0] = '\0';
760.     }else{
761.         strcpy(query,temp2);
762.     }
763.     return true;
764. }
765. /**
766.  * Returns status code's reason phrase.
767.  *
768.  * http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6
```

```
 * https://tools.ietf.org/html/rfc2324
 */
const char* reason(unsigned short code)
{
    switch (code)
    {
        case 200: return "OK";
        case 301: return "Moved Permanently";
        case 400: return "Bad Request";
        case 403: return "Forbidden";
        case 404: return "Not Found";
        case 405: return "Method Not Allowed";
        case 414: return "Request-URI Too Long";
        case 418: return "I'm a teapot";
        case 500: return "Internal Server Error";
        case 501: return "Not Implemented";
        case 505: return "HTTP Version Not Supported";
        default: return NULL;
    }
}

/**
 * Redirects client to uri.
 */
void redirect(const char* uri)
{
    char* template = "Location: %s\r\n";
    char headers[strlen(template) - 2 + strlen(uri) + 1];
    if (sprintf(headers, template, uri) < 0)
    {
        error(500);
        return;
    }
    respond(301, headers, NULL, 0);
}

/**
 * Reads (without blocking) an HTTP request's headers into memory dynamically allocated on heap.
 * Stores address thereof in *message and length thereof in *length.
 */
bool request(char** message, size_t* length)
{
    // ensure socket is open
    if (cfd == -1)
    {
        return false;
    }

```

```c
    // initialize message and its length
    *message = NULL;
    *length = 0;

    // read message
    while (*length < LimitRequestLine + LimitRequestFields * LimitRequestFieldSize + 4)
    {
        // read from socket
        BYTE buffer[BYTES];
        ssize_t bytes = read(cfd, buffer, BYTES);
        if (bytes < 0)
        {
            if (*message != NULL)
            {
                free(*message);
                *message = NULL;
            }
            *length = 0;
            break;
        }

        // append bytes to message
        *message = realloc(*message, *length + bytes + 1);
        if (*message == NULL)
        {
            *length = 0;
            break;
        }
        memcpy(*message + *length, buffer, bytes);
        *length += bytes;

        // null-terminate message thus far
        *(*message + *length) = '\0';

        // search for CRLF CRLF
        int offset = (*length - bytes < 3) ? *length - bytes : 3;
        char* haystack = *message + *length - bytes - offset;
        char* needle = strstr(haystack, "\r\n\r\n");
        if (needle != NULL)
        {
            // trim to one CRLF and null-terminate
            *length = needle - *message + 2;
            *message = realloc(*message, *length + 1);
            if (*message == NULL)
            {
                break;
            }
            *(*message + *length) = '\0';
```

```
865.
866.             // ensure request-line is no longer than LimitRequestLine
867.             haystack = *message;
868.             needle = strstr(haystack, "\r\n");
869.             if (needle == NULL || (needle - haystack + 2) > LimitRequestLine)
870.             {
871.                 break;
872.             }
873.
874.             // count fields in message
875.             int fields = 0;
876.             haystack = needle + 2;
877.             while (*haystack != '\0')
878.             {
879.                 // look for CRLF
880.                 needle = strstr(haystack, "\r\n");
881.                 if (needle == NULL)
882.                 {
883.                     break;
884.                 }
885.
886.
887.
888.                 // ensure field is no longer than LimitRequestFieldSize
889.                 if (needle - haystack + 2 > LimitRequestFieldSize)
890.                 {
891.                     break;
892.                 }
893.
894.                 // look beyond CRLF
895.                 haystack = needle + 2;
896.             }
897.
898.             // if we didn't get to end of message, we must have erred
899.             if (*haystack != '\0')
900.             {
901.                 break;
902.             }
903.
904.             // ensure message has no more than LimitRequestFields
905.             if (fields > LimitRequestFields)
906.             {
907.                 break;
908.             }
909.
910.             // valid
911.             return true;
912.         }
```

```
913.     }
914.
915.     // invalid
916.     if (*message != NULL)
917.     {
918.         free(*message);
919.     }
920.     *message = NULL;
921.     *length = 0;
922.     return false;
923. }
924.
925. /**
926.  * Responds to a client with status code, headers, and body of specified length.
927.  */
928. void respond(int code, const char* headers, const char* body, size_t length)
929. {
930.     // determine Status-Line's phrase
931.     // http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1
932.     const char* phrase = reason(code);
933.     if (phrase == NULL)
934.     {
935.         return;
936.     }
937.
938.     // respond with Status-Line
939.     if (dprintf(cfd, "HTTP/1.1 %i %s\r\n", code, phrase) < 0)
940.     {
941.         return;
942.     }
943.
944.     // respond with headers
945.     if (dprintf(cfd, "%s", headers) < 0)
946.     {
947.         return;
948.     }
949.
950.     // respond with CRLF
951.     if (dprintf(cfd, "\r\n") < 0)
952.     {
953.         return;
954.     }
955.
956.     // respond with body
957.     if (write(cfd, body, length) == -1)
958.     {
959.         return;
960.     }
```

```
961.
962.     // log response line
963.     if (code == 200)
964.     {
965.         // green
966.         printf("\033[32m");
967.     }
968.     else
969.     {
970.         // red
971.         printf("\033[33m");
972.     }
973.     printf("HTTP/1.1 %i %s", code, phrase);
974.     printf("\033[39m\n");
975. }
976.
977. /**
978.  * Starts server on specified port rooted at path.
979.  */
980. void start(short port, const char* path)
981. {
982.     // path to server's root
983.     root = realpath(path, NULL);
984.     if (root == NULL)
985.     {
986.         stop();
987.     }
988.
989.     // ensure root is executable
990.     if (access(root, X_OK) == -1)
991.     {
992.         stop();
993.     }
994.
995.     // announce root
996.     printf("\033[33m");
997.     printf("Using %s for server's root", root);
998.     printf("\033[39m\n");
999.
1000.     // create a socket
1001.     sfd = socket(AF_INET, SOCK_STREAM, 0);
1002.     if (sfd == -1)
1003.     {
1004.         stop();
1005.     }
1006.
1007.     // allow reuse of address (to avoid "Address already in use")
1008.     int optval = 1;
```

```
1009.     setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
1010.
1011.     // assign name to socket
1012.     struct sockaddr_in serv_addr;
1013.     memset(&serv_addr, 0, sizeof(serv_addr));
1014.     serv_addr.sin_family = AF_INET;
1015.     serv_addr.sin_port = htons(port);
1016.     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
1017.     if (bind(sfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) == -1)
1018.     {
1019.         printf("\033[33m");
1020.         printf("Port %i already in use", port);
1021.         printf("\033[39m\n");
1022.         stop();
1023.     }
1024.
1025.     // listen for connections
1026.     if (listen(sfd, SOMAXCONN) == -1)
1027.     {
1028.         stop();
1029.     }
1030.
1031.     // announce port in use
1032.     struct sockaddr_in addr;
1033.     socklen_t addrlen = sizeof(addr);
1034.     if (getsockname(sfd, (struct sockaddr*) &addr, &addrlen) == -1)
1035.     {
1036.         stop();
1037.     }
1038.     printf("\033[33m");
1039.     printf("Listening on port %i", ntohs(addr.sin_port));
1040.     printf("\033[39m\n");
1041. }
1042.
1043. /**
1044.  * Stop server, deallocating any resources.
1045.  */
1046. void stop(void)
1047. {
1048.     // preserve errno across this function's library calls
1049.     int errsv = errno;
1050.
1051.     // announce stop
1052.     printf("\033[33m");
1053.     printf("Stopping server\n");
1054.     printf("\033[39m");
1055.
1056.     // free root, which was allocated by realpath
```

```
1057.     if (root != NULL)
1058.     {
1059.         free(root);
1060.     }
1061.
1062.     // close server socket
1063.     if (sfd != -1)
1064.     {
1065.         close(sfd);
1066.     }
1067.
1068.     // stop server
1069.     exit(errsv);
1070. }
1071.
1072. /**
1073.  * Transfers file at path with specified type to client.
1074.  */
1075. void transfer(const char* path, const char* type)
1076. {
1077.     // ensure path is readable
1078.     if (access(path, R_OK) == -1)
1079.     {
1080.         error(403);
1081.         return;
1082.     }
1083.
1084.     // open file
1085.     FILE* file = fopen(path, "r");
1086.     if (file == NULL)
1087.     {
1088.         error(500);
1089.         return;
1090.     }
1091.
1092.     // load file's content
1093.     BYTE* content;
1094.     size_t length;
1095.     if (load(file, &content, &length) == false)
1096.     {
1097.         error(500);
1098.         return;
1099.     }
1100.
1101.     // close file
1102.     fclose(file);
1103.
1104.     // prepare response
```

```
1105.     char* template = "Content-Type: %s\r\n";
1106.     char headers[strlen(template) - 2 + strlen(type) + 1];
1107.     if (sprintf(headers, template, type) < 0)
1108.     {
1109.         error(500);
1110.         return;
1111.     }
1112.
1113.     // respond with file's content
1114.     respond(200, headers, content, length);
1115.
1116.     // free file's content
1117.     free(content);
1118. }
1119.
1120. /**
1121.  * URL-decodes string, returning dynamically allocated memory for decoded string
1122.  * that must be deallocated by caller.
1123.  */
1124. char* urldecode(const char* s)
1125. {
1126.     // check whether s is NULL
1127.     if (s == NULL)
1128.     {
1129.         return NULL;
1130.     }
1131.
1132.     // allocate enough (zeroed) memory for an undecoded copy of s
1133.     char* t = calloc(strlen(s) + 1, 1);
1134.     if (t == NULL)
1135.     {
1136.         return NULL;
1137.     }
1138.
1139.     // iterate over characters in s, decoding percent-encoded octets, per
1140.     // https://www.ietf.org/rfc/rfc3986.txt
1141.     for (int i = 0, j = 0, n = strlen(s); i < n; i++, j++)
1142.     {
1143.         if (s[i] == '%' && i < n - 2)
1144.         {
1145.             char octet[3];
1146.             octet[0] = s[i + 1];
1147.             octet[1] = s[i + 2];
1148.             octet[2] = '\0';
1149.             t[j] = (char) strtol(octet, NULL, 16);
1150.             i += 2;
1151.         }
1152.         else if (s[i] == '+')
```

```
1153.         {
1154.             t[j] = ' ';
1155.         }
1156.         else
1157.         {
1158.             t[j] = s[i];
1159.         }
1160.     }
1161.
1162.     // escaped string
1163.     return t;
1164. }
1165.
```

```c
#include <string.h>
#include <stdio.h>

int main()
{
   char str[80] = " /hello.php?name=hkhiuhi HTTP/1.1";
   const char s[2] = "? ";
   char *token;
  // int i=0;
   /* get the first token */
   token = strtok(str, s);
   printf( "%s--\n", token );
   token = strtok(NULL, s);
   /* walk through other tokens */

      printf( "%s--\n", token );

   return(0);
}
```