```c
#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <stdlib.h> /* for atoi() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
#define RCVBUFSIZE 32 /* Size of receive buffer */

void DieWithError(char *errorMessage); /* Error handling function */

int main(int argc, char *argv[]) {
    /* local declarations */ …
    /* command line processing */ …
    /* socket creation */ …
    /* connection establishment */ …
    /* socket sending */ …
    /* socket receiving */ …
    printf("\n"); /* Print a final linefeed */
    close(sock);
    exit(0);
}
```

DieWithError is implemented in a separate source file.
Note the client pattern: socket creation, connection establishment, {send, receive}*, close

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sock;    /* Socket descriptor */</td>
</tr>
<tr>
<td>struct sockaddr_in echoServAddr; /* Echo server address */</td>
</tr>
<tr>
<td>unsigned short echoServPort; /* Echo server port */</td>
</tr>
<tr>
<td>char <em>servIP; /</em> Server IP address (dotted quad) */</td>
</tr>
<tr>
<td>char <em>echoString; /</em> String to send to echo server */</td>
</tr>
<tr>
<td>unsigned int echoStringLen; /* Length of string to echo */</td>
</tr>
<tr>
<td>int bytesRcvd, totalBytesRcvd; /* Bytes read in single recv() and total bytes read */</td>
</tr>
</tbody>
</table>

Socket descriptor is a file descriptor (in Unix); In Winsock a similar role is played by a socket "handle"
Struct sockaddr_in is the structure containing internet addresses; other address families use different sockaddr structures
servIP is a character string containing an address such as "199.237.73.84"
if (argc < 3) || (argc > 4))
/* Test for correct number of arguments */
{
  fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [Echo Port]\n", argv[0]);
  exit(1);
}
servIP = argv[1]; /* First arg: server IP address (dotted quad) */
echoString = argv[2]; /* Second arg: string to echo */
if (argc == 4)
  echoServPort = atoi(argv[3]); /* Use given port, if any */
else
  echoServPort = 7; /* 7 is the well-known port for the echo service */

Note use of default port if none is provided on the command line. Our machines
don’t have an echo server running so this is not a useful feature.
As a rule I don’t like the practice of writing then and else clauses without
surrounding braces. I find such code to be error-prone.

/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
  DieWithError("socket() failed");
/* Construct the server address structure */
memset(echoServAddr, 0, sizeof(echoServAddr));
/* Zero out structure */
echoServAddr.sin_family = AF_INET;
/* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);
/* Server IP address */
echoServAddr.sin_port = htons(echoServPort);
/* Server port */
A socket descriptor is created with a call to socket()
inet_addr() converts a dotted quad string to the internal representation of a network
address
htons(), pronounced “host to network short”, converts a 2byte internal
representation to network byte order. Values in a struct sockaddr_in are in network
byte order. Similar functions ntohs, htonl, and ntohl all exist. The “L” variants
operate on 4 byte quantities.
/* Establish the connection to the echo server */
if (connect(sock, (struct sockaddr *) &echoServAddr,
    sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");

connect() carries out the initial handshake of the protocol for setting up a TCP connection. Note cast of a (struct sockaddr_in *) value to (struct sockaddr *) and the passing of an explicit length for the structure.

/* Establish the connection */
echoStringLen = strlen(echoString); /* Determine input length */
/* Send the string to the server */
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
    DieWithError("send() sent a different number of bytes than expected");

Socket Sending

Note that an explicit length is passed to send – it does not treat its first argument as a null-terminated string because send is used for sending binary data. The 4th argument, 0 here, is a set of flags. Note also the expectation here that send will return the length of the provided data.
Socket Receiving

/* Receive the same string back from the server */
totalBytesRcvd = 0;
printf("Received: "); /* Setup to print the echoed string */
while (totalBytesRcvd < echoStringLen)
{
    /* Receive up to the buffer size (minus 1 to leave space for
     * a null terminator) bytes from the sender */
    if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <= 0)
        DieWithError("recv() failed or connection closed prematurely");
totalBytesRcvd += bytesRcvd; /* Keep tally of total bytes */
    echoBuffer[bytesRcvd] = '\0'; /* Terminate the string */
    printf(echoBuffer); /* Print the echo buffer */
}

Again, the 4th argument to recv(), here 0, contains flags. We’ll talk more about those
later. Here we are prepared to receive the data piecemeal, hence the loop
surrounding recv().