Before we continue on with socket programming, a little refresher on C…

```
char *s;
*s++;
```
This will result in seg fault! because u declared a pointer, never pointed it at anything, then used the pointer!

```
char buf[4096];
strcpy(buf, "This is a string");
```
foo1 = sizeof(buf); // this gives you the size of the buffer, which is 4096
foo2 = strlen(buf); // this gives you the size of the string, which is 16

Do not count on compliers or the OS to clear memory that you requested, clear them yourself:
```
struct sockaddr_in sin;
memset((char*)&sin, 0, sizeof(struct sockaddr_in));
```

Some routines:
`htons()` - host to network short # returns unsigned short int
`htonl()` - host to network long # returns unsigned long int
`ntohs()` - network to host short # returns unsigned short int
`ntohl()` - network to host long # returns unsigned long int

Aight, we're ready to dive into some network code!

- Socket Creation

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

return: socket descriptor (int) - desribes an active socket that will live on the both ends of a connection
* always check for return value!

`family:`
`PF_INET, PF_INET6, PF_UNIX,` ...(Protocol Family)
Also may be listed as `AF_INET` (Address Family)

`type (socket types):`

Datagram	UDP	`SOCK_DGRAM`
Reliable stream	TCP	`SOCK_STREAM`
Raw	IP	`SOCK_RAW` (Must be administrator/root)

`protocol:`
Typically 0 = default protocol for Internet type packets

Example:

```
#include <sys/types.h>
#include <sys/socket.h>

int s;
if ((s = socket (PF_INET, SOCK_DGRAM, 0)) < 0 ) {
       perror (“Socket creation error!”);
}
```

Now we have a socket! We need to fill in some structures so it goes somewhere…

- Addressing

TCP/IP uses combination of IP address and port

- the IP address identifies the host to contact; the Port identifies the service on the host

For TCP/IP (version 4), each node will have a 32 bit Internet Protocol (IP) address

- The address is specified as four bytes
    - A period (.) is placed between byte values
- Each byte can hold the values of 0 -- 255
- Example IPv4 address: 1.2.3.4 129.74.250.18

The connect, bind, and accept functions require a pointer to a protocol specific socket address structure…

```
#include <netinet/in.h>
struct sockaddr {
       unsigned short      sa_family;          //  address family
       char                sa_data[14];        // 14 bytes of address info: contains a
       destination address, and port number for the socket
};

struct sockaddr_in {
       short  int            sin_family;       // AF_INET
       unsigned short int    sin_port;         // network byte order!
       struct in_addr        sin_addr;         // IP address
       unsigned char         sin_zero[8];      // padding
}; <= recommended!


struct in_addr {
       unsigned long         s_addr;           // 32 bit IP address
};
```

Convert IP address: (only work with IPv4)

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
```
converts from a dots-and-numbers string “cp” into a “struct in_addr” structure pointed by inp

```
char *inet_ntoa(struct in_addr in);
```
converts a network address from a “struct in_addr” to a dots-and-numbers format string

```
in_addr_t inet_addr(const char *cp);
```
It converts numbers and dots string “cp” to binary in network byte order.

Example:
```
struct sockaddr_in test;
char   *some_addr;

inet_aton("10.0.0.1", &test.sin_addr); // store IP in antelope
some_addr = inet_ntoa(test.sin_addr);  // return the IP
printf("%s\n", some_addr);             // prints "10.0.0.1”
test.sin_addr.s_addr = inet_addr("10.0.0.1"); // same as the inet_aton() call
```

Get socket addr IPv4 or IPv6

```
void *get_in_addr(struct sockaddr *sa) {
       if (sa->sa_family == AF_INET) {
              return &(((struct sockaddr_in*)sa)->sin_addr);
       }
       return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
```

Example:
```
#define PORT  89
int s;
struct sockaddr_in sin;
if((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )  {
      printf(“Error creating socket\n”);
      exit(0);
}
memset((char*)&sin, 0, sizeof(sin));
sin.sin_family = AF_INET; // AF_NET and PF_NET are equivalent
sin.sin_port = htons(PORT); // transfer to network byte order
sin.sin_addr.s_addr = inet_addr(“192.168.4.6”) ;
```

Now we have everything we need in order to communicate via UDP (a socket, an IPv4 structure telling us the address/port/family we wana talk to),
What's the socket connected to?

- bind()

```
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *sin, int addrlen);
```

returns: 0 if OK, -1 on error

The bind function tells the kernel to associate the socket address with the socket descriptor
`Sockfd`: socket file descriptor generated by socket() call
`Sockaddr`: it contains your address: i.e., the port number and IP address
`Addrlen`: the length in bytes of the sockaddr.

We don't have to call `bind()` on the client side if we dont care about what the local port is
(e.g., in telnet, you only care the remote host port is 23).
You can simply call `sendto` or `connect`, it will check to see if the port is bounded or not, and will bind automatically it to an unused local port if necessary

- sendto()

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd,  // socket descriptor
           const void *msg,  // pointer to a pre-allocated msg buffer
           int length, // # of bytes you wana send
           unsigned int flags, // flags, usually 0
           const struct sockaddr *dest_add, // address struct - where to packet is headed
          socklen_t dest_len); // length of address struct
```

return the number of bytes actually sent, or -1 on error

Example:

```
int sockfd, obytes, ibytes;
struct sockaddr_in sin;
char myBuffer[4096] = "Send this string to the server";
if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
       perror("socket() error!");
       exit(1);
}
memset( (char *)&sin,  0,  sizeof(struct sockaddr_in) );
sin.sin_family = AF_INET;
sin.sin_port = htons(atoi(argv[2]));
sin.sin_addr = inet_addr(argv[1]);
if((obytes = sendto(sockfd, myBuffer, strlen(myBuffer), 0,
                   (struct sockaddr *)&sin, sizeof(struct sockaddr_in))) < 0 )  {
       perror("Client-sendto() error!");
       exit(1);
}
```

- recvfrom()

```
int recvfrom(int sockfd, // socket descriptor
             void *buffer, // pointer to pre-allocated buffer to msgs
             int length, // size of buffer
             unsigned int flags, // flags, usually 0
             struct sockaddr *address, // address struct
             socklen_t *address_len); // length of address struct
```

returns the number of bytes actually received, or -1 on error

Example:

```
struct sockaddr_in sin;
int addr_len = sizeof(sin);
if((ibytes = recvfrom(sockfd, myBuffer, 4096, 0,
             (struct sockaddr *)&sin,&addr_len)) == -1) {
      perror("recvfrom() error ");
}
printf("Client-Received %d bytes from %s\n\n\n", ibytes, inet_ntoa(sin.sin_addr));
myBuffer[ibytes] = '\0';  //  add string terminator
printf("BUFFER: %s\n", myBuffer);

if (close(sockfd) != 0)
   perror("sockfd close failed!\n");
return 0;
```

- close()

```
close(int sockfd); // free a socket descriptor
int shutdown (int sockfd, int how); // changes the usability of a socket

Sockfd: Socket descriptor
How:
 0: further receives are not allowed
 1: further sends are not allowed
 2: further sends and receives are not allowed (like close())
```

Shutdown returns 0 on success and -1 on error

See more code sample from http://beej.us/guide/bgnet/