

# Introduction to Sockets

Marcus Birkenkrahe

February 24, 2025

## Contents

<b>1</b>	<b>Objectives</b>	<b>LECTURE:CSC410</b>	<b>2</b>
<b>2</b>	<b>Port numbers</b>		<b>2</b>
<b>3</b>	<b>Linux example</b>		<b>4</b>
<b>4</b>	<b>Clients and servers</b>		<b>5</b>
<b>5</b>	<b>Sockets</b>		<b>7</b>
<b>6</b>	<b>Hands-on tutorial: hello world</b>		<b>8</b>
6.1	Simple hello world program . . . . .		8
6.2	Simple hello world program with function . . . . .		9
6.3	Hello world with command-line argument . . . . .		10
<b>7</b>	<b>Files</b>		<b>12</b>
<b>8</b>	<b>What's your IP address</b>		<b>14</b>
<b>9</b>	<b>Interlude: C review</b>		<b>14</b>
<b>10</b>	<b>Listing network adapters from C</b>		<b>15</b>
<b>11</b>	<b>Include system header files</b>		<b>18</b>
<b>12</b>	<b>Obtain network interface addresses</b>		<b>21</b>
<b>13</b>	<b>Iterate through the interface list</b>		<b>22</b>
<b>14</b>	<b>Skipping NULL addresses</b>		<b>22</b>

<b>15 Filter for IPv4 and IPv6 addresses</b>	<b>23</b>
<b>16 Print interface name and address type</b>	<b>23</b>
<b>17 Convert binary address to string</b>	<b>23</b>
<b>18 Move to the next interface</b>	<b>23</b>
<b>19 Free allocated memory.</b>	<b>23</b>
<b>20 Revisit the whole program</b>	<b>23</b>
<code>#+src R :file :session <b>R</b> :results graphics output file :exports both:</code>	

## 1 Objectives

LECTURE:CSC410

The following topics are covered:

- ☐ What are port numbers?
- ☐ How to find port numbers on Linux?
- ☐ What is the client-server model?
- ☐ What's a socket?
- ☐ What's your IP address?
- ☐ How can you list network adapters using C?

## 2 Port numbers

- How does a packet know, which application is responsible for it once it arrives at a specific system?
- To direct a packet to the right application, **port** numbers are used. If IP addresses are phone numbers, then port numbers are like phone extensions.
- Applications on your system with specific ports:
  - multiple web browsers
  - an email client
  - a video-conferencing client

- When a packet arrives, the OS looks at the destination port number, and it is used to determine which application could handle it.
- Port numbers are stored as unsigned 16-bit integers - what's their range therefore?

$2^{16}$

- Some port numbers for common protocols:

Port Number		Protocol
20, 21	TCP	<b>File Transfer Protocol (FTP)</b>
22	TCP	<b>Secure Shell (SSH)</b>
23	TCP	<b>Telnet</b>
25	TCP	<b>Simple Mail Transfer Protocol (SMTP)</b>
53	UDP	<b>Domain Name System (DNS)</b>
80	TCP	<b>Hypertext Transfer Protocol (HTTP)</b>
110	TCP	<b>Post Office Protocol, Version 3 (POP3)</b>
143	TCP	<b>Internet Message Access Protocol (IMAP)</b>
194	TCP	<b>Internet Relay Chat (IRC)</b>
443	TCP	<b>HTTP over TLS/SSL (HTTPS)</b>
993	TCP	<b>IMAP over TLS/SSL (IMAPS)</b>
995	TCP	<b>POP3 over TLS/SSL (POP3S)</b>

- The listed port numbers are assigned by the IANA (Internet Assigned Numbers Authority).

### 3 Linux example

- To see some of these in Linux:

```
netstat -tulnp
```

Active Internet connections (only servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID
tcp	0	0	127.0.0.1:631	0.0.0.0:*	LISTEN	-
tcp	0	0	127.0.0.53:53	0.0.0.0:*	LISTEN	-
tcp6	0	0	:::1:631	:::*	LISTEN	-
tcp6	0	0	:::1716	:::*	LISTEN	22
tcp6	0	0	:::60000	:::*	LISTEN	-
udp	0	0	224.0.0.251:5353	0.0.0.0:*		17
udp	0	0	224.0.0.251:5353	0.0.0.0:*		17
udp	0	0	0.0.0.0:5353	0.0.0.0:*		-
udp	0	0	0.0.0.0:39386	0.0.0.0:*		-
udp	0	0	127.0.0.53:53	0.0.0.0:*		-
udp	0	0	127.0.0.1:323	0.0.0.0:*		-
udp6	0	0	:::5353	:::*		-
udp6	0	0	:::1:323	:::*		-
udp6	0	0	fe80::3411:8f1:989f:546	:::*		-
udp6	0	0	:::1716	:::*		22
udp6	0	0	:::35705	:::*		-

On my machine at home (PIDs not listed):

Active Internet connections (only servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.53:53	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:631	0.0.0.0:*	LISTEN
tcp6	0	0	:::60000	:::*	LISTEN
tcp6	0	0	:::1:631	:::*	LISTEN
udp	0	0	127.0.0.53:53	0.0.0.0:*	
udp	0	0	127.0.0.1:323	0.0.0.0:*	
udp	0	0	224.0.0.251:5353	0.0.0.0:*	
udp	0	0	224.0.0.251:5353	0.0.0.0:*	
udp	0	0	224.0.0.251:5353	0.0.0.0:*	
udp	0	0	224.0.0.251:5353	0.0.0.0:*	
udp	0	0	0.0.0.0:5353	0.0.0.0:*	

```

udp      0      0 0.0.0.0:46510      0.0.0.0:*
udp6    0      0 :::1:323           :::*
udp6    0      0 fe80::3411:8f1:989f:546 :::*
udp6    0      0 :::5353           :::*
udp6    0      0 :::47923          :::*

```

- Web server example (do this on the shell):
  1. start a web server with `python3` on port 8080
  2. `grep` for 8080 in the `netstat -tulnp` output

```

python3 -m http.server 8080 &
netstat -tulnp | grep 8080

```

- Output:

```

tcp  0  0 0.0.0.0:8080 0.0.0.0:* LISTEN  274074/python3

```

## 4 Clients and servers

- When you use a telephone, a call must be initiated by one party. The initiating party dials the number for the receiving party, and the receiving party answers.
- This describes the **client-server** model paradigm in networking:
  1. A server listens for connections.
  2. The client knows the address and port number that the server is listening to and establishes connection by sending the first packet.
- Example: Web server
  1. The web server at `example.com` listens on port 80 (HTTP) and on port 443 (HTTPS) for connections.
  2. A web browser (client) establishes the connection by sending the first packet to `http://example.com:80`.



## 5 Sockets

- A **socket** is one end-point of a communication link. It enables sending and receiving data over the network.
- An open socket is defined by a quintuple (5-tuple):
  1. Local IP address
  2. Local port
  3. Remote IP address
  4. Remote port
  5. Protocol (UDP or TCP)
- With this information, the OS knows which application is responsible for any packets received.
- Example: You have two web browsers (Chrome and Firefox) open simultaneously, both connecting to **example.com** on port 80. How can the OS keep the connections separate?

The OS keeps the connections separate by looking at the socket data. In this case, local and remote IP addresses, remote port and protocol are identical but the local port is different. The local (aka **ephemeral**) port was chosen to be different by the OS for the connection.

- The router NAT (Network Address Translation) also stores the socket 5-tuple to know how to route received packets back into the private network.
- Take a look at the **socket** man page on Linux. What does it do?

The function **socket()** creates an endpoint for communication and returns a **file descriptor** that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.
- What is a "file descriptor"?

For the OS (any OS), everything is a "file". The file descriptor is a small non-negative integer that uniquely identifies

a socket within a process. When calling `socket`, you get a descriptor that you can use to perform operations on that socket (binding, listening, accepting, reading, and writing).

- How does a `socket` call look like?

```
// socket: establish network connection to listen on
// returns: file descriptor (int)
// params: domain (protocol family), type (data stream),
//         protocol (TCP or UDP or another)
int socket(int domain, int type, int protocol);
```

For example:

- `domain = AF_STREAM` (Address Family - Internet: using IPv4 address)
- `type = SOCK_STREAM` (Socket Type - TCP)
- `protocol = 0` (automatically select the protocol - TCP)

- Where is the 5-tuple?

The 5-tuple is completed when a client connects to a server, and a server accepts a client (functions `connect` and `accept`).

- Before looking at socket programming, let's look at a similar abstraction, `file` handling.

## 6 Hands-on tutorial: hello world

### 6.1 Simple hello world program

- Problem: Create a "hello world" program in C.
- Open Emacs to `hello.c` (C-x C-f hello.c <RET>)
- Solution:

```
/******
// hello.c: print "Hello, World!" to stdout. Input: NONE. Output:
// Hello, World! (string). Author: Marcus Birkenkrahe. Date:
// 02/24/2025
/******
```



```
// include libraries
#include <stdio.h>

// main: main program
// returns: integer (int)
// params: none (void)
int main(void)
{
    puts("Hello, world!");
    return 0;
}
```

- Compile and run the file on the command-line. The object file (executable) should be called `hello`:

```
gcc ../src/hello.c -o hello
./hello
```

## 6.2 Simple hello world program with function

- Problem: Modify the "hello world" program with a function `hello` that prints the string "Hello, world!". Include a function prototype. Tangle the code to `hello2.c`
- Solution:

```

/*****
// hello2.c: print "Hello, World!" to stdout use a void function
//          hello() to print Input: NONE Output: Hello, World!
//          (string) Author: Marcus Birkenkrahe Date: 02/24/2025
*****/

// include libraries
#include <stdio.h>

// prototype functions

// hello: print hello world
// returns: nothing (void)
// params: none (void)
void hello(void);
```

```

// main: main program
// returns: integer (int)
// params: none (void)
int main(void)
{
    hello(); // function call
    return 0;
}

// function definition
void hello(void)
{
    puts("Hello, world!");
}

```

- Compile and run the file on the command-line. The object file (executable) should be called `hello2`:

```

gcc ../src/hello2.c -o hello2
./hello2

```

### 6.3 Hello world with command-line argument

- Modify the "Hello, World!" program to accept a name as a command-line argument and print:
  1. "Hello, [Name]!" if a name is provided.
  2. "Hello, World!" if no name is provided.
- To check if an argument is provided, check if `argc` (the number of command-line arguments) is greater than 1.
- The command-line argument itself (if one was given), is stored in the array `argv[]`. Its first element, `argv[0]` is the name of the program itself (e.g. `hello3`), the next elements are the command-line arguments.
- Starter code:

```

/*****/
// hello3.c: command-line version of "hello world" with input. Input:

```

```
// [name] (string) or none. Output: "Hello, [name]!" or "Hello,
// world!". Author: Marcus Birkenkrahe Date: 02/24/2025
/*****/
#include <stdio.h>

// main: main program
// returns: integer (int)
// params: argument count (argc), argument vector (argv)
int main(int argc, char *argv[])
{
    // TODO: Check if an argument is provided

    // TODO: Print "Hello, [Name]!" if a name is given

    // TODO: Otherwise, print "Hello, World!"

    return 0;
}
```

- Test the code on the command-line:

```
gcc hello3.c -o hello3
./hello3 Bob!
./hello3
```

- Expected output:

```
Hello, Bob!
Hello, world!
```

- Solution:

```
/*****/
// hello3.c: command-line version of "hello world" with input. Input:
// [name] (string) or none. Output: "Hello, [name]!" or "Hello,
// world!". Author: Marcus Birkenkrahe Date: 02/24/2025
/*****/
#include <stdio.h>

// main: main program
```

```

// returns: integer (int)
// params: argument count (argc), argument vector (argv)
int main(int argc, char *argv[])
{
    // TODO: Check if an argument is provided

    // TODO: Print "Hello, [Name]!" if a name is given

    // TODO: Otherwise, print "Hello, World!"

    return 0;
}

```

## 7 Files

- A similar abstraction is a **file**, which enables applications to read and write data using a file handle.
- In C for example, the `FILE *` type in the standard library is a file handle used with functions like `fopen`, `fread`, `fwrite`, and `fclose`.
- A lower-level system call `open` returns a file descriptor (an integer handle), which can be used with `read`, `write`, and `close`.
- Code along!
- Example 1: Using `FILE *` (C Standard Library)

1. `fopen` opens `file` (the 'handle') in write mode.
2. `file` is a `FILE` pointer to the beginning of the file.
3. `fprintf` writes text to the file.
4. `fclose` close the handle.

```

#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w"); // Open file for writing

    if (file == NULL) {
        perror("Error opening file"); // print error to stderr
    }
}

```

```

        return 1;                                // if no file found
    }

    fprintf(file, "Hello, File!\n"); // Write to the file
    fclose(file); // Close the file handle

    return 0;
}

```

- Testing example 1:

```

gcc fileh.c -o fileh
./fileh
ls -lt example.txt fileh.c fileh
cat example.txt

```

- Example 2: Using a file descriptor (**open**, **write**, **close**)

1. **open** opens the file with write-only access, creating it if necessary.
2. **write** writes raw bytes (char array/string) to the file.
3. **close** closes the file descriptor.

```

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd = open("example2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }

    const char *text = "Hello, File Descriptor!\n";
    write(fd, text, 25); // Write raw bytes to the file

    close(fd); // Close the file descriptor

    return 0;
}

```

- Testing example 2:

```
gcc fileh2.c -o fileh2
./fileh2
ls -lt example2.txt fileh2.c fileh2
cat example2.txt
```

## 8 What's your IP address

- You need to be able to find your IP address.
- On Windows: `ipconfig` - look for the IPv4 Address
- On Unix-based systems (MacOS or Linux): `ip addr` (or older: `ipconfig`)
- If you're behind a NAT, your public IP address may be concealed. You need to contact an internet server to find out, and connect to them in a web browser:

- `api.ipify.org`
  - `helloacm.com/api/what-is-my-ip-address`
  - `icanhazip.com`
  - `ifconfig.me/ip`

- Try one of these now with the Emacs `eww` browser (`M-x eww`).
- If someone knows your public IP address, they can scan your network for open ports, vulnerability, or DDoS attack.<sup>1</sup> Use of a VPN hides your real IP address, a firewall blocks unauthorized traffic, and dynamic IP addressing will change it (often) to avoid this.
- Getting your local IP address is what we'll try to do directly using C

## 9 Interlude: C review

In-class:

- Download and complete: `tinyurl.com/fizzbuzz-practice`

Home assignments (review in class):

---

<sup>1</sup>Distributed Denial of Service, by sending massive amounts of traffic to an address.

- FizzBuzz (loops, if else)
- FizzBuzz reloaded (functions, pointers, arrays)
- Convert MAC addresses into binary (lookup tables)

## 10 Listing network adapters from C

- It is useful for your network programs what your local address is. To find this out, we need to use an Application Programming Interface (API).
- The API for listing local addresses is very different between systems: The one for Windows is way more complicated than for Unix-based systems (MacOS and Linux).
- The design and practice of socket APIs is not wholly rationally designed: A lot of it has historical reasons. Cp. the changes documented in the `socket` man page for Linux (from 2.6).
- We will only cover the Unix case. To start, download the code, and then insert it (`C-x i`) here from the file `unix_list.c`:

```
wget -O unix_list.c tinyurl.com/unix-list-c
```

- `unix_list.c` -

```
/*
 * MIT License
 *
 * Copyright (c) 2018 Lewis Van Winkle
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
```

```

*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*/

#include <sys/socket.h>
#include <netdb.h>
#include <ifaddrs.h>
#include <stdio.h>
#include <stdlib.h>

int main() {

    struct ifaddrs *addresses;

    if (getifaddrs(&addresses) == -1) {
        printf("getifaddrs call failed\n");
        return -1;
    }

    struct ifaddrs *address = addresses;
    while(address) {
        if (address->ifa_addr == NULL) {
            address = address->ifa_next;
            continue;
        }
        int family = address->ifa_addr->sa_family;
        if (family == AF_INET || family == AF_INET6) {

            printf("%s\t", address->ifa_name);
            printf("%s\t", family == AF_INET ? "IPv4" : "IPv6");

            char ap[100];
            const int family_size = family == AF_INET ?

```



```

        sizeof(struct sockaddr_in) : sizeof(struct sockaddr_in6);
        getnameinfo(address->ifa_addr,
                    family_size, ap, sizeof(ap), 0, 0, NI_NUMERICHOST);
        printf("\t%s\n", ap);
    }
    address = address->ifa_next;
}

freeifaddrs(addresses);
return 0;
}

```

- Output on my machine at home:

```

: lo   IPv4           127.0.0.1
: enp4s0   IPv4       192.168.1.250
: wlo1     IPv4       192.168.68.56
: lo   IPv6           ::1
: enp4s0   IPv6       2600:1702:4ba0:4b0::42
: enp4s0   IPv6       2600:1702:4ba0:4b0:2eca:99eb:1063:393
: enp4s0   IPv6       2600:1702:4ba0:4b0:952e:b307:998b:9078
: enp4s0   IPv6       fe80::3411:8f1:989f:c525%enp4s0
: wlo1     IPv6       fe80::7c2f:481f:e10a:abe8%wlo1

```

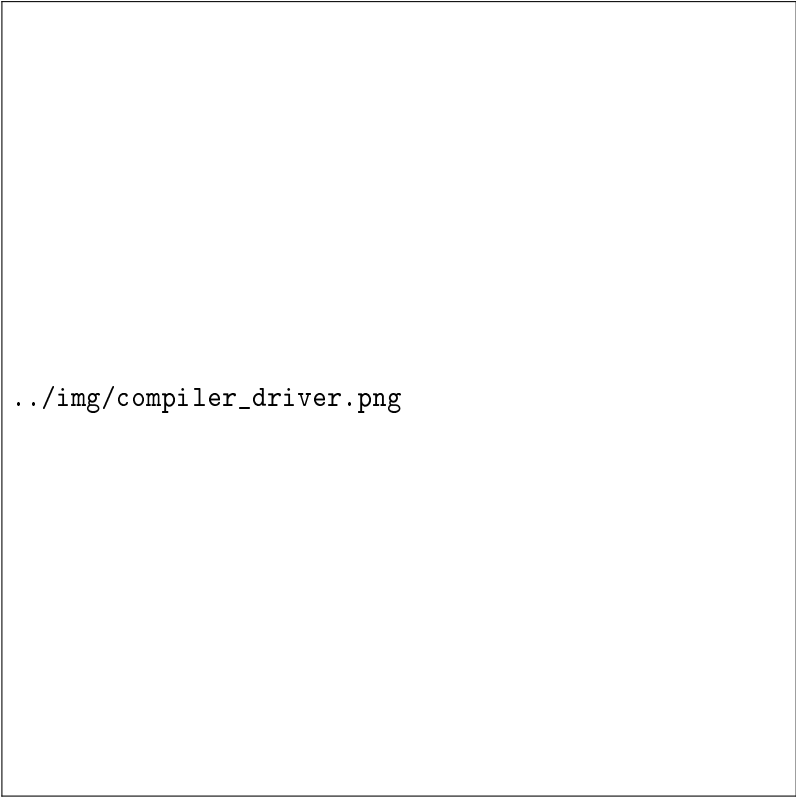
- Let's try to understand this file step-by-step.

1. Include system header files
2. Obtain network interface addresses
3. Iterate through the interface list
4. Skip NULL addresses
5. Filter for IPv4 and IPv6 addresses
6. Print interface name and address type
7. Convert binary address to string
8. Move to the next interface
9. Free allocated memory.

## 11 Include system header files

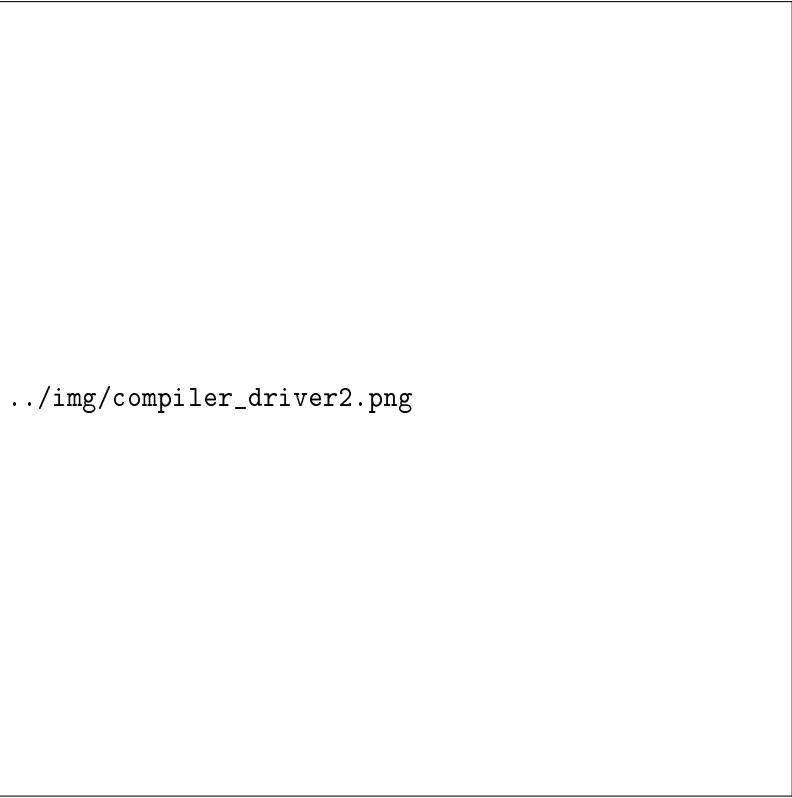
- System header files (`.h`) contain functions and constants that can be used in C programs. They are programmed to be both fast and secure.
- Why are they included as `<...>`? And what does the `#include` mean?
  - Header files are included as `<...>` when their location is known to the OS (they are in standard locations).
  - The `#include` command means that the files are included by the preprocessor before program compilation, and added to the machine code during the linking phase.
- This is how that looks like for the "hello world" program:

```
#include <stdio.h>
int main(void) {
    printf("Hello, world!");
    return 0;
}
```



```
../img/compiler_driver.png
```

- And with all the details and intermediate files:



../img/compiler\_driver2.png

- Let's look at the header files one by one:

```
#include <sys/socket.h>
#include <netdb.h>
#include <ifaddrs.h>
#include <stdio.h>
#include <stdlib.h>
```

Header File	Purpose
<sys/socket.h>	Socket definition functions ( <code>socket()</code> , <code>bind()</code> , <code>listen()</code> , etc.).
<netdb.h>	Network database operations ( <code>getaddrinfo()</code> , <code>gethostbyname()</code> ).
<ifaddrs.h>	Retrieves network interface addresses ( <code>getifaddrs()</code> ).
<stdio.h>	Standard I/O functions ( <code>printf()</code> , <code>scanf()</code> , <code>fopen()</code> ).
<stdlib.h>	Memory, conversions, process control ( <code>malloc()</code> , <code>atoi()</code> , <code>exit()</code> ).

## 12 Obtain network interface addresses

- Code:

```
struct ifaddrs *addresses;
if (getifaddrs(&addresses) == -1) {
    printf("getifaddrs call failed\n");
    return -1;
}
```

- Summary:

We declare a variable, **addresses**, which stores the addresses. A call to **getifaddrs** allocates memory and fills in a linked list of addresses. This function returns 0 on success or -1 on failure.

- Details:

1. **ifaddrs** is a linked list (a chain of nodes which each node containing a pointer to the next node, ending with a null pointer).

It is typically defined like this:

```
struct ifaddrs {
    struct ifaddrs *ifa_next;    /* Pointer to next interface in list */
    char            *ifa_name;    /* Interface name (e.g., "eth0") */
    unsigned int     ifa_flags;    /* Interface flags (IFF_UP, IFF_LOOPBACK, etc) */
    struct sockaddr *ifa_addr;    /* Address of the interface */
    struct sockaddr *ifa_netmask; /* Netmask of the interface */
    struct sockaddr *ifa_broadaddr; /* Broadcast or P2P destination address */
    struct sockaddr *ifa_dstaddr; /* Point-to-point destination address */
    void            *ifa_data;    /* Interface-specific data */
};
```

2. **getifaddr(&addresses)** retrieves the list of network interfaces and stores them in the linked list **addresses**.
3. If the call fails (e.g. because there is no network connection) then the message is printed, the error code -1 is returned, and the program terminates. Using **EXIT\_FAILURE** from **<stdlib.h>** is more portable and more readable.

## 13 Iterate through the interface list

- Summary:

We use a new pointer, `address`, to walk through the linked list of `addresses` using a `while` loop - it will end when the list is finished, and `address` becomes the null pointer.

- Code:

```
struct ifaddrs *address = addresses; // point to first address
while (address) {
```

- Details:

The long form of this condition test is `while (address != NULL)`.

## 14 Skipping NULL addresses

- Summary:

Some interfaces might not have assigned addresses so they are skipped.

- Code:

```
if (address->ifa_addr == NULL) {
    address = address->ifa_next;
    continue;
}
```

- Details:

1. `address` is an `ifaddrs` structure. The `->` operator is used to access elements in the list without having to worry about pointer arithmetic. `address->ifa_addr` is short for `(*address).ifa_addr`.
2. The field `ifa_addr` is a pointer to a `struct sockaddr`, which represents the network access of the interface.
3. The `sockaddr` structure represents a generic socket address:

```
struct sockaddr {
    sa_family_t sa_family; // e.g. AF_INET (IPv4), AF_PACKET (MAC address)
    char        sa_data[]; // Address data (depends on sa_family)
}
```

- 15 Filter for IPv4 and IPv6 addresses
- 16 Print interface name and address type
- 17 Convert binary address to string
- 18 Move to the next interface
- 19 Free allocated memory.
- 20 Revisit the whole program

- Code:

```
<<include system header files>>  
<<obtain network interface addresses>>  
<<iterate through the interface list>>  
    <<skipping NULL addresses>>
```