CSE 333 Section 9 - Servers, netcat and boost

Welcome back to section! We're glad that you're here :)

Step-by-step Server-Side Networking

Step 1. Figure out what IP address and port to use for the server. (`getaddrinfo()`)

```
// returns 0 on success, negative number on failure
int getaddrinfo(const char *hostname,    // hostname to lookup
          const char *servname,          // service name
          const struct addrinfo *hints, // desired output
      (optional)
          struct addrinfo **res);        // results structure

struct addrinfo {
      int ai_flags;               // additional flags
      int ai_family;              // AF_INET, AF_INET6, AF_UNSPEC
      int ai_socktype;                 // SOCK_STREAM, SOCK_DGRAM, 0
      int ai_protocol;                 // IPPROTO_TCP, IPPROTO_UDP, 0
      size_t ai_addrlen;          // length of socket addr in bytes
      struct sockaddr* ai_addr;   // pointer to socket addr
      char* ai_canonname;         // canonical name
      struct addrinfo* ai_next;   // can have linked list of records
}
```

Step 2. Create a socket. (`socket()`)

```
// returns file descriptor on success, -1 on failure (errno set)
int socket(int domain,          // AF_INET, AF_INET6, etc.
          int type,             // SOCK_STREAM, SOCK_DGRAM, etc.
          int protocol);        // usually 0
```

Step 3. Bind the socket to an IP address for the server. (`bind()`)

```
// returns 0 on success, -1 on failure (errno set)
int bind(int sockfd,                     // fd from step 2
        struct sockaddr *serv_addr,     // socket addr from step 1
        socklen_t addrlen);             // size of serv_addr
```

Step 4. Mark the socket as "passive", one that listens for incoming connections. (`listen()`)

```
// returns 0 on success, -1 on failure (errno set)
int listen(int sockfd,   // fd from step 2
          int backlog); // max amount of pending connections.
                        // Usually just use SOMAXCONN
```

Step 5. .Accepts an incoming connection & provides information on the connection (`accept()`)

```
// returns a file descriptor that handles the connection to a
// specific client connection, -1 on failure (errno set)
int accept(int listen_fd,               // fd from step 2
        struct sockaddr *client_addr, // output param for client
info
        socklen_t addrlen);                    // size of client_addr
```

Step 6. Transfer data through the accepted socket. (`read()` and `write()`)

```
// returns amount read, 0 for EOF, -1 on failure (errno set)
ssize_t read(int fd, void *buf, size_t count);

// returns amount written, -1 on failure (errno set)
ssize_t write(int fd, void *buf, size_t count);
```
These are the same POSIX calls used for files, so remember to deal with partial reads/writes!

Step 7. Close the listen and accept sockets when done. (`close()`)

```
// returns 0 for success, -1 on failure (errno set)
int close(int fd);
```

Exercise 1 (Diagram on the next page)
Fitting the Pieces Together. The diagram on the next page depicts the basic skeleton of a C++ program for server-side networking, with arrows representing the flow of data between them. Fill in the names of the functions being called, and the arguments being passed. Then, for each arrow in the diagram, fill in the type and/or data that it represents.

```
                        ┌─────────────────────────┐
                        │  specify lookup hints   │
                        └─────────────────────────┘
┌ ─ ─ ─ ─ ─ ─ ─ ─ ┐                  │
   struct addrinfo*  ┈ ┈ ┈ ┈ ┈ ┈     │
└ ─ ─ ─ ─ ─ ─ ─ ─ ┘                  │
                                     │
      ╭───╮  ┌──────────────────────────────────────────┐
      │ 1 │  │ getaddrinfo(hostname, servname, hints , res) │
      ╰───╯  └──────────────────────────────────────────┘
                             │              ┈ ┈ ┈ ┈ ┈   ┌ ─ ─ ─ ─ ─ ─ ─ ─ ┐
                             │                            struct addrinfo*
                             │                           └ ─ ─ ─ ─ ─ ─ ─ ─ ┘
              ┌──────────────────────────────┐
              │  extract fields from result  │
              │       (IPv4 vs IPv6)         │
              └──────────────────────────────┘
┌ ─ ─ ─ ─ ─ ─ ─ ┐      │            │
  int (sa_family) ┈ ┈ ┈ ┈           │        ┌ ─ ─ ─ ─ ─ ─ ─ ─ ┐
└ ─ ─ ─ ─ ─ ─ ─ ┘      │            │            struct
                       │            │         sockaddr_storage*
      ╭───╮  ┌────────────────────────────────┐  └ ─ ─ ─ ─ ─ ─ ─ ─ ┘
      │ 2 │  │  socket(domain, type, protocol)│
      ╰───╯  └────────────────────────────────┘
                   │                │
┌ ─ ─ ─ ─ ─ ─ ┐    │                │
  _____   ┈ ┈ ┈                │
└ ─ ─ ─ ─ ─ ─ ┘    │                │
      ╭───╮  ┌────────────────────────────────┐
      │ 3 │  │ _____ ( _____ ,_____ , addrlen) │
      ╰───╯  └────────────────────────────────┘
                        │
      ╭───╮  ┌────────────────────────────────┐
      │ 4 │  │ _____ (_____, backLog)  │
      ╰───╯  └────────────────────────────────┘

      ╭───╮  ┌────────────────────────────────────┐
      │ 5 │  │ _____(_____, client_addr, addrlen) │
      ╰───╯  └────────────────────────────────────┘
                   │        ┈ ┈ ┈  ┌ ─ ─ ─ ─ ─ ─ ┐
                   │                 _____
                   │               └ ─ ─ ─ ─ ─ ─ ┘
      ╭───╮  ┌────────────────────────────────┐
      │ 6 │  │  read/write(_____, buf, count) │
      ╰───╯  └────────────────────────────────┘

      ╭───╮  ┌────────────────────────────────┐
      │ 7 │  │         close(_____)           │
      ╰───╯  └────────────────────────────────┘
```
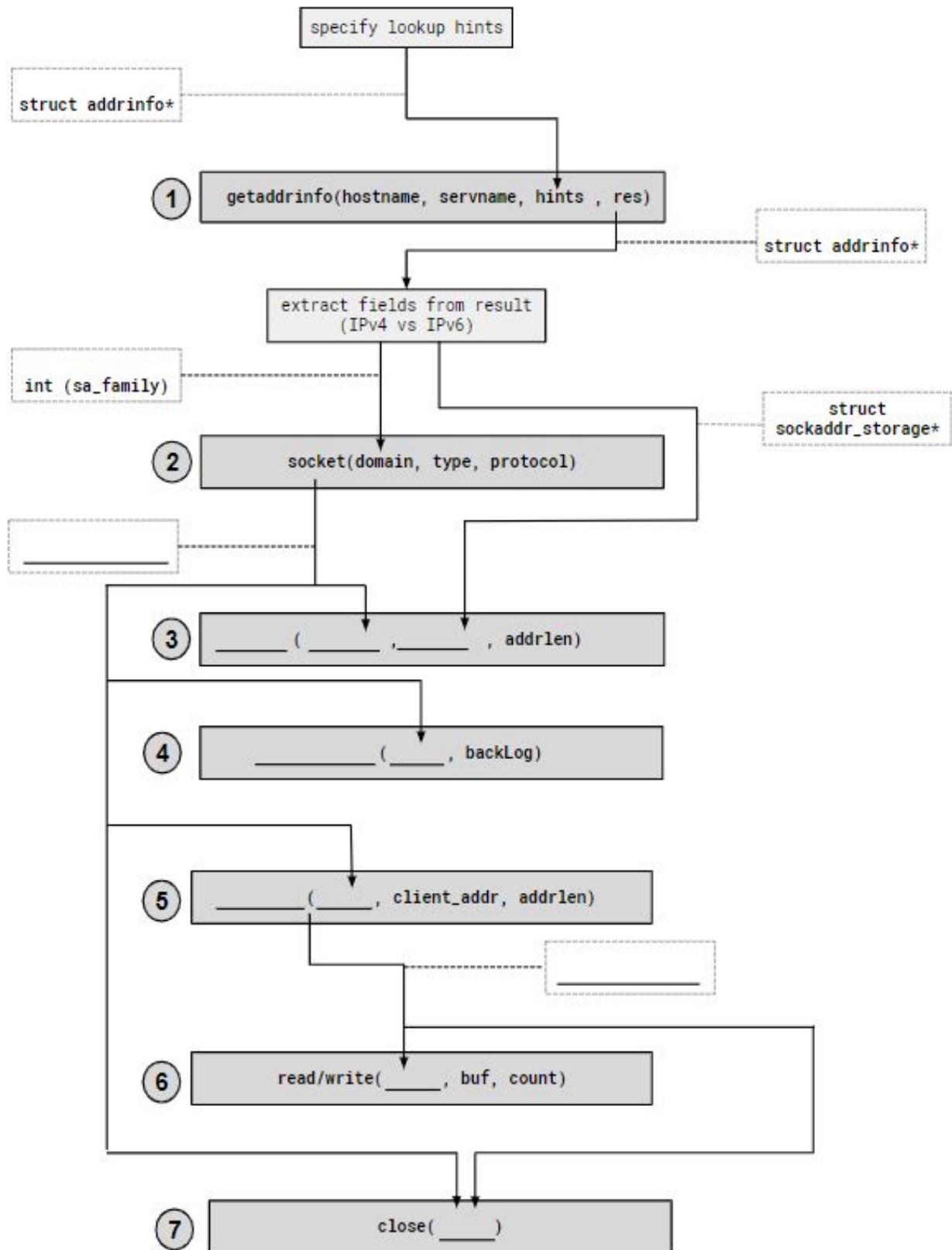
Boost Library

- Very useful for dealing with strings (HW 4!), such as trimming, pattern matching, splitting, replacing, etc.
- `#include <boost/algorithm/string.hpp>`.
- Boost is template madness, so we have handy reference information here :)

```
using namespace boost;

// Trim the whitespace off the left and right of to_modify
// to_modify is input and output parameter
void boost::trim(string& to_modify);

// Replace all instances of to_find with to_replace.
// to_modify is input and output parameter
void boost::replace_all(string& to_modify,
                        const string& to_find,
                        const string& to_replace);

// Returns a predicate that matches on any of the characters in tokens
boost::PredicateT boost::is_any_of(const string& tokens);

// Split the string by the characters in match_on
void boost::split(vector<string>& output,
                  const string& input,
                  boost::PredicateT match_on,
                  boost::token_compress_mode_type compress);
```

Examples

```
TRIM:    string s("  \t  HI  \n  ");
         boost::algorithm::trim(s);  // s == "HI"

REPLACE_ALL:
         string s1("ynrnrt");
         boost::algorithm::replace_all(s1, "nr", "e");  // s1 ==
         "yeet"

         string s2("queue?");
         boost::algorithm::replace_all(s2, "que", "q"); // s2 ==
     "que?"

SPLIT:   string str1("hello abc-*-ABC-*-aBc goodbye");
         vector<string> SplitVec;  // #2: Search for tokens
```

```
split( SplitVec, str1, is_any_of("-*"), token_compress_on );
    // SplitVec == { "hello abc", "ABC", "aBc goodbye" }
```

Exercise 2

Write a function that takes in a string that contains words separated by whitespace and returns a vector that contains all of the words in that string, in the same order as they show up, but with no duplicates. Ignore all leading and trailing whitespace in the input string.

Example:
```
RemoveDuplicates(" Hi I'm sorry  jon sorry hi hihi  hi hi    ")
```
should return the vector `["Hi", "I'm", "sorry", "jon", "hi", "hihi"]`

```
vector<string> RemoveDuplicates(const string& input){
```

```
}
```