CSE 333 Section 9

Server-Side Programming & HW4 Tools

Logistics

- Wednesday, November 27th
  - Exercise 17 due 11 AM
- Thursday, December 5th
  - HW 4 due 8:59 PM
  - Can only use one late day!!

Client Side vs Server Side

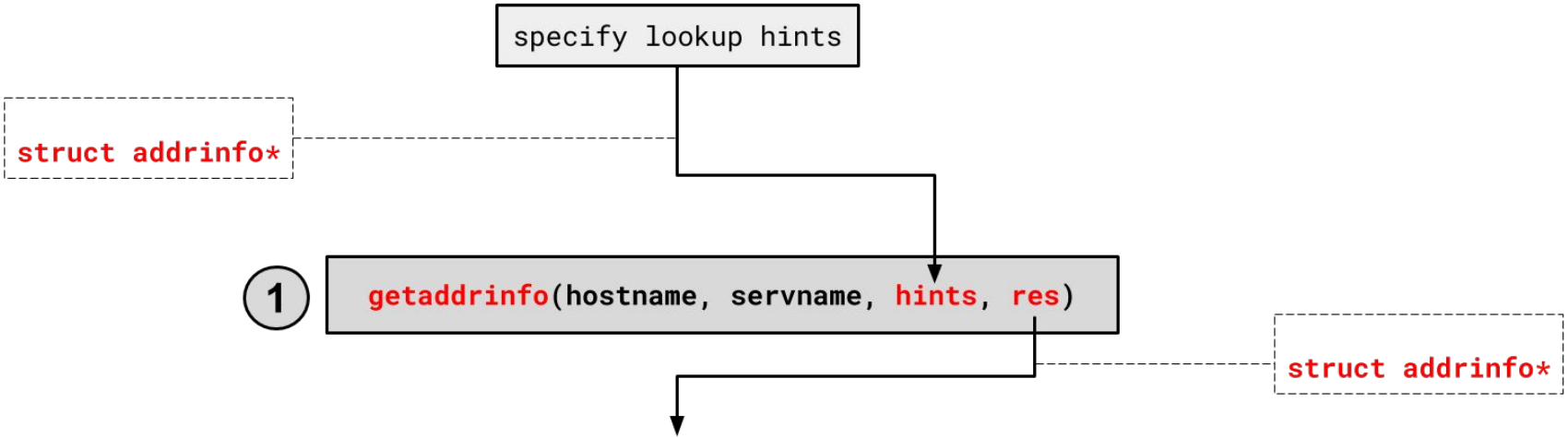
What Differentiates a Client and a Server?

- Clients:
  - Connects to another host in the server, then handles data transfer
- Servers:
  - “Sets up shop” and waits for incoming requests, transfers data with each client connection it accepts.

1. getaddrinfo()

```
int getaddrinfo(const char *hostname,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

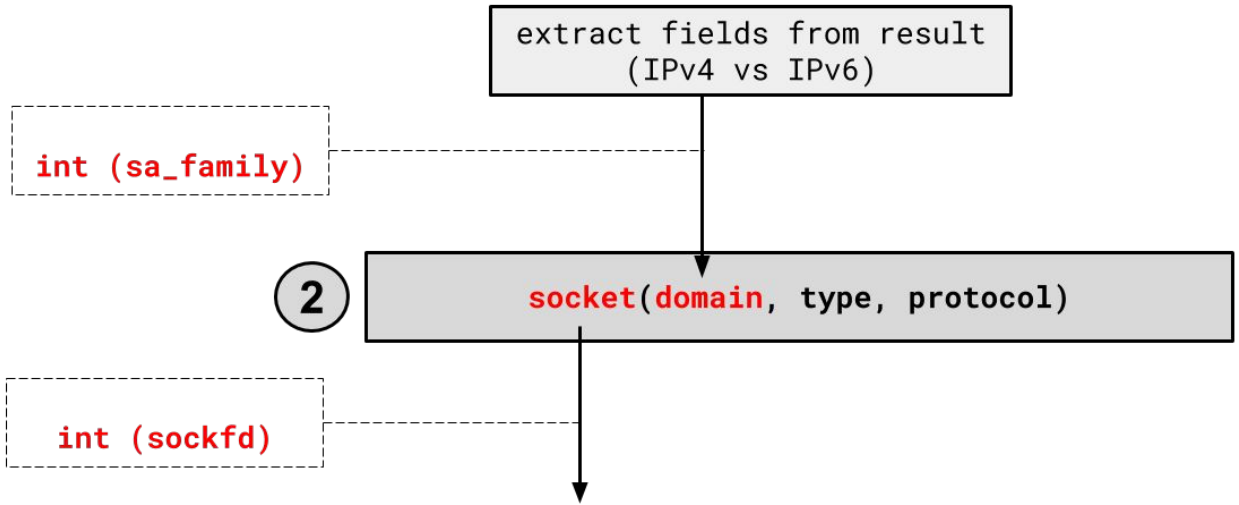
- Finds an address for the server application to use
  - Does this when `hints->ai_flags = AI_PASSIVE` and `hostname = nullptr`
- Use “hints” to specify constraints (`struct addrinfo *`)
- Get back a linked list of `struct addrinfo` results



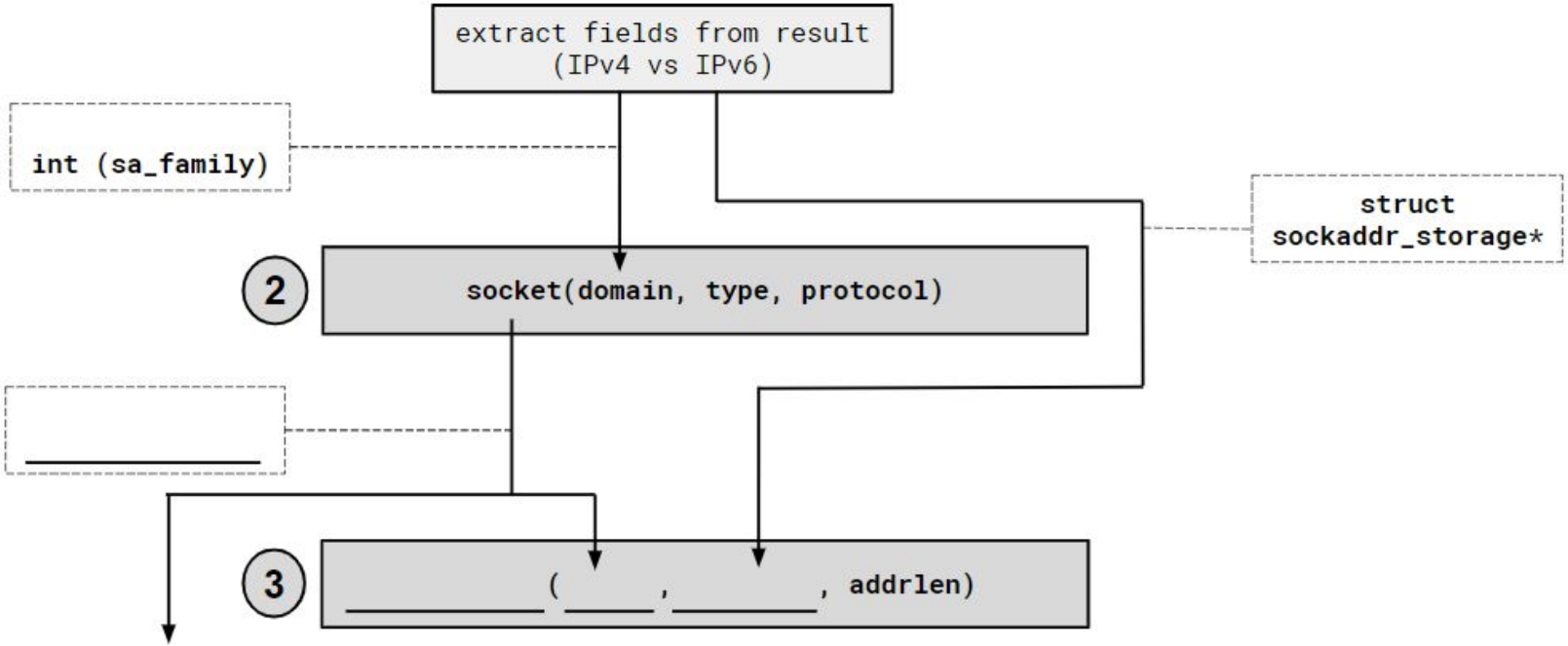
2. socket()

```
int socket(int domain,     // AF_INET, AF_INET6
           int type,       // SOCK_STREAM (TCP)
           int protocol); // 0
```

- Creates a “raw” socket, ready to be bound
- Returns file descriptor (`sockfd`) on success, `-1` on failure



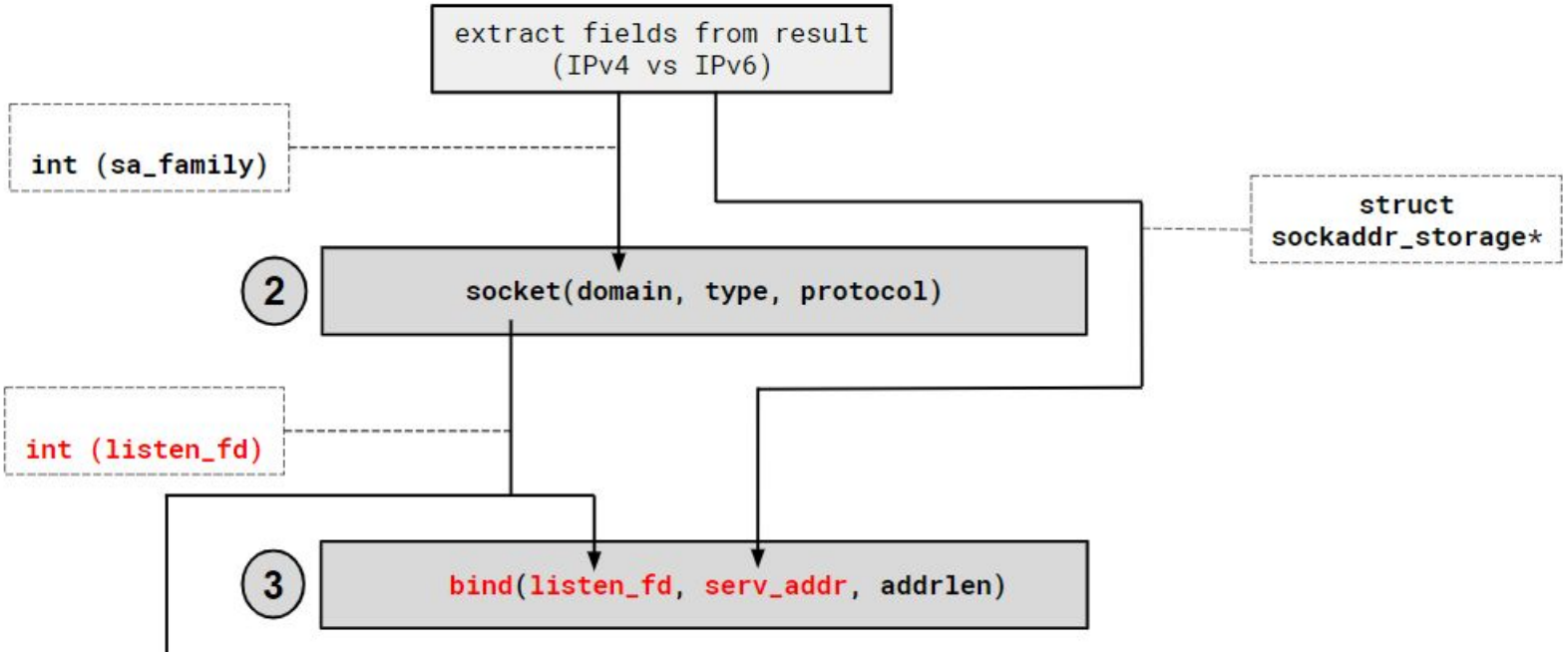
3.


extract fields from result
(IPv4 vs IPv6)
int (sa_family)
struct
sockaddr_storage*
2
socket(domain, type, protocol)
3
addrlen)

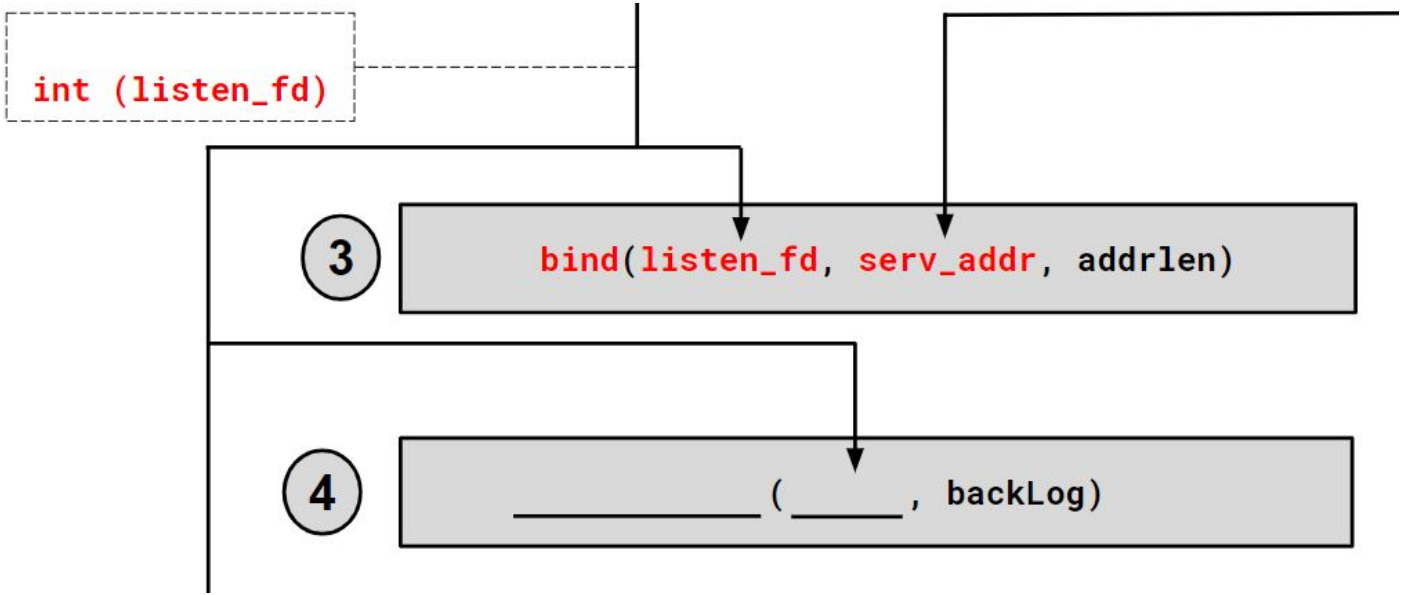
3. bind()

```
int bind (int sockfd,                          // from 2
          const struct sockaddr *serv_addr, // from 1
          socklen_t addrlen);     // size of serv_addr
```

- Binds an available socket to a specified address
- Returns 0 on success, -1 on failure



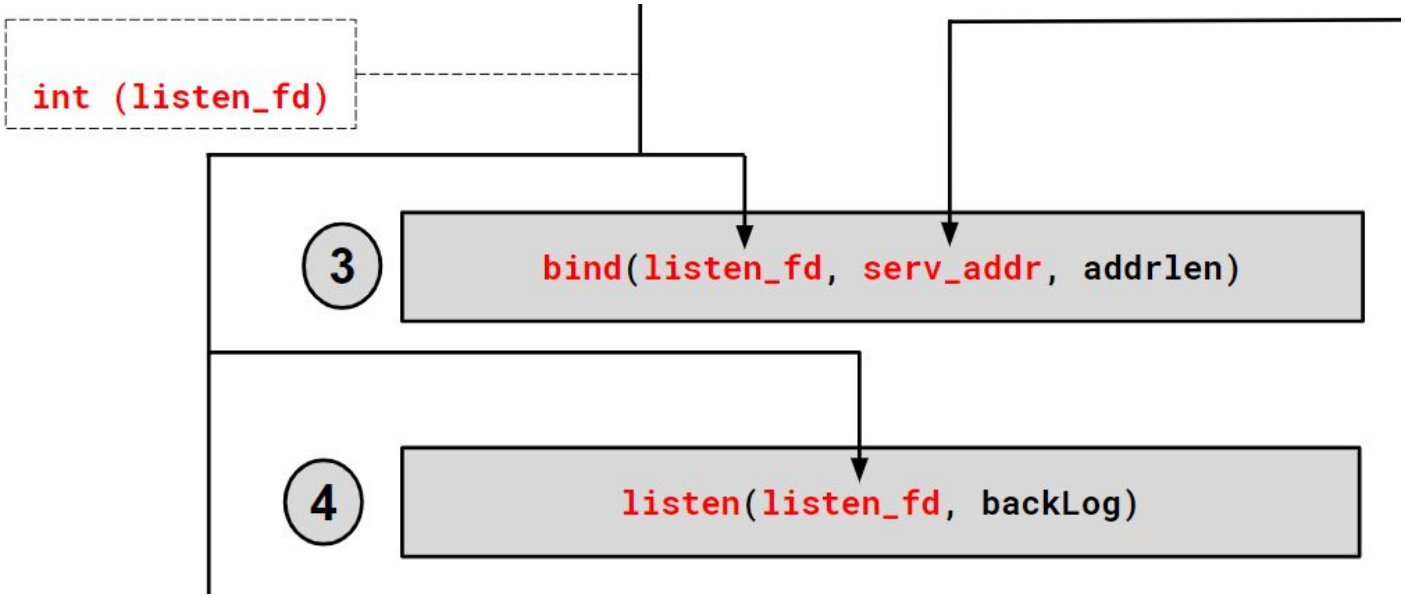
4.



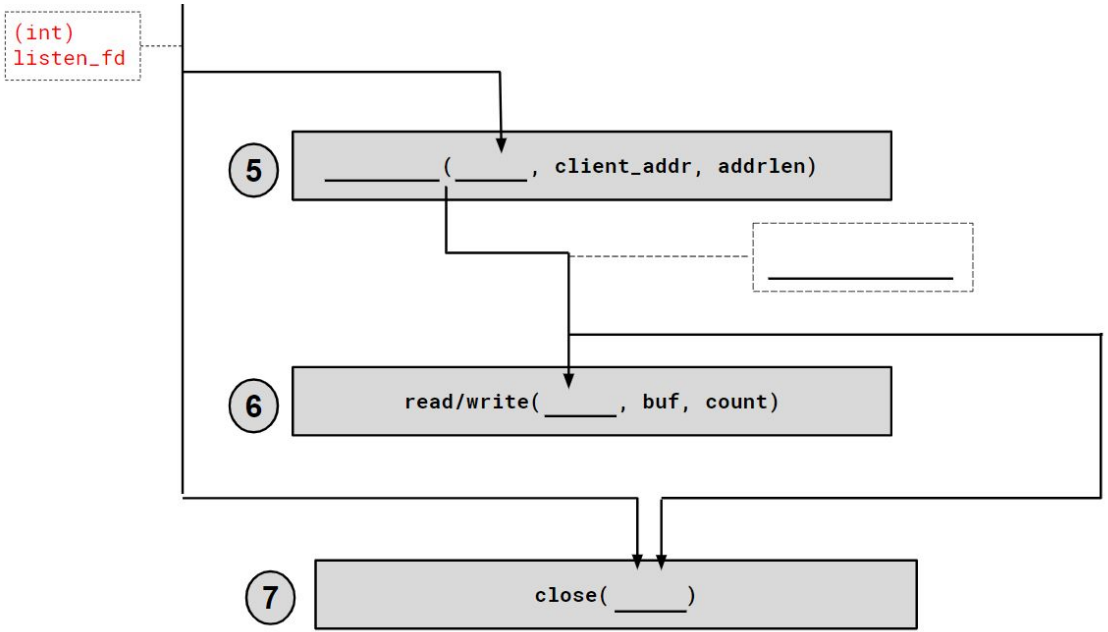
4. listen()

```
int connect (int sockfd,    // from 2
             int backlog); // max amount of
                           // pending connections
```

- Marks the socket as “passive” and used to listen for incoming connections
- Returns 0 on success, -1 on failure

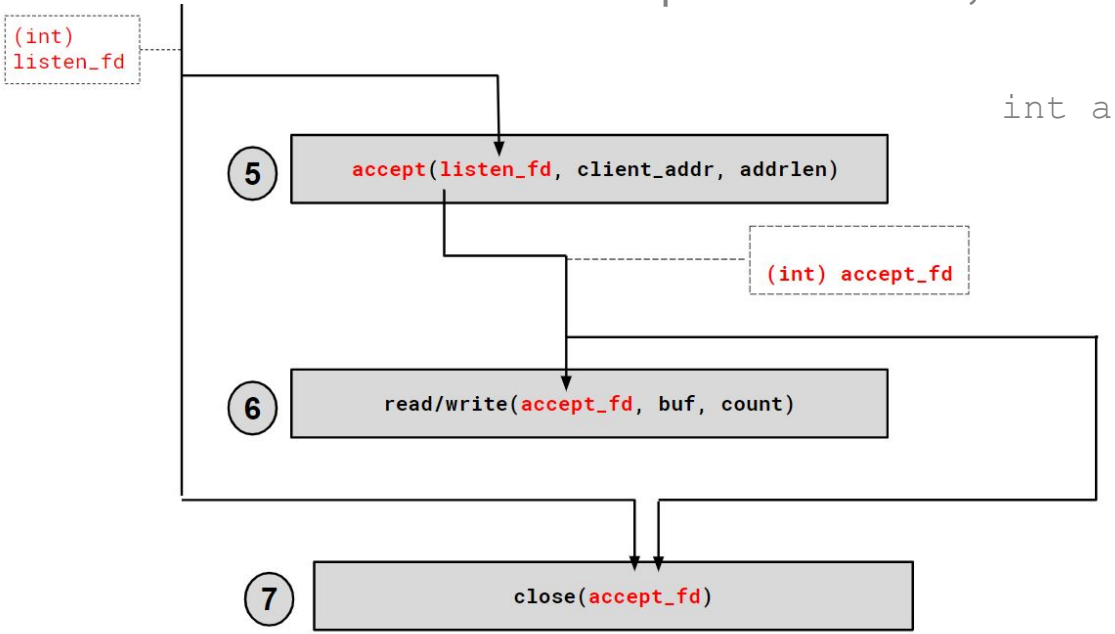


5. _____ 6. read/write and 7. close



5. accept 6. read/write and 7. close

- Accepts an incoming connection, returns client info from output parameters
- Returns a socket file descriptor on success, -1 on failure




```
int accept(int listen_fd,    // from 2
           struct sockaddr *serv_addr,
           socklen_t* addrlen) ;
```

Client Side vs Server Side

Client Side:

1. getaddrinfo // Lookup server ip
2. socket // Create a socket
3. connect() //Connect to server
4. read/write
5. close

Server Side:

1. getaddrinfo // Lookup our ip addr
2. socket // Create a socket
3. bind // prepare socket, links it to addr
4. listen // start waiting for connections
5. accept // handle the next connection
6. read/write
7. close

HW4 Tools

Using Netcat

1. Launch the server

   ```
   ./http333d <port> ../projdocs/ ../hw3/unit_test_indices/*
   ```

2. Connect with netcat

   ```
   nc -C <HostName> <port>
   ```

3. Write an HTTP request and send it

4. To exit netcat:
   - **`Ctrl+d`**

Writing an HTTP Request

- Example HTTP Request layout can be found in HttpRequest.h & Lecture 24 slides.

- Example file request:
  - `GET /static/test_tree/books/artofwar.txt HTTP/1.1`

- Example query request:
  - `GET /query?terms=books+of+war HTTP/1.1`

- To send a request, hit [Enter] *twice*
- Compare the output of `solution_binaries/http3d` to `./http3d`

HTTP REQUEST DEMO

BOOOOOST

Boost is a free C++ library that provides support for various tasks in C++

- **Note:** Boost does NOT follow the Google style guide!!!

Boost adds many string algorithms that you may have seen in Java

- Include with `#include <boost/algorithm/string.hpp>`

We are showcasing a few we think could be useful for HW4, but more can be found here:

- https://www.boost.org/doc/libs/1_60_0/doc/html/string_algo.html

trim

```
void boost::trim(string& input);
```

- Removes all leading and trailing whitespace from the string
- `input` is an input *and* output parameter (non-const reference)

```
string s("    HI    ");
boost::algorithm::trim(s);

// results in s == "HI"
```

replace_all

```
void boost::replace_all(string& input, const string& search,
                                       const string& format);
```

- Replaces all instances of `search` inside `input` with `format`

```
string s("ynrnrt");
boost::algorithm::replace_all(s, "nr", "e");

// results in s == "yeet"
```

replace_all

```
void boost::replace_all(string& input, const string& search,
                                       const string& format);
```

- Replaces all instances of `search` inside `input` with `format`

```
string s("queue?");
boost::algorithm::replace_all(s, "que", "q");

// results in s == "que?"
```

`replace_all()` guarantees that '`format`' will be in the final result if-and-only-if '`search`' existed.

`replace_all()` makes a *single* pass over `input`.

split

```
void boost::split(vector<string>& output,
          const string& input,
          boost::PredicateT match_on,
          boost::token_compress_mode_type compress);
```

- Split the string by the characters in `match_on`

```
boost::PredicateT boost::is_any_of(const string& tokens);
```

- Returns predicate that matches on any of the characters in tokens

split Examples

```
vector<string> tokens;
string s("I-am--split");

boost::split(tokens, s, boost::is_any_of("-"),
                        boost::token_compress_on);
// results in tokens == ["I", "am", "split"]

boost::split(tokens, s, boost::is_any_of("-"),
                        boost::token_compress_off);
// results in tokens == ["I", "am", "", “split"]
```

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){
```

Exercise 2

```
}
```