# ecet4640-lab5

1.0

Generated by Doxygen 1.9.8

# Chapter 1

# ecet4640-lab5

## 1.1 Intro

This program starts a Ipv4 server that listens on a port for incomming connections. For each new connection, it starts a thread. Connecting users send strings to execute various actions on the server.

## 1.2 Contributions

- On 9/29, Christian made the initial repository.
- On 9/29, Karl made the Makefile, copied the example client and server, created the Build, Data, File, Process, Util, and map modules.
- On 9/30, Karl added logic for reading the registered users file into a map and initializing/binding the server socket; created the Server and Connection modules.
- On 10/1, Paul and Karl did work on reading the server-settings.txt file, and creating a new thread for a client connection.
- On 10/2, Christian added a Log module and added some functionality to support various log levels.
- On 10/5, Paul, Karl, and Christian began adding different command line arguments the server could take.
- On 10/5, Karl and Paul fixed some bugs related to the multithreading.
- On 10/5, Paul added the myinfo and register commands and fixed typos.
- On 10/8, Christian added features for initializing the logger and fixed a bug related to disconnects.
- On 10/12, Karl, Paul and Christian changed the message format to the current format, set commands to be lowercase, and improved the messages sent to users.
- On 10/16, Christian added features to the _rand_age function.
- On 10/18, Karl fixed a segfault bug that occurred when users entered an invalid ID.
- On 10/19, Christian added the call to updating the registered file as users register
- On 10/19, Christian, Karl, and Paul added more command line arguments to the server.
- On 10/21, Christian implemented the advertisement feature and removed debug prints.

## 1.3 Overview

When the program runs, it reads from server-settings.txt to determine how it should be configured. As users register, they are added to a registered.txt file. The Users data structure has a dirty flag and a mutex that indicates when user data has been changed. A separate cleanup thread checks this dirty flag every few seconds to see if the registered.txt file should be updated.

Only one server process should be running at a given time. To that end, a running server creates a lockfile in the /tmp folder and deletes the lockfile when it is done. New servers will not be started if a lockfile exists, but the running server can be stopped by passing the command line argument 'stop' to the binary. There are other command line arguments available, as detailed below.

| Argument | Description | Calls |
|---|---|---|
| none | Defaults to RunCommand; runs server attached to terminal | RunCommand() |
| headless | Runs the server with .nohup, as a background process. | RunHeadless() |
| stop | Stops an existing server process if it is running. | StopCommand() |

**Author**

Karl Miller

Paul Shriner

Christian Messmer

# Chapter 2

# Compilation

## 2.1 Compilation Pipelines

There are several compilation pipelines, which are described in more detail in the Makefile comments.

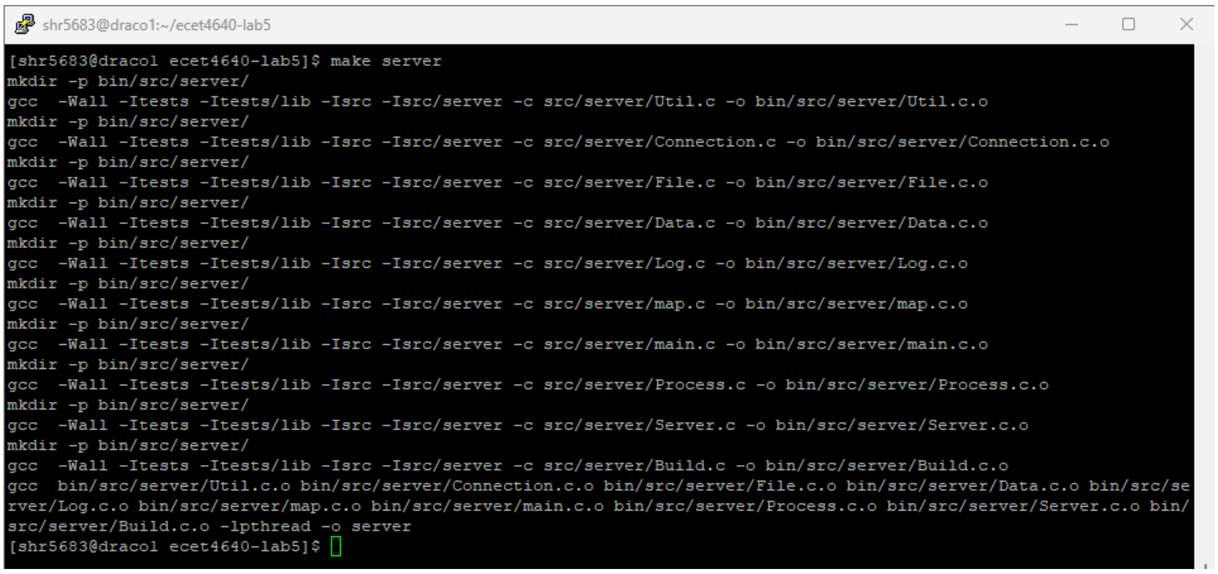
The first is for making and running the regular server process. Calling `make` executes this. It uses the files in `src/server` to generate the binary and runs it. This will run the binary after it is built, and the default command will cause it to run in the server. Executing `make server` will make the server binary without running it.

The second is for making the test binary. This compiles the files in `tests` and the files in `src/server`, but excludes `src/main.c` so that `tests/main_test.c` will be the program entry point instead. The tests use `CuTest`. The tests are not documented here in order to not inflate the documentation size any further.

## 2.2 Compiling and running

1. Copy the .zip file to the server.
2. Extract the zip file.
3. Enter the unzipped folder.
4. Run `make server`
5. Run `./server` to run the server attached to the shell.
6. Press ctrl+c to exit and close the server.
7. Run `./server headless` to run the server headlessly.
8. Run `./server stop` to stop the headless server.
9. If a better client is not available, you can use the example client to connect.
10. cd into the /example folder
11. run gcc client.c
12. run ./a.out and input '3000' as the port.

## 2.3 Screenshot of Compilation



```
[shr5683@draco1 ecet4640-lab5]$ make server
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Util.c -o bin/src/server/Util.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Connection.c -o bin/src/server/Connection.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/File.c -o bin/src/server/File.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Data.c -o bin/src/server/Data.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Log.c -o bin/src/server/Log.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/map.c -o bin/src/server/map.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/main.c -o bin/src/server/main.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Process.c -o bin/src/server/Process.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Server.c -o bin/src/server/Server.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -c src/server/Build.c -o bin/src/server/Build.c.o
gcc  bin/src/server/Util.c.o bin/src/server/Connection.c.o bin/src/server/File.c.o bin/src/server/Data.c.o bin/src/se
rver/Log.c.o bin/src/server/map.c.o bin/src/server/main.c.o bin/src/server/Process.c.o bin/src/server/Server.c.o bin/
src/server/Build.c.o -lpthread -o server
[shr5683@draco1 ecet4640-lab5]$
```

**Figure 2.1 Compiling on draco1**

## 2.4 Cleaning

`make clean` will clean all .o files and binaries.

# Chapter 3

# Topic Index

## 3.1 Topics

Here is a list of all topics with brief descriptions:

# Chapter 4

# Data Structure Index

## 4.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 5

# File Index

## 5.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 6

# Topic Documentation

## 6.1 Build

Functions for creating and populating data structures.

**Functions**

- User ∗ CreateUsersArray (char ∗∗userIDs, char ∗∗userNames, int recordsCount)
- map ∗ CreateUsersMap (User ∗usersArray, int recordsCount)

### 6.1.1 Detailed Description

### 6.1.2 Function Documentation

#### 6.1.2.1 CreateUsersArray()

```
User * CreateUsersArray (
            char ** userIDs,
            char ** userNames,
            int recordsCount )
```

Mallocs a new array of User structs.

**Parameters**

| | |
|---|---|
| *userIDs* | An array of userIDs to set. |
| *userNames* | An array of userNames corresponding to the userIDs. |
| *recordsCount* | The number of records in userIDs and userNames, and the size of the created an array. |

**Returns**

A malloced array of user structs.

Definition at line 9 of file Build.c.

#### 6.1.2.2 CreateUsersMap()

```
map * CreateUsersMap (
            User * usersArray,
            int recordsCount )
```

Given a user's array, initializes a new map that points to the underlying data in the array, using the user's ID as a key.

**Parameters**

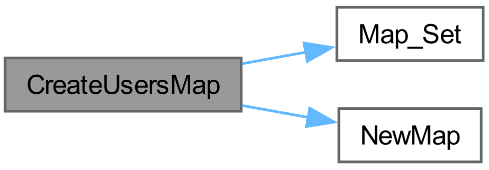
| | |
|---|---|
| *usersArray* | The array used to build the user's map. |
| *recordsCount* | The number of records in the user's array. |

**Returns**

A map

Definition at line 23 of file Build.c.

Here is the call graph for this function:



## 6.2 Connection

This module handles an individual user's active connection.

### Data Structures

- struct ClientShared
- struct Connection

**Functions**

- ClientShared ∗ InitializeShared (map ∗users_map, size_t send_buffer_size, size_t receive_buffer_size)
- void ∗ StartUpdateThread (void ∗parameter)
- void ∗ StartConnectionThread (void ∗p_connection)
- int MessageOrClose (char ∗send_buffer, char ∗receive_buffer, Connection ∗connection)
- void MessageAndClose (char ∗send_buffer, Connection ∗connection)
- void _help (Connection ∗connection, char ∗response)
- int _register (Connection ∗connection, char ∗response)
- int _myinfo (Connection ∗connection, char ∗response)
- void _who (char ∗response)
- void _rand_gpa (Connection ∗connection, char ∗response)
- void _rand_age (Connection ∗connection, char ∗response)
- void _advertisement (Connection ∗connection, char ∗response)

    *responds with a random ascii art*

### 6.2.1 Detailed Description

### 6.2.2 Function Documentation

#### 6.2.2.1 InitializeShared()

```
ClientShared * InitializeShared (
            map * users_map,
            size_t send_buffer_size,
            size_t receive_buffer_size )
```

Initializes the structure that shares data between connections and the server.

**Parameters**

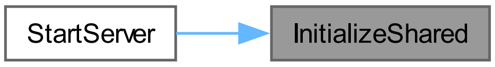
| | |
|---|---|
| *users_map* | The map of User structs. |

**Returns**

A pointer to the same ClientShared object seen by the connection threads.

Definition at line 19 of file Connection.c.

Here is the caller graph for this function:



#### 6.2.2.2 StartUpdateThread()

```
void * StartUpdateThread (
            void * parameter )
```

Starts an update thread. This thread is responsible for checking shared.dirty. If it is, it writes the user's data to a file and sets dirty to 0.

**Parameters**

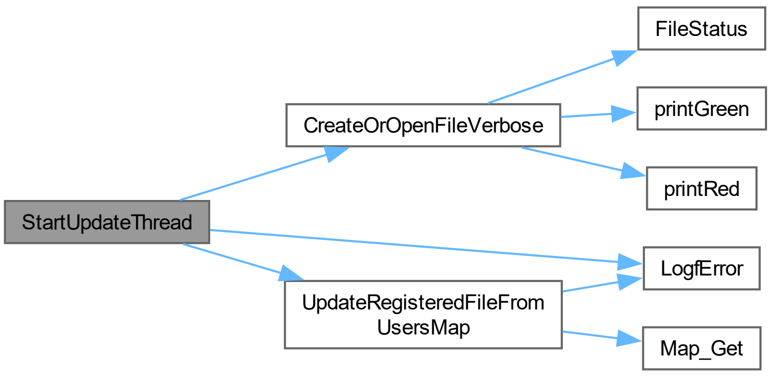
| | |
|---|---|
| *paramter* | None. |

**Returns**

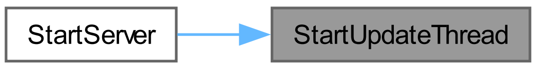
NULL

Definition at line 30 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 6.2.2.3 StartConnectionThread()

```
void * StartConnectionThread (
            void * connection )
```

Starts a connection thread

**Parameters**

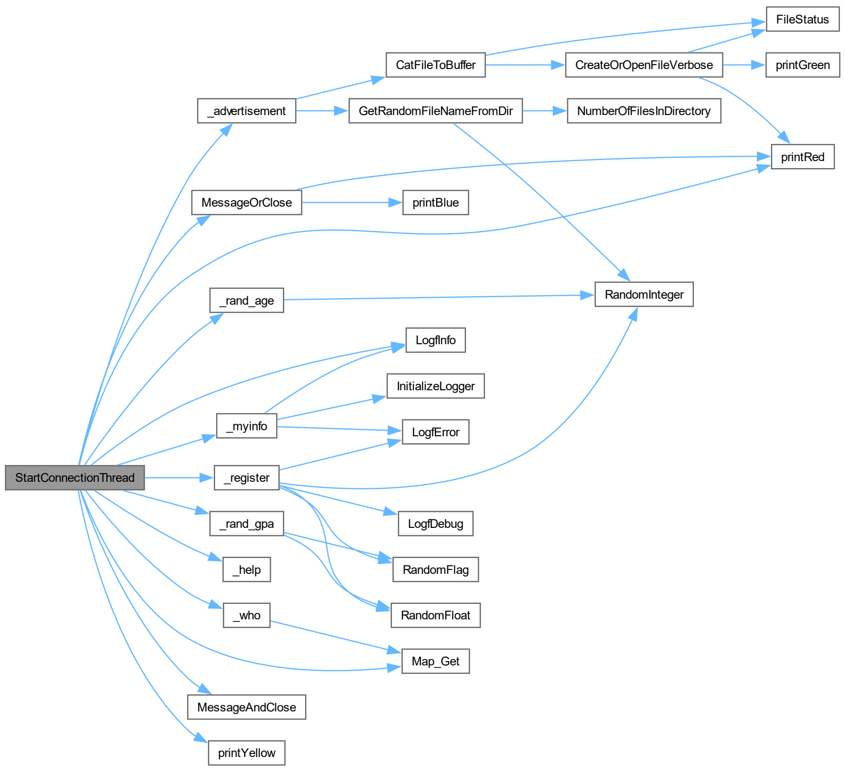
| | |
|---|---|
| *connection* | A pointer to a Connection structure from the server's connections array. |

**Returns**

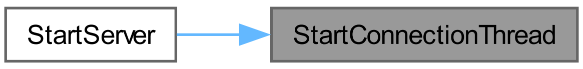
NULL

Definition at line 52 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 6.2.2.4 MessageOrClose()

```
int MessageOrClose (
            char * send_buffer,
            char * receive_buffer,
            Connection * connection )
```

Sends send_buffer to the socket referenced by connection, then memsets send_buffer to 0. Memsets receive_←
buffer to 0, then receives a message from the client. If this length is 0, assumes the connection was closed and sets connection->active to 0.

**Warning**

send_buffer and receive_buffer must be the size specified in shared.

**Parameters**

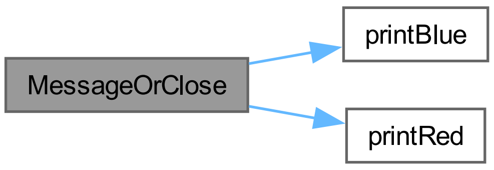
| | |
|---|---|
| *send_buffer* | A message to send to the client. |
| *receive_buffer* | The message received by the client. |
| *connection* | The socket's Connection |

**Returns**

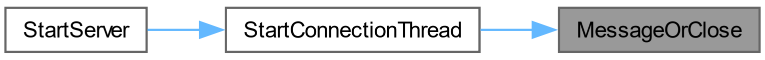
The number of bytes read into receive_buffer, or 0 if the connection closed.

Definition at line 163 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



### 6.2.2.5 MessageAndClose()

```
void MessageAndClose (
            char * send_buffer,
            Connection * connection )
```

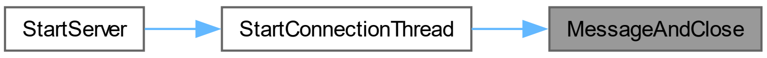
Sends send_buffer to the socket referenced by connection, then sets connection.active to 0.

**Parameters**

| | |
|---|---|
| *send_buffer* | The send buffer. Should be shared.send_length in size. |
| *connection* | The socket's Connection. |

Definition at line 191 of file Connection.c.

Here is the caller graph for this function:



### 6.2.2.6 _help()

```
void _help (
            Connection * connection,
            char * response )
```

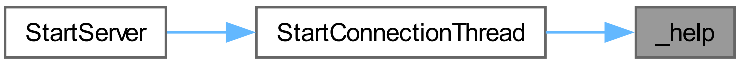
Returns the functions available to the user

**Parameters**

| | |
|---|---|
| *connection* | connection the user is on |
| *response* | fills the response buffer with what to send to the client |

Definition at line 200 of file Connection.c.

Here is the caller graph for this function:



#### 6.2.2.7 _register()

```
int _register (
            Connection * connection,
            char * response )
```

Registers the user from connection

**Parameters**

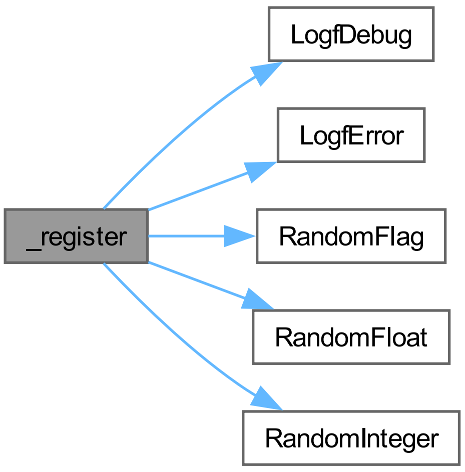
| | |
|---|---|
| *connection* | connection the users is on |
| *response* | fills the response buffer with what to send to the client |

**Returns**

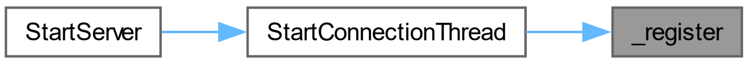
int 1 if successful, 0 if not

Definition at line 216 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 6.2.2.8 _myinfo()

```
int _myinfo (
             Connection * connection,
             char * response )
```

Returns the info of the user to the client

**Parameters**

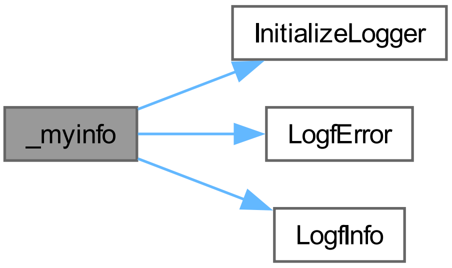
| | |
|---|---|
| *connection* | connection the user is on |
| *response* | fills the response buffer with what to send to the client |

**Returns**

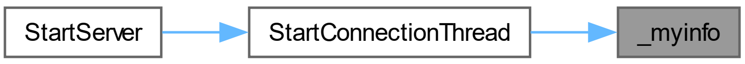
int 1 if successful, 0 if not

Definition at line 252 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 6.2.2.9 _who()

```
void _who (
             char * response )
```

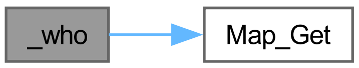
Sets response buffer to be a list a userIDs that are connected.

**Parameters**

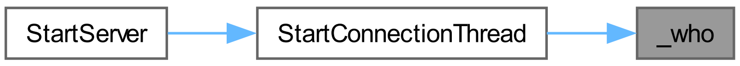
| | |
|---|---|
| *response* | fills the response buffer with what to send to the client |

Definition at line 273 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



### 6.2.2.10 _rand_gpa()

```
void _rand_gpa (
            Connection * connection,
            char * response )
```

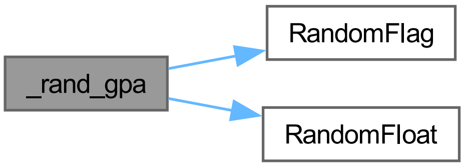
Randomly changes the gpa of the user

**Parameters**

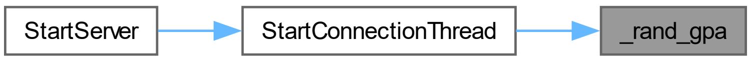
| | |
|---|---|
| *connection* | connection the user is on |
| *response* | fills the response buffer with what to send to the client |

Definition at line 288 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 6.2.2.11 _rand_age()

```
void _rand_age (
            Connection * connection,
            char * response )
```

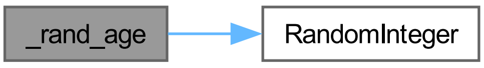
Randomly changes the age of the user

**Parameters**

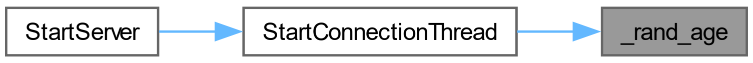
| | |
|---|---|
| *connection* | connection the user is on |
| *response* | fills the response buffer with what to send to the client |

Definition at line 303 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



### 6.2.2.12 _advertisement()

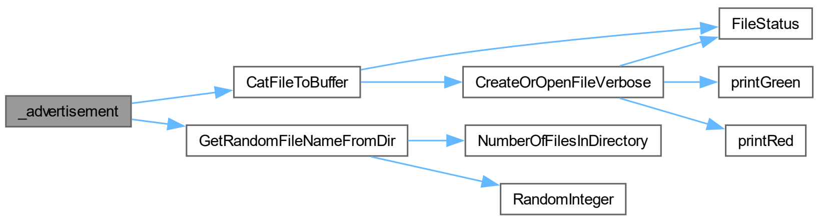
```
void _advertisement (
            Connection * connection,
            char * response )
```

**Parameters**

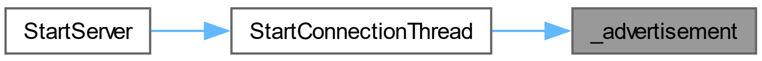
| | |
|---|---|
| *connection* | connection the user is on |
| *response* | fills the response buffer with what to send to the client |

Definition at line 315 of file Connection.c.

Here is the call graph for this function:



Here is the caller graph for this function:



## 6.3 Data

This module describes structures used in this program.

**Data Structures**

- struct User

**Macros**

- #define RECORD_COUNT 17
- #define ID_MAX_LENGTH 9
- #define NAME_MAX_LENGTH 21
- #define IP_LENGTH 16

**Variables**

- char ∗ accepted_userIDs [ ]
- char ∗ userFullNames [ ]
- char ∗ accepted_userIDs [ ]
- char ∗ userFullNames [ ]

## 6.3.1 Detailed Description

## 6.3.2 Macro Definition Documentation

### 6.3.2.1 RECORD_COUNT

```
#define RECORD_COUNT 17
```

The total count of records.

Definition at line 12 of file Data.h.

### 6.3.2.2 ID_MAX_LENGTH

```
#define ID_MAX_LENGTH 9
```

The amount of memory (bytes) required to be allocated for the ID field. Equal to the longest name in Data_IDs, "mes08346", plus the null terminator

Definition at line 17 of file Data.h.

### 6.3.2.3 NAME_MAX_LENGTH

```
#define NAME_MAX_LENGTH 21
```

The amount of memory (bytes) required to be allocated for the Name field. Equal to the longest name in Data_↩
Names, "Assefa Ayalew Yoseph", plus the null terminator

Definition at line 22 of file Data.h.

#### 6.3.2.4 IP_LENGTH

```
#define IP_LENGTH 16
```

The amount of memory (bytes) required to be allocated for the IP field. Large enough to store '111.111.111.111' plus the null terminator.

Definition at line 28 of file Data.h.

### 6.3.3 Variable Documentation

#### 6.3.3.1 accepted_userIDs [1/2]

```
char* accepted_userIDs[]
```

**Initial value:**

```
= {
    "chen",
    "bea1389",
    "bol4559",
    "cal6258",
    "kre5277",
    "lon1150",
    "mas9309",
    "mes08346",
    "mil7233",
    "nef9476",
    "nov7488",
    "pan9725",
    "rac3146",
    "rub4133",
    "shr5683",
    "vay3083",
    "yos2327"}
```

An array of the accepted userIDs.

Definition at line 7 of file Data.c.

#### 6.3.3.2 userFullNames [1/2]

```
char* userFullNames[]
```

**Initial value:**

```
= {
    "Weifeng Chen",
    "Christian Beatty",
    "Emily Bolles",
    "Cameron Calhoun",
    "Ty Kress",
    "Cody Long",
    "Caleb Massey",
    "Christian Messmer",
    "Karl Miller",
    "Jeremiah Neff",
    "Kaitlyn Novacek",
    "Joshua Panaro",
    "Caleb Rachocki",
    "Caleb Ruby",
    "Paul Shriner",
    "Alan Vayansky",
    "Assefa Ayalew Yoseph"}
```

An array of the full names, where the index of the name corresponds to the id in accepted_userIDs.

Definition at line 26 of file Data.c.

#### 6.3.3.3 accepted_userIDs [2/2]

```
char* accepted_userIDs[]  [extern]
```

An array of the accepted userIDs.

Definition at line 7 of file Data.c.

#### 6.3.3.4 userFullNames [2/2]

```
char* userFullNames[]  [extern]
```

An array of the full names, where the index of the name corresponds to the id in accepted_userIDs.

Definition at line 26 of file Data.c.

## 6.4 Files

This module contains functions that interact with files.

### Macros

- #define LOCKFILE "/tmp/lab5.lock"
- #define REGISTERED_FILE "registered.txt"
- #define SERVER_SETTINGS_FILE "server-settings.txt"
- #define ADS_DIR "ads"

### Functions

- short FileStatus (char ∗filename)
- FILE ∗ CreateOrOpenFileVerbose (char ∗filename, char ∗defaultContents)
- int ReadRegisteredFileIntoUsersMap (FILE ∗reg_file, map ∗users_map)
- void UpdateRegisteredFileFromUsersMap (FILE ∗reg_file, map ∗users_map)

  *Updates the registered file with of all users from user map that are marked as registered.*
- int NumberOfFilesInDirectory (char ∗dir_name)

  *Finds the number of files/directories in a given directory.*
- void GetRandomFileNameFromDir (char ∗dir_name, char ∗file_name)

  *Get the Random File Name From Dir object.*
- int ReadSettingsFileIntoSettingsMap (FILE ∗settings_file, map ∗settings_map)
- void CatFileToBuffer (char ∗file_name, char ∗buffer, size_t buffer_size)

  *Concatinates the contents of file_name into the buffer string.*

### 6.4.1 Detailed Description

### 6.4.2 Macro Definition Documentation

#### 6.4.2.1 LOCKFILE

```
#define LOCKFILE "/tmp/lab5.lock"
```

The presence of a lockfile indicates that a server process is already running. The lockfile contains the process ID of the running process.

Definition at line 16 of file File.h.

#### 6.4.2.2 REGISTERED_FILE

```
#define REGISTERED_FILE "registered.txt"
```

This file contains a list of registered users and their data, with fields tab-delimited.

**Note**

(1) The userID of the user.

(2) The age of the user.

(3) The GPA of the user.

(4) The IP address of the user.

(5) The last connection time of a user.

Definition at line 26 of file File.h.

#### 6.4.2.3 SERVER_SETTINGS_FILE

```
#define SERVER_SETTINGS_FILE "server-settings.txt"
```

Contains settings for the server. Each setting row contains a key, 0 or more space, an '=' symbol, and a value. Valid keys:

**Note**

port; the port the server will listen on.

send_buffer_size; the size of the send buffer

receive_buffer_size; the size of the receive buffer

backlog; the quantity of allowed backlogged unprocessed connections.

Definition at line 38 of file File.h.

### 6.4.2.4 ADS_DIR

```
#define ADS_DIR "ads"
```

Contains files of ascii art to sent to clients.

**Note**

should be the relative directory to the file the ads are in

Definition at line 45 of file File.h.

## 6.4.3 Function Documentation

### 6.4.3.1 FileStatus()

```
short FileStatus (
            char * filename )
```

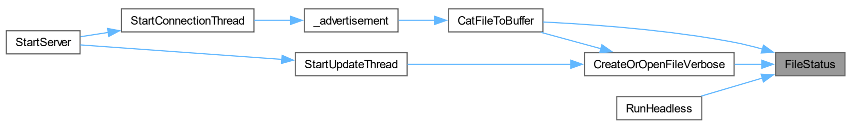
Determines if a file indicated by filename exists and is accesible by the user.

**Returns**

0 if the file does not exist. 1 if the file exists and the user has access. 2 if the file exists and the user does not have read and write permissions.

Definition at line 17 of file File.c.

Here is the caller graph for this function:



### 6.4.3.2 CreateOrOpenFileVerbose()

```
FILE * CreateOrOpenFileVerbose (
            char * filename,
            char * defaultContents )
```

Will call fopen() on a file and put default data inside, or nothing if defaultContents is NULL. Will print the results of its attempt.

**Warning**

Does not close the file; returns the open file.

**Note**

Prints successes and errors.

**Parameters**

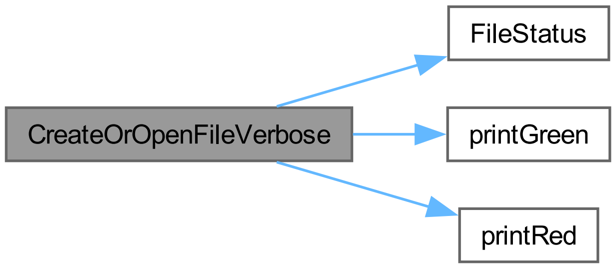
| | |
|---|---|
| *filename* | The file name to create or open. |
| *defaultContents* | The contents to put in the file, if creating a default file, or NULL if no contents should be added. |

**Returns**

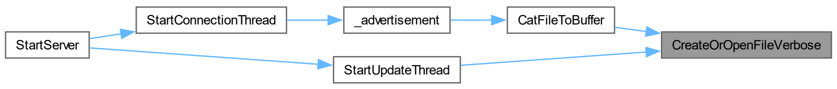
The opened file, or NULL on failure.

Definition at line 29 of file File.c.

Here is the call graph for this function:



Here is the caller graph for this function:



### 6.4.3.3 ReadRegisteredFileIntoUsersMap()

```
int ReadRegisteredFileIntoUsersMap (
             FILE * reg_file,
             map * users_map )
```

Reads the registered file into the user's map, by checking the IDs in the first field and setting the data at that location.

**Note**

Prints warnings and errors.

**Parameters**

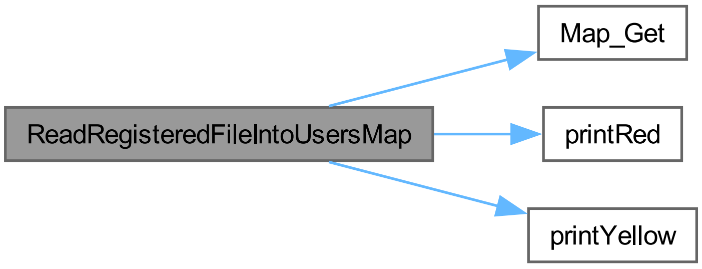
| | |
|---|---|
| *reg_file* | The registered users file, open for reading. |
| *users_map* | The user's map to read into. |

**Returns**

0 if success, error code if there was an error.

Definition at line 78 of file File.c.

Here is the call graph for this function:



#### 6.4.3.4 UpdateRegisteredFileFromUsersMap()

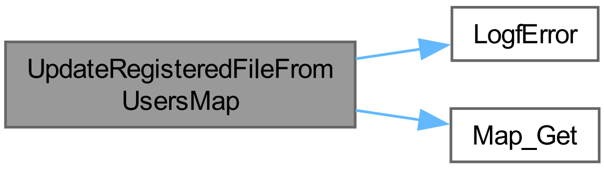
```
void UpdateRegisteredFileFromUsersMap (
            FILE * reg_file,
            map * users_map )
```

**Parameters**

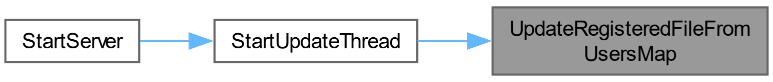
| | |
|---|---|
| *reg_file* | file to update to |
| *users_map* | the map of users to use to update |

Definition at line 111 of file File.c.

Here is the call graph for this function:



Here is the caller graph for this function:



### 6.4.3.5 NumberOfFilesInDirectory()

```
int NumberOfFilesInDirectory (
             char * dir_name )
```

**Parameters**

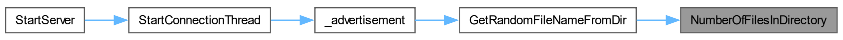
| | |
|---|---|
| *dir_name* | directory to count files from |

**Returns**

int number of files in the directory

Definition at line 127 of file File.c.

Here is the caller graph for this function:



#### 6.4.3.6 GetRandomFileNameFromDir()

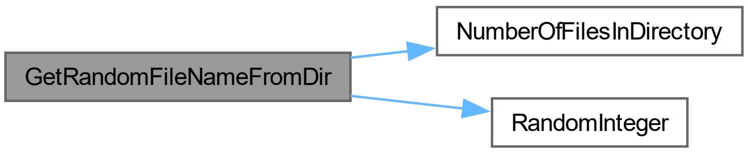
```
void GetRandomFileNameFromDir (
             char * dir_name,
             char * file_name )
```

**Parameters**

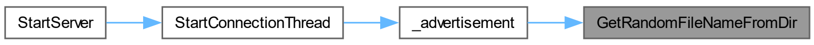
| | |
|---|---|
| *dir_name* | name of the director to get a file name of |
| *file_name* | sets the name of the file into file_name |

Definition at line 146 of file File.c.

Here is the call graph for this function:



Here is the caller graph for this function:



#### 6.4.3.7 ReadSettingsFileIntoSettingsMap()

```
int ReadSettingsFileIntoSettingsMap (
             FILE * settings_file,
             map * settings_map )
```

Reads the settings file into the settings map, by checking each line for a key value pair separated by a "=". It mallocs each key and value string it finds.

**Note**

Prints warnings and errors.

**Parameters**

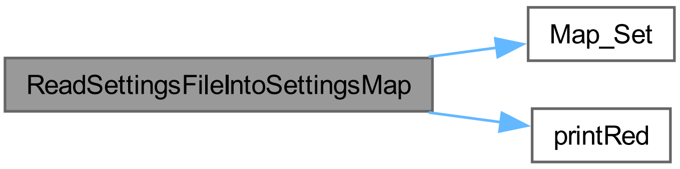
| | |
|---|---|
| *settings_file* | The settings file, open for reading. |
| *users_map* | The settings_map to read into. |

**Returns**

0 if success, an error code if there was an error.

Definition at line 165 of file File.c.

Here is the call graph for this function:



### 6.4.3.8 CatFileToBuffer()

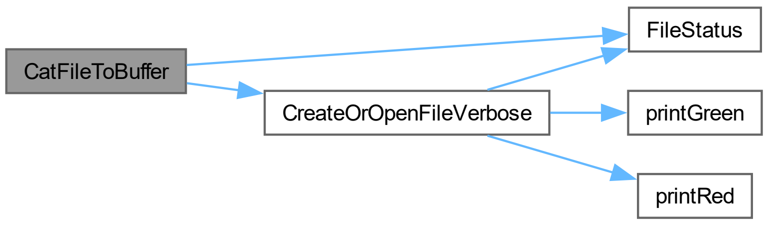
```
void CatFileToBuffer (
            char * file_name,
            char * buffer,
            size_t buffer_size )
```

**Parameters**

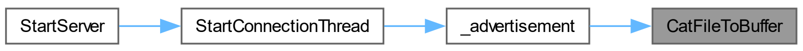
| | |
|---|---|
| *file_name* | file to concatinate |
| *buffer* | string to copy it to |
| *buffer_size* | max size of buffer |

Definition at line 190 of file File.c.

Here is the call graph for this function:



Here is the caller graph for this function:



## 6.5 Log

Handles logging; can be to the console or a file.

### Data Structures

- struct LogSettings

### Macros

- #define **TRACE** 0

  *These define the levels that a log can be printed as.*

### Functions

- void LogfFatal (const char ∗format,...)

  *Logs at a FATAL level the formatted string and parameters of the string.*
- void LogfError (const char ∗format,...)

  *Logs at an ERROR level the formatted string and parameters of the string.*
- void LogfWarning (const char ∗format,...)

  *Logs at a WARNING level the formatted string and parameters of the string.*
- void LogfInfo (const char ∗format,...)

  *Logs at a INFO level the formatted string and parameters of the string.*
- void LogfDebug (const char ∗format,...)

  *Logs at a DEBUG level the formatted string and parameters of the string.*
- void LogfTrace (const char ∗format,...)

  *Logs at a TRACE level the formatted string and parameters of the string.*
- void InitializeLogger (FILE ∗_printStream, char printLevel, char logLevel, char printAllToStdOut)

  *Instantiates the logger settings to run off of, must be called before logging can occur.*

### 6.5.1 Detailed Description

Log was initially going to also maintain a log.txt file that recorded user interactions with the server, but this feature was not finished. For now it just logs to the console.

### 6.5.2 Function Documentation

#### 6.5.2.1 LogfFatal()

```
void LogfFatal (
            const char * format,
             ... )
```

**Parameters**

| | |
|---|---|
| *format* | formatted string |
| ... | what to put in the formatted string |

Definition at line 28 of file Log.c.

#### 6.5.2.2 LogfError()

```
void LogfError (
            const char * format,
             ... )
```

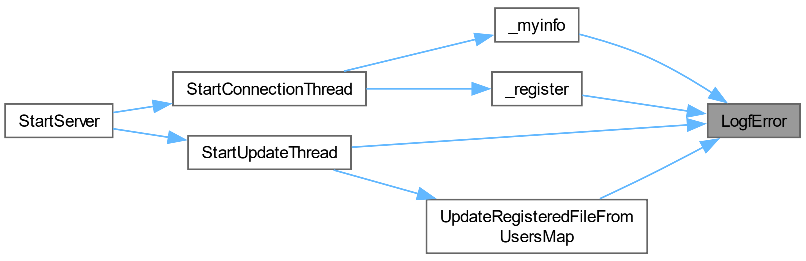
Is used

**Parameters**

| | |
|---|---|
| *format* | formatted string |
| ... | what to put in the formatted string |

Definition at line 35 of file Log.c.

Here is the caller graph for this function:



#### 6.5.2.3 LogfWarning()

```
void LogfWarning (
             const char * format,
              ... )
```

**Parameters**

| | |
|---|---|
| *format* | formatted string |
| ... | what to put in the formatted string |

Definition at line 42 of file Log.c.

#### 6.5.2.4 LogfInfo()

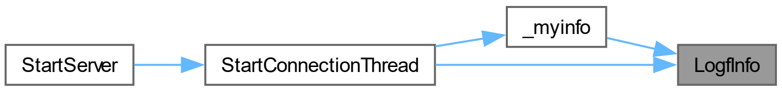
```
void LogfInfo (
             const char * format,
              ... )
```

**Parameters**

| | |
|---|---|
| *format* | formatted string |
| ... | what to put in the formatted string |

Definition at line 49 of file Log.c.

Here is the caller graph for this function:



#### 6.5.2.5 LogfDebug()

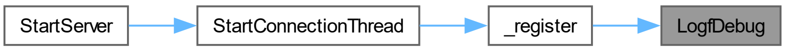
```
void LogfDebug (
             const char * format,
              ... )
```

**Parameters**

| | |
|---|---|
| *format* | formatted string |
| ... | what to put in the formatted string |

Definition at line 56 of file Log.c.

Here is the caller graph for this function:



#### 6.5.2.6 LogfTrace()

```
void LogfTrace (
            const char * format,
             ...  )
```

**Parameters**

| | |
|---|---|
| *format* | formatted string |
| *...* | what to put in the formatted string |

Definition at line 63 of file Log.c.

#### 6.5.2.7 InitializeLogger()

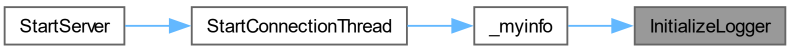
```
void InitializeLogger (
            FILE * _printStream,
            char printLevel,
            char logLevel,
            char printAllToStdOut )
```

**Parameters**

| | |
|---|---|
| *_printStream* | output stream to print at |
| *outputLevel* | mimumum level needed to print to the printstream |
| *logLevel* | minimum level needed to print to log file |
| *printAllToStdOut* | will print everyting to stdout if not set to 0 as well as _printStream |

Definition at line 9 of file Log.c.

Here is the caller graph for this function:



## 6.6 Map

Functions that implement a hash map data structure.

### Data Structures

- struct _map_bucket

  *map_bucket is an endpoint in the map. It is also a node in a linked list; if there were collisions, then the buckets are appended to the linked list at that location, then traversed until the matching key is found.*
- struct map

  *A map. Stores key-value pairs for near constant lookup and insertion time.*
- struct map_result

  *The result of a map retrieval.*

### Functions

- map ∗ NewMap (int capacity)
- void Map_Set (map ∗a_map, char ∗key, void ∗value)

  *Sets a value in the map.*
- map_result Map_Get (map ∗a_map, char ∗key)

  *Gets a value from the map. It will return a map_get_result describing whether it was succesful, and possibly containing the data sought, or NULL if it was unsuccesful.*
- map_result Map_Delete (map ∗a_map, char ∗key, short free_it)

  *Deletes a key from the map. Returns a map_get_result describing whether the delete was succesful and containing the removed data, if extant.*

### 6.6.1 Detailed Description

Karl's take on a simple hash map structure, which maps strings to void pointers. You can use casting to convert the void pointers into most of whatever else is needed.

Example usage, casting an int into the data part of the map.
```
int myfunc() {
    map *mymap = NewMap(100);
    Map_Set(mymap, "age", (void*)55);
    map_result result = Map_Get(mymap, "age");
    int age;
    if(result.found) {
        age = (int) map_result.data;
    }
}
```

Note, with this simple implementation, the map cannot change its capacity. A change to its capacity would change the hashing.

Ultimately there are really only three things you need to do with the map.

Initialize it, with some capacity larger than you will use. EG map ∗ mymap = NewMap(100). The bigger it is, the fewer collisions (which are pretty rare anyway).

Set some values in it. Eg Map_Set(mymap, "key", &value);

You can cast numbers to void pointers to put them in the map, or you can use the pointers as references to, for example, strings malloced somewhere.

Get some values from it. Eg void∗ myval = Map_Get(mymap, "key");

Delete some values from it. For example Map_Delete(mymap, "key", 0);

Note that the last parameter, 'free it', tells the map whether it should call 'free' on the underyling data in memory. If this is 1, and the underyling data is not a reference to a malloced part of the heap, errors will result.

### 6.6.2 Function Documentation

#### 6.6.2.1 NewMap()

```
map * NewMap (
             int capacity )
```

Creates a new map. The map capacity will be a power of 2 that is large enough to contain the estimated size.

**Parameters**

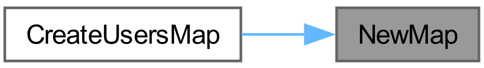
| | |
|---|---|
| *capacity* | The estimated required capacity of the map. |

**Returns**

A pointer to the heap allocated map.

Definition at line 49 of file map.c.

Here is the caller graph for this function:



#### 6.6.2.2 Map_Set()

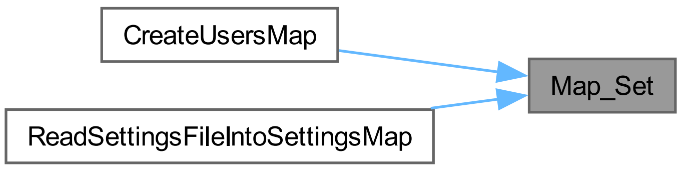
```
void Map_Set (
             map * a_map,
             char * key,
             void * value )
```

**Parameters**

| | |
|---|---|
| *map* | The map to set a key in. |
| *key* | The key to use. |
| *keylen* | The length of the key. |
| *value* | The pointer to the data stored at that location. |

Definition at line 89 of file map.c.

Here is the caller graph for this function:



#### 6.6.2.3 Map_Get()

```
map_result Map_Get (
             map * a_map,
             char * key )
```

**Parameters**

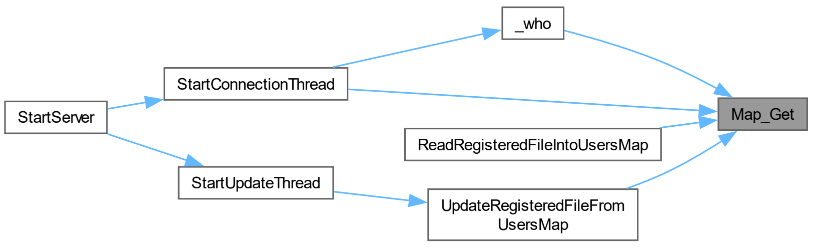
| | |
|---|---|
| *map* | The map to retrieve from. |
| *key* | The key of the item. |

**Returns**

A map_get_result containing the sought data.

Definition at line 119 of file map.c.

Here is the caller graph for this function:


_who
StartConnectionThread
StartServer
ReadRegisteredFileIntoUsersMap
Map_Get
StartUpdateThread
UpdateRegisteredFileFrom UsersMap

#### 6.6.2.4 Map_Delete()

```
map_result Map_Delete (
             map * a_map,
             char * key,
             short free_it )
```

**Parameters**

| | |
|---|---|
| *map* | The map to delete the key from. |
| *key* | The key to delete. |
| *free↩ _it* | Whether to call free() on the underlying data. |

**Returns**

A map_get_result with the data that was removed.

Definition at line 154 of file map.c.

## 6.7 Server

Functions for running the server.

### Data Structures

- struct ServerProperties

### Functions

- int StartServer (map ∗users_map)
- Connection ∗ NextAvailableConnection ()
- int InitializeServer ()

### 6.7.1 Detailed Description

### 6.7.2 Function Documentation

#### 6.7.2.1 StartServer()

```
int StartServer (
          map * users_map )
```

Starts the server.

**Note**

This is a blocking call that will start a loop until SIGINT is received.

**Parameters**

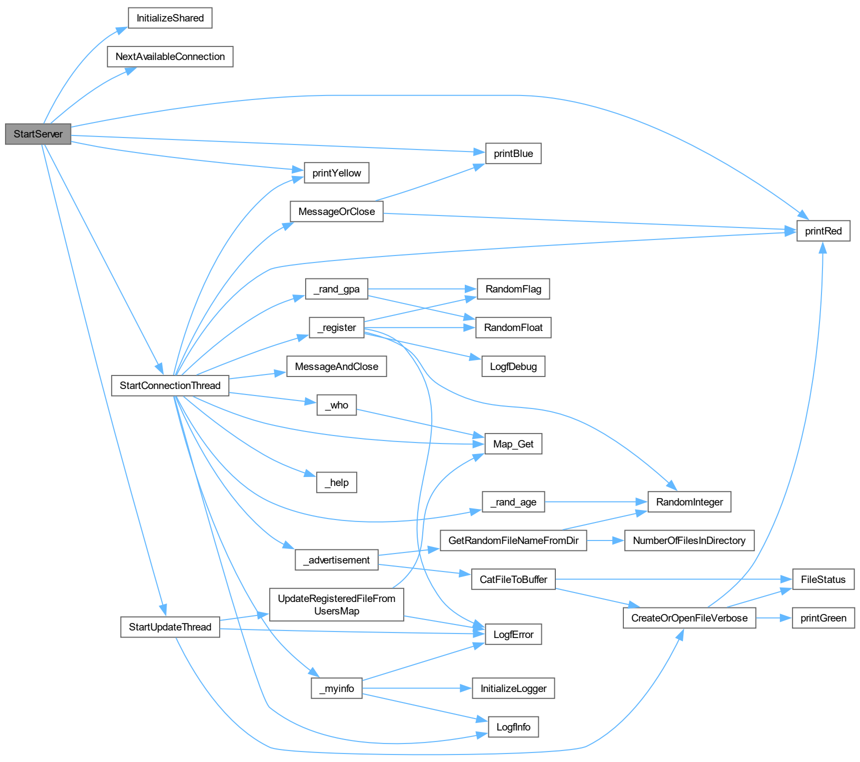
| | |
|---|---|
| *users_map* | The user's map. |

**Returns**

1 if the server ran and shutdown gracefully, 0 if there was an error during setup.

Definition at line 103 of file Server.c.

Here is the call graph for this function:



### 6.7.2.2 NextAvailableConnection()

```
Connection * NextAvailableConnection ( )
```

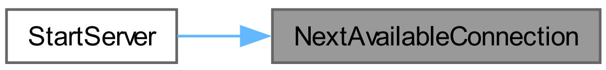
Iterates through the Connections array until it finds one whose 'active' field is false and returns it. If it iterates through the array and fails to find a connection, it returns NULL.

**Returns**

A Connection struct or null.

Definition at line 156 of file Server.c.

Here is the caller graph for this function:



#### 6.7.2.3 InitializeServer()

```
int InitializeServer ( )
```

Initializes the server properties structure and the structures for holding Connection objects.

**Note**

Prints initialization status.

**Returns**

1 of it was able to initialize, otherwise 0.

## 6.8 Util

Utility functions used by various modules but not dependent on any other modules.

**Macros**

- #define COLOR_RED "\e[38;2;255;75;75m"
- #define COLOR_GREEN "\e[38;2;0;240;0m"
- #define COLOR_YELLOW "\e[38;2;255;255;0m"
- #define COLOR_BLUE "\e[38;2;0;240;240m"
- #define COLOR_RESET "\e[0m"

**Functions**

- void printRed (const char ∗format,...)
- void printGreen (const char ∗format,...)
- void printYellow (const char ∗format,...)
- void printBlue (const char ∗format,...)
- int RandomInteger (int min, int max)
- float RandomFloat (float min, float max)
- short RandomFlag (float percentage_chance)

### 6.8.1 Detailed Description

### 6.8.2 Macro Definition Documentation

#### 6.8.2.1 COLOR_RED

```
#define COLOR_RED "\e[38;2;255;75;75m"
```

A virtual terminal escape sequence to print foreground red.

Definition at line 10 of file Util.h.

#### 6.8.2.2 COLOR_GREEN

```
#define COLOR_GREEN "\e[38;2;0;240;0m"
```

A VTE for green.

Definition at line 12 of file Util.h.

#### 6.8.2.3 COLOR_YELLOW

```
#define COLOR_YELLOW "\e[38;2;255;255;0m"
```

A VTE for yellow.

Definition at line 14 of file Util.h.

#### 6.8.2.4 COLOR_BLUE

```
#define COLOR_BLUE "\e[38;2;0;240;240m"
```

A VTE for blue.

Definition at line 16 of file Util.h.

#### 6.8.2.5 COLOR_RESET

```
#define COLOR_RESET "\e[0m"
```

A VTE to reset the printing color.

Definition at line 18 of file Util.h.

### 6.8.3 Function Documentation

#### 6.8.3.1 printRed()

```
void printRed (
             const char * format,
              ... )
```

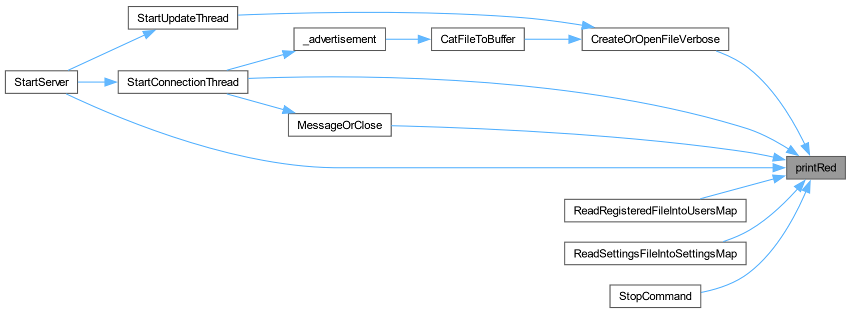
Prints to the console in red.

**Parameters**

| | |
|---|---|
| *format* | A format, as printf. |
| ... | args, as printf. |

Definition at line 11 of file Util.c.

Here is the caller graph for this function:



### 6.8.3.2 printGreen()

```
void printGreen (
             const char * format,
              ...  )
```

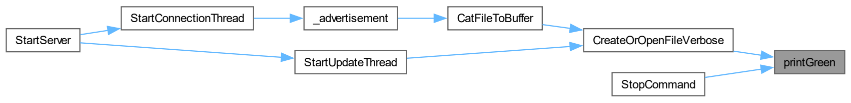
Prints to the console in green.

**Parameters**

| | |
|---|---|
| *format* | A format, as printf. |
| ... | args, as printf. |

Definition at line 20 of file Util.c.

Here is the caller graph for this function:



### 6.8.3.3 printYellow()

```
void printYellow (
             const char * format,
              ...  )
```

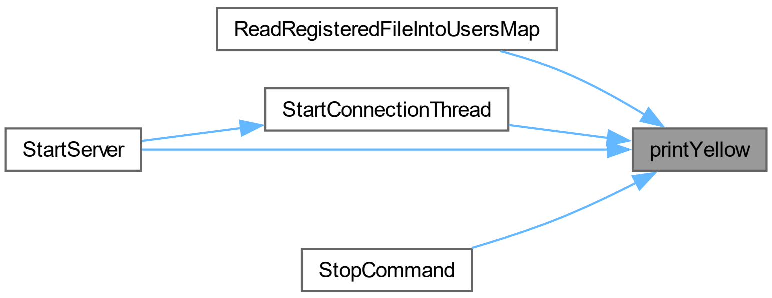
Prints to the console in yellow.

**Parameters**

| | |
|---|---|
| *format* | A format, as printf. |
| ... | args, as printf. |

Definition at line 29 of file Util.c.

Here is the caller graph for this function:



### 6.8.3.4 printBlue()

```
void printBlue (
             const char * format,
              ... )
```

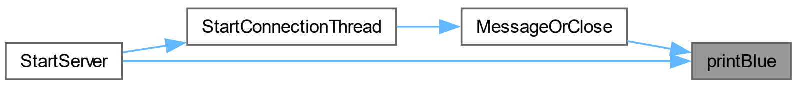
Prints to the console in blue.

**Parameters**

| | |
|---|---|
| *format* | A format, as printf. |
| ... | args, as printf. |

Definition at line 38 of file Util.c.

Here is the caller graph for this function:



#### 6.8.3.5 RandomInteger()

```
int RandomInteger (
             int min,
             int max )
```

Returns an integer between min and max.

**Parameters**

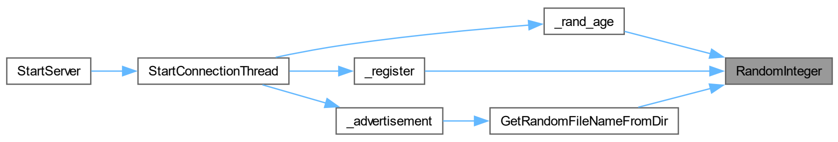
| | |
|---|---|
| *min* | The minimum, inclusive. |
| *max* | The maximum, inclusive. |

**Returns**

A random integer between min and max.

Definition at line 47 of file Util.c.

Here is the caller graph for this function:



#### 6.8.3.6 RandomFloat()

```
float RandomFloat (
             float min,
             float max )
```

Returns a float between min and max.

**Parameters**

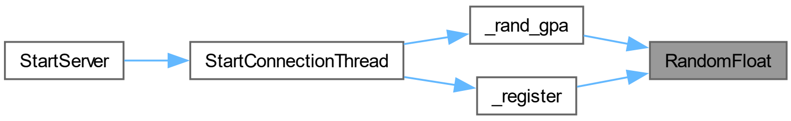
| | |
|---|---|
| *min* | The minimum, inclusive. |
| *max* | The maximum, inclusive. |

**Returns**

A random integer between min and max.

Definition at line 53 of file Util.c.

Here is the caller graph for this function:



#### 6.8.3.7 RandomFlag()

```
short RandomFlag (
            float percentage_chance )
```

Returns 1, percentage_chance of the time.

**Parameters**

| | |
|---|---|
| *percentage_chance* | The chance to return 1. |

**Note**

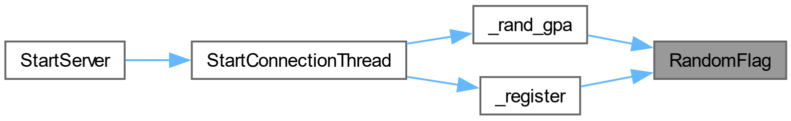
If percentage_chance $>$ 1, this will always return true.

**Returns**

1 or 0

Definition at line 60 of file Util.c.

Here is the caller graph for this function:



## 6.9 Process

### Functions

- void RunHeadless (char ∗processName)
- void **StopCommand** ()

  *Stops the server that is running headlessly and prints the results of running the command.*

**Variables**

- User ∗ users_array
- map ∗ users_map
- map ∗ settings_map
- char ∗ default_settings
- int active_clients

## 6.9.1 Detailed Description

## 6.9.2 Function Documentation

### 6.9.2.1 RunHeadless()

```
void RunHeadless (
             char * processName )
```

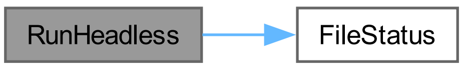
Uses nohup `./{processName} run` to run the process headlessly.

**Parameters**

| | |
|---|---|
| *processName* | The name of the currently running process, by default, 'server'. |

Definition at line 190 of file Process.c.

Here is the call graph for this function:



## 6.9.3 Variable Documentation

### 6.9.3.1 users_array

```
User* users_array
```

The array of users. This will be populated on initialize by functions in Build.

Definition at line 20 of file Process.c.

#### 6.9.3.2 users_map

```
map* users_map
```

The map of userIDs to users. Populated on Initialize by functions in Build.

Definition at line 22 of file Process.c.

#### 6.9.3.3 settings_map

```
map* settings_map
```

The map of settings stored in the server settings file.

Definition at line 24 of file Process.c.

#### 6.9.3.4 default_settings

```
char* default_settings
```

**Initial value:**

```
= "port                   = 3000\n"
                             "send_buffer_size    = 1024\n"
                             "receive_buffer_size = 1024\n"
                             "backlog             = 10\n"
                             "max_connections     = 20\n"
                             "log_file            = log.txt\n"
                             "log_level           = 1\n"
                             "log_to_console      = true"
```

The default contents of the settings file, if it doesn't exist.

Definition at line 26 of file Process.c.

#### 6.9.3.5 active_clients

```
int active_clients
```

The number of active clients.

Definition at line 36 of file Process.c.

# Chapter 7

# Data Structure Documentation

## 7.1 _map_bucket Struct Reference

map_bucket is an endpoint in the map. It is also a node in a linked list; if there were collisions, then the buckets are appended to the linked list at that location, then traversed until the matching key is found.

### 7.1.1 Detailed Description

Definition at line 81 of file map.h.

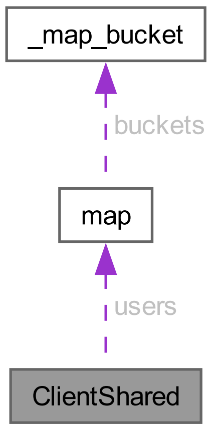
The documentation for this struct was generated from the following file:

- src/server/map.h

## 7.2 ClientShared Struct Reference

```
#include <Connection.h>
```

Collaboration diagram for ClientShared:



**Data Fields**

- map ∗ users
- pthread_mutex_t mutex
- short dirty
- short shutting_down
- size_t send_buffer_size
- size_t receive_buffer_size

## 7.2.1 Detailed Description

Shared between the Connections and the Server.

Definition at line 17 of file Connection.h.

## 7.2.2 Field Documentation

### 7.2.2.1 users

```
map* users
```

The user's map.

Definition at line 19 of file Connection.h.

### 7.2.2.2 mutex

```
pthread_mutex_t mutex
```

A mutex to provide mutual-exclusion to connection threads operating on the user's map.

Definition at line 21 of file Connection.h.

### 7.2.2.3 dirty

```
short dirty
```

Whether there were changes to the user's map that need to be saved in a file.

Definition at line 23 of file Connection.h.

### 7.2.2.4 shutting_down

```
short shutting_down
```

Whether the server is shutting down.

Definition at line 25 of file Connection.h.

#### 7.2.2.5 send_buffer_size

```
size_t send_buffer_size
```

Passed along from server settings at the time shared is initialized

Definition at line 28 of file Connection.h.

#### 7.2.2.6 receive_buffer_size

```
size_t receive_buffer_size
```

Passed along from server settings at the time shared is initialized

Definition at line 30 of file Connection.h.

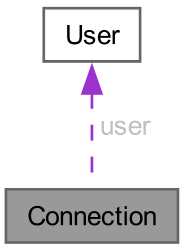
The documentation for this struct was generated from the following file:

- src/server/Connection.h

## 7.3 Connection Struct Reference

```
#include <Connection.h>
```

Collaboration diagram for Connection:



**Data Fields**

- ConnectionState status
- int socket
- struct sockaddr_in address
- socklen_t address_length
- pthread_t thread_id
- time_t time_connected
- ClientState state
- User ∗ user

### 7.3.1 Detailed Description

Data for a single client socket connection to the server. Passed into the thread runner as the parameter.

Definition at line 46 of file Connection.h.

### 7.3.2 Field Documentation

#### 7.3.2.1 status

```
ConnectionState status
```

Whether this connection is closed (0) or active (1) or closing (2). This is set by the SERVER just prior to starting the thread. The thread sets it back to 0 when it is completely done.

Definition at line 48 of file Connection.h.

#### 7.3.2.2 socket

```
int socket
```

The underlying socket file descriptor.

Definition at line 50 of file Connection.h.

#### 7.3.2.3 address

```
struct sockaddr_in address
```

The socket address of the connection.

Definition at line 52 of file Connection.h.

#### 7.3.2.4 address_length

```
socklen_t address_length
```

The actual size of the client address; send by accept.

Definition at line 54 of file Connection.h.

#### 7.3.2.5 thread_id

```
pthread_t thread_id
```

The pthread ID of this client thread.

Definition at line 56 of file Connection.h.

#### 7.3.2.6 time_connected

```
time_t time_connected
```

When the client connected.

Definition at line 58 of file Connection.h.

#### 7.3.2.7 state

```
ClientState state
```

The client state.

Definition at line 60 of file Connection.h.

#### 7.3.2.8 user

```
User* user
```

The user associated with this client.

Definition at line 62 of file Connection.h.

The documentation for this struct was generated from the following file:

- src/server/Connection.h

## 7.4 LogSettings Struct Reference

### Data Fields

- FILE ∗ **ostream**
  *output the logger should go to*
- char **printLevel**
  *minimum level that the log should be in order to print stdout*
- char **logLevel**
  *minimum level that the log should be for it to be printed to log file*
- char **printAllToStdOut**
  *if not 0 will print to stdout as well as ostream*

### 7.4.1 Detailed Description

Definition at line 27 of file Log.h.

The documentation for this struct was generated from the following file:

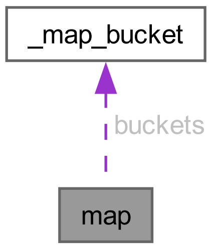
- src/server/Log.h

## 7.5 map Struct Reference

A map. Stores key-value pairs for near constant lookup and insertion time.

```
#include <map.h>
```

Collaboration diagram for map:



### 7.5.1 Detailed Description

**Note**

Use NewMap() to create a new map.

Use Map_Set() to set a key in the map.

Use Map_Get() to get a value from the map.

The values stored are of type void pointer.

Definition at line 101 of file map.h.

The documentation for this struct was generated from the following file:

- src/server/map.h

## 7.6 map_result Struct Reference

The result of a map retrieval.

```
#include <map.h>
```

### 7.6.1 Detailed Description

Definition at line 111 of file map.h.

The documentation for this struct was generated from the following file:

- src/server/map.h

## 7.7 ServerProperties Struct Reference

```
#include <Server.h>
```

#### Data Fields

- uint16_t port
- size_t send_buffer_size
- size_t receive_buffer_size
- int backlog
- int active_connections
- int max_connections
- time_t time_started

### 7.7.1 Detailed Description

Defines the properties for the server.

Defined in server-settings.txt, a configuration file.

Definition at line 20 of file Server.h.

### 7.7.2 Field Documentation

#### 7.7.2.1 port

```
uint16_t port
```

The port the server will connect on.

Definition at line 22 of file Server.h.

#### 7.7.2.2 send_buffer_size

```
size_t send_buffer_size
```

The size of each send buffer.

Definition at line 24 of file Server.h.

#### 7.7.2.3 receive_buffer_size

```
size_t receive_buffer_size
```

The size of each receive buffer.

Definition at line 26 of file Server.h.

#### 7.7.2.4 backlog

```
int backlog
```

The size of the backlog of unprocessed connections.

Definition at line 28 of file Server.h.

#### 7.7.2.5 active_connections

```
int active_connections
```

The number of active connections.

Definition at line 30 of file Server.h.

#### 7.7.2.6 max_connections

```
int max_connections
```

The maximum number of active connections the server supports.

Definition at line 32 of file Server.h.

#### 7.7.2.7 time_started

```
time_t time_started
```

The time the server was started.

Definition at line 34 of file Server.h.

The documentation for this struct was generated from the following file:

- src/server/Server.h

## 7.8 User Struct Reference

```
#include <Data.h>
```

**Data Fields**

- char **id** [ID_MAX_LENGTH]

  *The user ID; equal to an element in accepted_userIDs.*
- char **name** [NAME_MAX_LENGTH]

  *The user's real name; equal to an element in userFullNames.*
- int **age**

  *The user's age, randomized between 18 and 22.*
- float **gpa**

  *The user's gpa, randomized between 2.5 and 4.0.*
- short **connected**

  *Whether the user is connected.*
- char **ip** [IP_LENGTH]

  *The last IP used by the user; set on connection.*
- long **lastConnection**

  *A unix timestamp representing the last time a user connected.*
- short **registered**

  *Whether user has executed the 'register' command.*

## 7.8.1 Detailed Description

A User of this server. The ID and Name fields are populated initially. GPA and age are populated at the time a user is registered, and saved and loaded from a file. Active is set and unset when a user connects. IP is set each time a user connects, and saved in the file.

Definition at line 44 of file Data.h.

The documentation for this struct was generated from the following file:

- src/server/Data.h

# Chapter 8

# File Documentation

## 8.1 Build.c

```

#include <stdlib.h>
#include <string.h>
#include "Build.h"

User * CreateUsersArray(char ** userIDs, char ** userNames, int recordsCount)
{
    size_t uarr_size = sizeof(User) * recordsCount;
    User * uarr = malloc(uarr_size);
    memset(uarr, 0, uarr_size);
    int i;
    for(i = 0; i < recordsCount; i++)
    {
        strcpy(uarr[i].id, userIDs[i]);
        strcpy(uarr[i].name, userNames[i]);
    }
    return uarr;
}

map * CreateUsersMap(User * usersArray, int recordsCount)
{
    map * umap = NewMap(recordsCount * 3);
    int i;
    for(i = 0; i < recordsCount; i++) {
        Map_Set(umap, usersArray[i].id, &usersArray[i]);
    }
    return umap;
}


```

## 8.2 Build.h

```
#ifndef Build_h
#define Build_h
#include "Data.h"
#include "map.h"

User * CreateUsersArray(char ** userIDs, char ** userNames, int recordsCount);

map * CreateUsersMap(User * usersArray, int recordsCount);



#endif
```

## 8.3 Connection.c

```

```

```c
#include "Connection.h"
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>
#include "Util.h"
#include "Log.h"
#include "Data.h"
#include "File.h"

ClientShared shared;

ClientShared * InitializeShared(map * users_map, size_t send_buffer_size, size_t receive_buffer_size)
{
    shared.users = users_map;
    shared.dirty = 0;
    shared.shutting_down = 0;
    shared.send_buffer_size = send_buffer_size;
    shared.receive_buffer_size = receive_buffer_size;
    pthread_mutex_init(&(shared.mutex), NULL);
    return &shared;
}

void * StartUpdateThread(void * parameter)
{
    while(shared.shutting_down == 0) {
        if(shared.dirty) {
            pthread_mutex_lock(&(shared.mutex));
            shared.dirty = 0;
            FILE * reg_file = CreateOrOpenFileVerbose(REGISTERED_FILE, NULL);
            if(reg_file != NULL) {
                UpdateRegisteredFileFromUsersMap(reg_file, shared.users);
                fclose(reg_file);
            } else {
                LogfError("FAILED TO OPEN REGISTERED FILE - NO DATA WILL BE UPDATED");
                shared.dirty = 1;
            }
            pthread_mutex_unlock(&(shared.mutex));
        }
        sleep(1);

    }
    return NULL;
}

void * StartConnectionThread(void * p_connection)
{
    Connection * connection = (Connection *) p_connection;
    connection->state = ClientState_ENTRY;
    connection->user = NULL;
    time(&(connection->time_connected));
    // allocate send and receive buffers.
    char * send_buffer = malloc(shared.send_buffer_size);
    char * receive_buffer = malloc(shared.receive_buffer_size);
    //int bytes_received;
    map_result result;

    // ask for their user ID initially, or disconnect them.
    strcpy(send_buffer, "<Message>Welcome. Please send your user ID.");
    MessageOrClose(send_buffer, receive_buffer, connection);
    if(connection->status == ConnectionStatus_ACTIVE) {
        result = Map_Get(shared.users, receive_buffer);
        if(!result.found)
        {
            printYellow("Unauthorized access attempt by %s with name '%s'.\n",
inet_ntoa(connection->address.sin_addr), receive_buffer);
            strcpy(send_buffer, "<Error>No such user");
            MessageAndClose(send_buffer, connection);
            // send a one-way message to the client
        } else {
            User * user = (User *) result.data;
            if(user->connected) {
                printYellow("User %s attempted to double connect from IP %s.\n", user->id,
inet_ntoa(connection->address.sin_addr));
                strcpy(send_buffer, "<Error>You are already connected.");
                MessageAndClose(send_buffer, connection);
                // send the other connected user an informative message?
            } else {
                connection->user = user;
                connection->user->connected = 1;
                strcpy(connection->user->ip, inet_ntoa(connection->address.sin_addr));
                if(connection->user->registered) {
                    connection->state = ClientState_REGISTERED;
                } else {
                    connection->state = ClientState_ACCESSING;
                }
            }
        }
    }

    if(connection->state == ClientState_ACCESSING && connection->status == ConnectionStatus_ACTIVE) {
        strcpy(send_buffer, "<Message>Say something, unregistered user!");
    } else if (connection->state == ClientState_REGISTERED && connection->status ==
ConnectionStatus_ACTIVE) {
        strcpy(send_buffer, "<Message>Say something, registered user!");
    }

    while(connection->status == ConnectionStatus_ACTIVE)
    {
        if(connection->state == ClientState_ACCESSING) {
            MessageOrClose(send_buffer, receive_buffer, connection);
            if (strcmp(receive_buffer, "help") == 0) {
                _help(connection, send_buffer);
            } else if (strcmp(receive_buffer, "exit") == 0) {
                strcpy(send_buffer, "<Message>Goodbye.");
                MessageAndClose(send_buffer, connection);
            } else if (strcmp(receive_buffer, "register") == 0) {
                _register(connection, send_buffer);
            } else {
                strcpy (send_buffer, "<Error>Invalid command, use 'help' for list of commands");
            }
        } else if(connection->state == ClientState_REGISTERED) {
            MessageOrClose(send_buffer, receive_buffer, connection);
            if (strcmp(receive_buffer, "help") == 0) {
                _help(connection, send_buffer);
            } else if (strcmp(receive_buffer, "exit") == 0) {
                strcpy(send_buffer, "<Message>Goodbye.");
                MessageAndClose(send_buffer, connection);
            } else if (strcmp(receive_buffer, "myinfo") == 0) {
                _myinfo(connection, send_buffer);
            } else if (strcmp(receive_buffer, "who") == 0) {
                _who(send_buffer);
            } else if(strcmp(receive_buffer, "random-gpa") == 0) {
                _rand_gpa(connection, send_buffer);
            } else if(strcmp(receive_buffer, "random-age") == 0) {
                _rand_age(connection, send_buffer);
            } else if(strcmp(receive_buffer, "advertisement") == 0){
                _advertisement(connection, send_buffer);
            }else {
                strcpy(send_buffer, "<Error>Invalid command, use 'help' for list of commands");
            }
            // call a function for processing this state.
        } else {
            printRed("Client entered invalid state. Disconnecting. \n");
            strcpy(send_buffer, "<Error>You entered an invalid state!");
            MessageAndClose(send_buffer, connection);
            connection->status = ConnectionStatus_CLOSING;
        }
    }

    if(connection->user != NULL) {
        connection->user->connected = 0;
        LogfInfo("User %s from ip %s disconnected.\n", connection->user->id, connection->user->ip);
    } else {
        LogfInfo("Ip %s disconnected.\n", inet_ntoa(connection->address.sin_addr));
    }


    free(send_buffer);
    free(receive_buffer);
    close(connection->socket);
    if(connection->user != NULL) {
        connection->user->connected = 0;
    }
    connection->status = ConnectionStatus_CLOSED;
    return NULL;
}


int MessageOrClose(char * send_buffer, char * receive_buffer, Connection * connection) {
    receive_buffer[0] = '\0';
    //memset(receive_buffer, 0, shared.receive_buffer_size);
    if(send(connection->socket, send_buffer, shared.send_buffer_size, 0) < 0) {
        printRed("Failed to send message to %s. Disconnecting.\n",
inet_ntoa(connection->address.sin_addr));
        perror("Error:");
        connection->status = ConnectionStatus_CLOSING;
        return 0;
    }
    int received_size = recv(connection->socket, receive_buffer, shared.receive_buffer_size, 0);
    if(received_size < 0) {
        printRed("Failed to receive message from %s. Disconnecting.\n",
inet_ntoa(connection->address.sin_addr));
        perror("Error: ");
        connection->status = ConnectionStatus_CLOSING;
        return 0;
    }
    if(received_size == 0 ) {
        printBlue("%s disconnected.\n", inet_ntoa(connection->address.sin_addr));
        connection->status = ConnectionStatus_CLOSING;
        return 0;
    }
    send_buffer[0] = '\0';
    // memset(send_buffer, 0, shared.send_buffer_size);
    return received_size;
}



void MessageAndClose(char * send_buffer, Connection * connection) {
    strcat(send_buffer, "<Disconnect>");
    send(connection->socket, send_buffer, shared.send_buffer_size, 0);
    connection->status = ConnectionStatus_CLOSING;
    if(connection -> user != NULL) {
        connection->user->connected = 0;
    }
}

void _help(Connection* connection, char* response) {
    if(connection->state != ClientState_REGISTERED) {
        strcpy(response, "<Message>help - get a list of available commands\n");
        strcat(response, "register - register your user\n");
        strcat(response, "exit - disconnect from the server");
    } else if(connection->state == ClientState_REGISTERED) {
        strcpy(response, "<Message>help- get a list of available commands\n");
        strcat(response, "exit - disconnect from the server\n");
        strcat(response, "who - get a list of online users\n");
        strcat(response, "random-gpa - set your gpa to a new random value\n");
        strcat(response, "random-age - set your age to a new random value\n");
        strcat(response, "advertisement - get a colorful advertisement\n");
        strcat(response, "myinfo - get info about yourself");
    }
}

int _register(Connection * connection, char* response) {

    if(connection->user->registered) {
        strcpy(response, "<Error>");
        strcat(response, connection->user->id);
        strcat(response, " is already registered.");

        LogfError("%s from ip %s has attempted to register a second time.\n", connection->user->id,
inet_ntoa(connection->address.sin_addr));
        return 0;
    }

    pthread_mutex_lock(&(shared.mutex));

    connection->user->registered = 1;

    connection->user->age = RandomInteger(18, 22);

    if(RandomFlag(.4)) {
        connection->user->gpa = 4.0;
    } else {
        connection->user->gpa = RandomFloat(2.5, 4);
    }

    connection->state = ClientState_REGISTERED;

    LogfDebug("%s has been registered.\n", connection->user->id);

    shared.dirty = 1;
    pthread_mutex_unlock(&(shared.mutex));

    strcpy(response, "<Message>You Have been registered ");
    strcat(response, connection->user->name);

    return 1;
}

int _myinfo(Connection* connection, char* response) {
    InitializeLogger(stdout, 0, 0, 0);

    if (!(connection->user->registered)) {
        strcpy(response, "<Error>");
        strcat(response, connection->user->id);
        strcat(response, " is not registered.");

        LogfError("%s from ip %s has attempted to view their information as an unregistered user.\n", connection->user->id, inet_ntoa(connection->address.sin_addr));

        return 1;
    }

    //Referenced snprintf from https://cplusplus.com/reference/cstdio/snprintf/
    snprintf(response, shared.send_buffer_size, "<User.Name>%s<User.Age>%d<User.GPA>%.2f<User.IP>%s", connection->user->name, connection->user->age, connection->user->gpa, inet_ntoa(connection->address.sin_addr));

    LogfInfo("%s viewed their information.\n", connection->user->id);

    return 0;
}

void _who(char * response) {
    int i;
    for(i = 0; i < RECORD_COUNT; i++) {
        map_result result = Map_Get(shared.users, accepted_userIDs[i]);
        if(result.found) {
            User* user = (User *) result.data;

            if(user->connected) {
                strcat(response, "<OnlineUser>");
                strcat(response, user->id);
            }
        }
    }
}

void _rand_gpa(Connection* connection, char* response) {
    char gpa_str[5];
    pthread_mutex_lock(&(shared.mutex));
    if(RandomFlag(.4)) {
        connection->user->gpa = 4.0;
    } else {
        connection->user->gpa = RandomFloat(2.2, 4.0);
    }
    shared.dirty = 1;
    pthread_mutex_unlock(&(shared.mutex));
    sprintf(gpa_str, "%.2f", connection->user->gpa);
    strcat(response, "<User.GPA>");
    strcat(response, gpa_str);
}

void _rand_age(Connection* connection, char * response) {
    char age_str[5];
    pthread_mutex_lock(&(shared.mutex));
    connection->user->age = RandomInteger(18, 22);
    shared.dirty = 1;
    pthread_mutex_unlock(&(shared.mutex));

    sprintf(age_str, "%d", connection->user->age);
    strcat(response, "<User.Age>");
    strcat(response, age_str);
}

void _advertisement(Connection * connection, char * response) {
    char filename[FILENAME_MAX];

    GetRandomFileNameFromDir(ADS_DIR, filename);

    char* filepath = malloc(FILENAME_MAX + sizeof(ADS_DIR));
    strcpy(filepath, ADS_DIR);
    strcat(filepath, "/");
    strcat(filepath, filename);

    strcat(response, "<Message>");
    CatFileToBuffer(filepath, response, shared.send_buffer_size);
}
```

## 8.4 Connection.h

```
#ifndef Connection_h
#define Connection_h
#include <netinet/in.h>
#include <pthread.h>
#include <time.h>
#include <Data.h>
#include "map.h"

typedef struct {
    map * users;
    pthread_mutex_t mutex;
    short dirty;
    short shutting_down;

    size_t send_buffer_size;
    size_t receive_buffer_size;
} ClientShared;

#define ClientState_ENTRY 1
#define ClientState_ACCESSING 2
#define ClientState_REGISTERED 3
typedef short ClientState;

#define ConnectionStatus_CLOSED 0
#define ConnectionStatus_ACTIVE 1
#define ConnectionStatus_CLOSING 2
typedef short ConnectionState;

typedef struct {
    ConnectionState status;
    int socket;
    struct sockaddr_in address;
    socklen_t address_length;
    pthread_t thread_id;
    time_t time_connected;
    ClientState state;
    User * user;

} Connection;

ClientShared * InitializeShared(map * users_map, size_t send_buffer_size, size_t receive_buffer_size);

void * StartConnectionThread(void * connection);

int MessageOrClose(char * send_buffer, char * receive_buffer, Connection * connection);

void MessageAndClose(char * send_buffer, Connection * connection);

void * StartUpdateThread(void * parameter);

int _register(Connection * connection, char* response);

void _help(Connection* connection, char* response);

int _myinfo(Connection* connection, char* response);

void _who(char* response);

void _rand_age(Connection* connection, char* response);

void _rand_gpa(Connection* connection, char* response);

void _advertisement(Connection * connection, char * response);
#endif
```

## 8.5 Data.c

```

#include "Data.h"

char * accepted_userIDs[] = {
    "chen",
    "bea1389",
    "bol4559",
    "cal6258",
    "kre5277",
    "lon1150",
    "mas9309",
    "mes08346",
    "mil7233",
    "nef9476",
    "nov7488",
    "pan9725",
    "rac3146",
    "rub4133",
    "shr5683",
    "vay3083",
    "yos2327"};

char * userFullNames[] = {
    "Weifeng Chen",
    "Christian Beatty",
    "Emily Bolles",
    "Cameron Calhoun",
    "Ty Kress",
    "Cody Long",
    "Caleb Massey",
    "Christian Messmer",
    "Karl Miller",
    "Jeremiah Neff",
    "Kaitlyn Novacek",
    "Joshua Panaro",
    "Caleb Rachocki",
    "Caleb Ruby",
    "Paul Shriner",
    "Alan Vayansky",
    "Assefa Ayalew Yoseph"};

```

## 8.6 Data.h

```
#ifndef Data_h
#define Data_h
#define RECORD_COUNT 17
#define ID_MAX_LENGTH 9
#define NAME_MAX_LENGTH 21

#define IP_LENGTH 16


extern char * accepted_userIDs[];

extern char * userFullNames[];

typedef struct
{
    char id[ID_MAX_LENGTH];
    char name[NAME_MAX_LENGTH];
    int age;
    float gpa;
    short connected;
    char ip[IP_LENGTH];
    long lastConnection;
    short registered;
} User;

#endif
```

## 8.7 File.c

```

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include "File.h"
#include "Data.h"
#include "Util.h"
#include "Log.h"

short FileStatus(char * filename) {
    int err = access(filename, F_OK);
    if(!err) {
        err = access(filename, F_OK | R_OK | W_OK);
        if(!err) {
            return 1;
        }
        return 2;
    }
    return 0;
}

FILE * CreateOrOpenFileVerbose(char * filename, char * defaultContents) {
    FILE * file = NULL;
    int status = FileStatus(filename);

    if(status == 2) {
        printRed("Error: %s exists but you do not have permission to access it.\n", filename);
        return NULL;
    }

    if(status == 0) {
        printf("Creating %s.\n", filename);
        file = fopen(filename, "w+");
    } else if(status == 1) {
        printf("Opening %s.\n", filename);
        file = fopen(filename, "r+");
    }


    if(file == NULL) {
        printf(COLOR_RED);
        if(status == 0) {
            printf("Failed to create %s.\n", filename);
            perror("Error: ");
        } else if(status == 1) {
            printf("Failed to open %s.\n", filename);
            perror("Error: ");
        } else {
            printf("Unknown error opening %s.", filename);
        }
        printf(COLOR_RESET);
        return NULL;
    }

    if(status == 0) {
        printGreen("Created %s.\n", filename);
        if(defaultContents != NULL) {
            fpos_t start_pos;
            fgetpos(file, &start_pos);
            fprintf(file, defaultContents, 0);
            fsetpos(file, &start_pos);
        }
    } else if(status == 1) {
        printGreen("Opened %s.\n", filename);
    }

    return file;
}


int ReadRegisteredFileIntoUsersMap(FILE * reg_file, map * users_map) {

    char userID[ID_MAX_LENGTH];
    int user_age;
    float user_gpa;
    char userLastIP[IP_LENGTH];
    long lastConnection;

    int scan_items;
    int line = 1;

    while( (scan_items = fscanf(reg_file, "%s\t%d\t%f\t%s\t%ld", userID, &user_age, &user_gpa,
userLastIP, &lastConnection)) == 5) {
        map_result result = Map_Get(users_map, userID);
        if(result.found == 0) {
            printYellow("Couldn't find user %s. Continuing read.\n", userID);
            continue;
        }
        User * user = (User*)result.data;
        user->age = user_age;
        user->gpa = user_gpa;
        strcpy(user->ip, userLastIP);
        user->lastConnection = lastConnection;
        user->registered = 1;
        line++;
    }

    if(scan_items != EOF) {
        printRed("Error scanning registered file on line %d. Expected 5 items but had %d.\n", line,
scan_items);
        return 1;
    }
    return 0;
}

void UpdateRegisteredFileFromUsersMap(FILE * reg_file, map * users_map) {
    int i;
    for(i = 0; i < RECORD_COUNT; i++) {
        map_result result = Map_Get(users_map, accepted_userIDs[i]);
        if(!result.found) {
            LogfError("User %s was not found in users map.", accepted_userIDs[i]);
            continue;
        }

```

```c
        User * user = (User *) result.data;
        if(user->registered) {
            fprintf(reg_file, "%s\t%d\t%f\t%s\t%ld", user->id, user->age, user->gpa, user->ip,
user->lastConnection);
        }
    }
}

int NumberOfFilesInDirectory(char* dir_name) {
    int count = 0;

    DIR * dirp;
    struct dirent * entry;

    dirp = opendir(dir_name);

    while((entry = readdir(dirp)) != NULL) {
        if(entry->d_type == DT_REG) {
            count++;
        }
    }

    closedir(dirp);

    return count;
}

void GetRandomFileNameFromDir(char * dir_name, char* file_name) {
    int file = RandomInteger(0, NumberOfFilesInDirectory(dir_name) - 1);

    DIR* dirp = opendir(dir_name);
    struct dirent * entry;

    while(file >= 0 && ((entry = readdir(dirp)) != NULL)) {
        if(entry->d_type == DT_REG) {
            file--;
        }
    }

    if(entry != NULL) {
        strcpy(file_name, entry->d_name);
    }

    closedir(dirp);
}

int ReadSettingsFileIntoSettingsMap(FILE * settings_file, map * settings_map) {
    char key_read[100];
    char value_read[100];

    int scan_items;
    int line = 1;

    while( (scan_items = fscanf(settings_file, " %s = %s ", key_read, value_read)) == 2) {
        char * key_alloc = malloc( (strlen(key_read)+1) * sizeof(char));
        memset(key_alloc, 0, strlen(key_read)+1);
        strcpy(key_alloc, key_read);
        char * val_alloc = malloc( (strlen(value_read)+1) * sizeof(char));
        memset(val_alloc, 0, strlen(value_read)+1);
        strcpy(val_alloc, value_read);
        Map_Set(settings_map, key_alloc, val_alloc);
        line++;
    }

    if(scan_items != EOF) {
        printRed("Error scanning settings file on line %d. Expected 2 items but had %d.\n", line,
scan_items);
        return 1;
    }
    return 0;
}

void CatFileToBuffer(char* file_name, char* buffer, size_t buffer_size) {
    if(FileStatus(file_name)) {
        FILE* file = CreateOrOpenFileVerbose(file_name, NULL);
        char* temp = malloc(buffer_size);

        while(fgets(temp, buffer_size - strlen(buffer), file) && buffer_size - strlen(buffer) > 1) {
            strcat(buffer, temp);
        }

        free(temp);
    }
}

int CreateLockfile()
{
    FILE * file = fopen(LOCKFILE, "w");
    if(file == NULL) {
        return 0;
    }
    fprintf(file, "0 %d", getpid());
    fclose(file);
    return 1;
}

int DeleteLockfile()
{
    return remove(LOCKFILE);
}

```

## 8.8 File.h

```
#ifndef Files_h
#define Files_h
#include <stdio.h>
#include "map.h"

// ~~~~ Macros ~~~~ //

#define LOCKFILE "/tmp/lab5.lock"

#define REGISTERED_FILE "registered.txt"

#define SERVER_SETTINGS_FILE "server-settings.txt"

#define ADS_DIR "ads"

// ~~~~~ General File Functions ~~~~~ //

short FileStatus(char * filename);

FILE * CreateOrOpenFileVerbose(char * filename, char * defaultContents);

int ReadRegisteredFileIntoUsersMap(FILE * reg_file, map * users_map);


int ReadSettingsFileIntoSettingsMap(FILE * settings_file, map * settings_map);

void UpdateRegisteredFileFromUsersMap(FILE * reg_file, map * users_map);

void GetRandomFileNameFromDir(char* dir_name, char* file_name);

int NumberOfFilesInDirectory(char* dir_name);

void CatFileToBuffer(char* file_name, char* buffer, size_t buffer_size);

/***
    Creates a lockfile.
    @warning This should only be called by a running server process when a lockfile does not already exist.
    @returns 1 on success, otherwise 0.
*/
int CreateLockfile();

/***
    Deletes a lockfile.
    @returns 1 on success, otherwise 0.
*/
int DeleteLockfile();

#endif
```

## 8.9 Log.c

```

#include "Log.h"
#include <stdarg.h>

LogSettings L;

void InitializeLogger(FILE* _printStream, char printLevel, char logLevel, char printAllToStdOut) {
    L.ostream = _printStream;
    L.printLevel = printLevel;
    L.logLevel = logLevel;
```

```
    L.printAllToStdOut = printAllToStdOut;
}

void _logf(int level, const char * format, va_list argptr) {

    // TODO add logic to include timestamp to the
    // Stamp level to log
    // if printing to standard out is used print with color based on level
    if (level >= L.printLevel || L.printAllToStdOut != 0) {
        vprintf(format, argptr);
    } else if (level >= L.logLevel) {
        vfprintf(L.ostream, format, argptr);
    }
}

void LogfFatal(const char * format, ...) {
    va_list argptr;
    va_start(argptr, format);
    _logf(FATAL, format, argptr);
    va_end(argptr);
}

void LogfError(const char * format, ...) {
    va_list argptr;
    va_start(argptr, format);
    _logf(ERROR, format, argptr);
    va_end(argptr);
}

void LogfWarning(const char * format, ...) {
    va_list argptr;
    va_start(argptr, format);
    _logf(WARNING, format, argptr);
    va_end(argptr);
}

void LogfInfo(const char * format, ...) {
    va_list argptr;
    va_start(argptr, format);
    _logf(INFO, format, argptr);
    va_end(argptr);
}

void LogfDebug(const char * format, ...) {
    va_list argptr;
    va_start(argptr, format);
    _logf(DEBUG, format, argptr);
    va_end(argptr);
}

void LogfTrace(const char * format, ...) {
    va_list argptr;
    va_start(argptr, format);
    _logf(TRACE, format, argptr);
    va_end(argptr);
}
```

## 8.10 Log.h

```
#ifndef Log_h
#define Log_h

#include <stdio.h>

#define TRACE 0
#define DEBUG 1
#define INFO 2
#define WARNING 3
#define ERROR 4
#define FATAL 5

typedef struct
{
    FILE * ostream;
    char printLevel;
    char logLevel;
    char printAllToStdOut;

} LogSettings;

//set up functions that print to stdout no matter if printtostdout is set to true

void LogfFatal(const char * format, ...);

void LogfError(const char * format, ...);

void LogfWarning(const char * format, ...);

void LogfInfo(const char * format, ...);

void LogfDebug(const char* format, ...);

void LogfTrace(const char * format, ...);

void InitializeLogger(FILE* _printStream, char printLevel, char logLevel, char printAllToStdOut);


#endif
```

## 8.11 main.c

```
#include <stdio.h>
#include "Util.h"
#include "Process.h"
#include <string.h>


int main(int argc, char **argv) {

    if(argc <= 1 )
    {
        RunCommand();
    }
    else if (strcmp(argv[1], "headless") == 0)
    {
        RunHeadless(argv[0]);
    }
    else if (strcmp(argv[1], "stop") == 0)
    {
        StopCommand();
    }
    else
    {
        RunCommand();
    }
    return 0;
}
```

## 8.12 map.c

```

#include "stdlib.h"
#include "string.h"
#include "map.h"
#include "math.h"

int hash_log2(int num_to_log)
{
    int t = 1;
    int i = 0;
    do
    {
        num_to_log = num_to_log & ~t;
        t = t « 1;
        i++;
    } while (num_to_log > 0);
    return i;
}

int hash_upperLimit(int bitsize)
{
    return 1 « bitsize;
}

int char_ratio = (int)(sizeof(int) / sizeof(char));

int hash_string(int hash_table_size, char *string, int strlen)
{
    int i, hash = 2166136261;
    for (i = 0; i < strlen; i += 1)
    {
        hash *= 16777619;
```

```c
        hash ^= string[i];
    }
    if (hash < 0)
    {
        hash *= -1;
    }
    return hash % hash_table_size;
}

map *NewMap(int capacity)
{
    int log2 = hash_log2(capacity);
    int capac = hash_upperLimit(log2);
    int sz = sizeof(struct _map_bucket) * capac;
    struct _map_bucket *buckets = malloc(sz);
    memset(buckets, 0, sz);
    int i;
    for (i = 0; i < capac; i++)
    {
        buckets[i] = (struct _map_bucket){NULL, NULL, NULL};
    }
    map newm = (map){capac, buckets};
    map *map_p = malloc(sizeof(map));
    *map_p = newm;
    return map_p;
}

void _bucket_insert(struct _map_bucket *bucket, char *key, void *value)
{
    struct _map_bucket *check = bucket;
    while (check->key != NULL)
    {
        if (strcmp(check->key, key) == 0)
        {
            check->data = value;
            return;
        }
        if (check->next == NULL)
        {
            check->next = malloc(sizeof(struct _map_bucket));
            *(check->next) = (struct _map_bucket){NULL, NULL, NULL};
        }
        check = check->next;
    }
    check->key = key;
    check->data = value;
}

void Map_Set(map *a_map, char *key, void *value)
{
    int keyl = (int)strlen(key);
    int hash = hash_string(a_map->size, key, keyl);
    _bucket_insert(&(a_map->buckets[hash]), key, value);
}
void _bucket_get(struct _map_bucket *bucket, char *key, map_result *result)
{
    struct _map_bucket *check = bucket;
    while (check->key != NULL)
    {
        if (strcmp(check->key, key) == 0)
        {
            result->found = 1;
            result->data = check->data;
            return;
        }
        else if (check->next != NULL)
        {
            check = check->next;
        }
        else
        {
            result->found = 0;
            break;
        }
    }
}

map_result Map_Get(map *a_map, char *key)
{
    map_result res = (map_result){0, NULL};
    int keyl = (int)strlen(key);
    int hash = hash_string(a_map->size, key, keyl);
    _bucket_get(&(a_map->buckets[hash]), key, &res);
    return res;
}

void _bucket_delete(struct _map_bucket *bucket, char *key, short free_it, map_result *result)
```

```
{
    struct _map_bucket *last = bucket;
    struct _map_bucket *next = bucket->next;
    while (next != NULL)
    {
        if (strcmp(next->key, key) == 0)
        {
            result->found = 1;
            result->data = next->data;
            if (free_it)
            {
                free(next->data);
                result->data = NULL;
            }
            last->next = next->next;
            free(next);
        }
        else
        {
            last = next;
            next = next->next;
        }
    }
}

map_result Map_Delete(map *a_map, char *key, short free_it)
{
    map_result res = (map_result){0, NULL};
    int keyl = (int)strlen(key);
    int hash = hash_string(a_map->size, key, keyl);

    struct _map_bucket top = a_map->buckets[hash];
    if (top.key == NULL)
    {
        return res;
    }
    if (strcmp(top.key, key) == 0)
    {
        res.found = 1;
        res.data = top.data;
        if (free_it)
        {
            free(top.data);
            res.data = NULL;
        }
        if (top.next != NULL)
        {
            a_map->buckets[hash] = *(top.next);
            free(top.next);
        }
        else
        {
            a_map->buckets[hash] = (struct _map_bucket){NULL, NULL, NULL};
        }
        return res;
    }
    if (top.next == NULL)
    {
        return res;
    }
    _bucket_delete(&(a_map->buckets[hash]), key, free_it, &res);

    return res;
}
```

## 8.13 map.h

```
#ifndef map_h
#define map_h

// -----------------------------
//         Hashing Math
// -----------------------------

int hash_log2(int number_to_log);

int hash_string(int hash_table_capacity, char *string, int strlen);

int hash_upperLimit(int bitsize);

// -----------------------------------
//         General Map Operations
// -----------------------------------
```

```
struct _map_bucket
{
    char *key;
    void *data;
    struct _map_bucket *next;
};

typedef struct
{
    int size;
    struct _map_bucket *buckets;
} map;

typedef struct
{

    short found;
    void *data;
} map_result;

map *NewMap(int capacity);

void Map_Set(map *a_map, char *key, void *value);

map_result Map_Get(map *a_map, char *key);

map_result Map_Delete(map *a_map, char *key, short free_it);

#endif
```

## 8.14 Process.c

```

#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include "Data.h"
#include "Build.h"
#include "map.h"
#include "File.h"
#include "Util.h"
#include "Server.h"
#include "Log.h"
#include "Connection.h"

User * users_array;
map * users_map;
map * settings_map;
char * default_settings = "port                = 3000\n"
                          "send_buffer_size    = 1024\n"
                          "receive_buffer_size = 1024\n"
                          "backlog             = 10\n"
                          "max_connections     = 20\n"
                          "log_file            = log.txt\n"
                          "log_level           = 1\n"
                          "log_to_console      = true";

int active_clients;

int _initializeLogger() {
    //char* fileName = "log.txt";
    int printLevel, LogLevel;// printAlltoStdOut;
    map_result result = Map_Get(settings_map, "log_file");
    if(!result.found) {
        printYellow("No output file found. Defaulting to 'log.txt'\n");
    } else {
        //fileName = result.data;
    }

    result = Map_Get(settings_map, "print_level");
    if(!result.found) {
        printYellow("No print_level found, defaulting to 3\n");
        printLevel = 3;
    } else {
        printLevel = atoi(result.data);
        if(printLevel < 0 || printLevel > 5) {
            printYellow("Invalid print_level of %d, defaulting to 3\n", printLevel);
            printLevel = 3;
        }
    }

    result = Map_Get(settings_map, "log_level");
    if(!result.found) {
        printYellow("No log_level found, defaulting to 3\n");
        LogLevel = 3;
    } else {
        LogLevel = atoi(result.data);
        if(LogLevel < 0 || LogLevel > 5) {
            printYellow("Invalid log_level of %d, defaulting to 3\n", LogLevel);
            LogLevel = 3;
        }
    }

    result = Map_Get (settings_map, "log_to_console");
    if(!result.found) {
        printYellow("No log_to_console found, defaulting to true\n");
        //printAlltoStdOut = 1;
    } else {
        if(strcmp(result.data, "true") == 0) {
            //printAlltoStdOut = 1;
        } else if(strcmp(result.data, "false") == 0) {
            //printAlltoStdOut = 0;
        } else {
            printYellow("invalid data in log_to_console, defaulting to true\n");
            //printAlltoStdOut = 1;
        }
    }
    return 1;
}

int Initialize() {

    // Create the data structures on the heap.
    printf("Initializing User data structures.\n");
    users_array = CreateUsersArray(accepted_userIDs, userFullNames, RECORD_COUNT);
    users_map = CreateUsersMap(users_array, RECORD_COUNT);
    active_clients = 0;
    printGreen("User data structures initialized.\n");

    // Create the registered file that tracks registered users.
    printf("Checking for registered file.\n");
    FILE * reg_file = CreateOrOpenFileVerbose(REGISTERED_FILE, NULL);
    if(reg_file == NULL) {
        printRed("Initialization failed during accessing of file: %s.\n", REGISTERED_FILE);
        return 0;
    }

    // Update the User’s map with with the data from the registered file.
    printf("Reading registered file.\n");
    int read_error = ReadRegisteredFileIntoUsersMap(reg_file, users_map);
    fclose(reg_file);
    if(read_error) {
        printRed("Initialization failed during reading of file: %s.\n", REGISTERED_FILE);
        return 0;
    }
    printGreen("Loaded %s into users map.\n", REGISTERED_FILE);

    printf("Reading settings file.\n");
    settings_map = NewMap(50);
    FILE * settings_file = CreateOrOpenFileVerbose(SERVER_SETTINGS_FILE, default_settings);
    if(settings_file == NULL) {
        printRed("Initialization failed during accessing of file: %s.\n", SERVER_SETTINGS_FILE);
        return 0;
    }
    int settings_read_err = ReadSettingsFileIntoSettingsMap(settings_file, settings_map);
    if(settings_read_err) {
        printRed("Initialization failed while reading settings file %s. Correct this file or delete it
so a default can be generated.\n", SERVER_SETTINGS_FILE);
        return 0;
    }
    fclose(settings_file);
    printGreen("Read %s.\n", SERVER_SETTINGS_FILE);

    printf("Initializing logger.\n");
    int logger_initialized = _initializeLogger();
    if(!logger_initialized) {
        printRed("Failed to initalize logger.\n");
    }


    printf("Initializing server.\n");
    int server_initialized = InitializeServer(settings_map);
    if(!server_initialized) {
        printRed("Failed to initialize server.\n");
        return 0;
    }

```

```c

    return 1;
}

void SignalHandle(int signo) {
    if(signo == SIGINT || signo == SIGTERM) {
        printYellow("Received signal. Shutting down server.\n");
        DeleteLockfile();
        exit(0);
    }

}

int RunCommand() {
    if (FileStatus(LOCKFILE) > 0)
    {
        printf("Server process already running.\n");
        return 0;
    }
    signal(SIGTERM, SignalHandle);
    signal(SIGINT, SignalHandle);
    int lockfile_success = CreateLockfile();
    if(!lockfile_success) {
        printRed("Failed to create Lockfile! Server cannot start.");
        return 0;
    }
    int init_success = Initialize();
    if(!init_success) {
        printRed("Could not start the server due to failed initialization.\n");
        return 0;
    }
    printf("Running server.\n");
    int server_success = StartServer(users_map);
    if(!server_success) {
        printRed("There was a problem running the server.\n");
        return 0;
    }
    int delete_lockfile_success = DeleteLockfile();
    if(!delete_lockfile_success) {
        printRed("There was a problem deleting the Lockfile.\n");
        return 0;
    }
    return 1;
}

void RunHeadless(char *processName) {
    if (FileStatus(LOCKFILE) > 0)
    {
        printf("Server process already running.\n");
        return;
    }
    char commandFront[] = " nohup ";
    char commandEnd[] = " & exit";
    size_t comm_length = strlen(commandFront) + strlen(commandEnd) + strlen(processName) + 1;
    char *commandFull = malloc(comm_length * sizeof(char));
    memset(commandFull, 0, comm_length * sizeof(char));
    strcpy(commandFull, commandFront);
    strcat(commandFull, processName);
    strcat(commandFull, commandEnd);

    printf("Executing: %s\n", commandFull);
    popen(commandFull, "we");
    printf("Server running headlessly.\n");
}

int TerminateExistingServer()
{
    FILE *file = fopen(LOCKFILE, "r");
    if (file == NULL)
    {
        perror("Error opening lockfile");
        return -1;
    }
    int need_rewrite;
    int pid = 0;
    fscanf(file, "%d %d", &need_rewrite, &pid);
    fclose(file);
    if (pid > 0)
    {
        return kill(pid, SIGTERM);
    }
    return -2;
}

void StopCommand() {
    printYellow("\nStopping server...\n");
    int err = TerminateExistingServer();
    if (err)
    {
        if (err == -1)
        {
            printRed("Server isn't running.\n");
        }
        else if (err == -2)
        {
            printRed("Lockfile did not contain a valid process id!\n");
        }
        else
        {
            printRed("Sending terminate signal failed!\n");
        }
    }
    else
    {
        printGreen("Server terminated.\n");
    }
}

void ResetCommand() {
    if(FileStatus(REGISTERED_FILE)) {
        fclose(fopen(REGISTERED_FILE, "w")); //empties the registered file
    }
}
```

## 8.15 Process.h

```
#ifndef Process_h
#define Process_h
/**
 * \defgroup Process
 * \brief This module holds functions that realize the primary business logic of the program.
 * \details When command line arguments are parsed by main, functions in this module are called.
 * @{
*/

/**
    Handles an interrupt or quit signal.

    Send server to shutdown mode, resulting in graceful deletion of lockfile.
*/
void SignalHandle(int signo);

/**
 * Performs initializing activities which must occur prior to a server loop starting.
 *
 * Will print errors if there are problems initializing.
 *
 * @returns 1 on success, otherwise 0.
*/
int Initialize();

/**
 * Runs the server.
 * @returns 1 on success, otherwise 0.
*/
int RunCommand();

/**
    Finds the process ID of a running server using the lockfile, and calls kill on it, sending a
SIGTERM.
    @returns -1 if the file doesn't exist, -2 if no valid process ID existed in the file, 1 if sending
the kill signal failed, or 0 on success.

*/
int TerminateExistingServer();

void StopCommand();

void RunHeadless(char *processName);

#endif
```

## 8.16 Server.c

```

#include <stdlib.h>
```

```
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <linux/net.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#include "Server.h"
#include "Connection.h"
#include "Util.h"
#include "map.h"

ServerProperties server;
Connection * connections;

// A private function just for reading the settings map into the server struct and printing warnings
as necessary.
void _readSettingsMapIntoServerStruct(map * server_settings) {
    map_result result = Map_Get(server_settings, "port");
    if(!result.found) {
        printYellow("No port setting found. Defaulting to 3000.\n");
        server.port = 3000;
    } else {
        int found_port = atoi(result.data);
        if(found_port <= 0) {
            printYellow("Invalid port setting: %s. Defaulting to 3000.\n", result.data);
            server.port = htons(3000);
        } else {
            server.port = htons(found_port);
        }
    }
    result = Map_Get(server_settings, "send_buffer_size");
    if(!result.found) {
        printYellow("No send_buffer_size setting found. Defaulting to 1024.\n");
        server.send_buffer_size = 1024;
    } else {
        int found_sb_size = atoi(result.data);
        if(found_sb_size <= 0) {
            printYellow("Invalid send_buffer_size setting: %s. Defaulting to 1024.\n", result.data);
            server.send_buffer_size = 1024 * sizeof(char);
        } else {
            server.send_buffer_size = found_sb_size * sizeof(char);
        }
    }
    result = Map_Get(server_settings, "receive_buffer_size");
    if(!result.found) {
        printYellow("No receive_buffer_size setting found. Defaulting to 1024.\n");
        server.send_buffer_size = 1024;
    } else {
        int found_rb_size = atoi(result.data);
        if(found_rb_size <= 0) {
            printYellow("Invalid receive_buffer_size setting: %s. Defaulting to 1024.\n",
result.data);
            server.receive_buffer_size = 1024 * sizeof(char);
        } else {
            server.receive_buffer_size = found_rb_size * sizeof(char);
        }
    }
    result = Map_Get(server_settings, "backlog");
    if(!result.found) {
        printYellow("No backlog setting found. Defaulting to 10.\n");
        server.backlog = 10;
    } else {
        int found_backlog = atoi(result.data);
        if(found_backlog <= 0) {
            printYellow("Invalid backlog setting: %s. Defaulting to 10.\n", result.data);
            server.backlog = 10;
        } else {
            server.backlog = found_backlog;
        }
    }
    result = Map_Get(server_settings, "max_connections");
    if(!result.found) {
        printYellow("No max_connections setting found. Defaulting to 20.\n");
        server.max_connections = 20;
    } else {
        int found_max_connections = atoi(result.data);
        if(found_max_connections <= 0) {
            printYellow("Invalid max_connections setting: %s. Defaulting to 20.\n", result.data);
            server.max_connections = 20;
        } else {
            server.max_connections = found_max_connections;
        }
    }
};
```

```

int InitializeServer(map * server_settings) {
    _readSettingsMapIntoServerStruct(server_settings);
    connections = malloc(server.max_connections * sizeof(Connection));
    int i;
    for(i=0; i < server.max_connections; i++) {
        connections[i].status = ConnectionStatus_CLOSED;
    }
    printGreen("Server initialized with %d max connections.\n", server.max_connections);
    return 1;
}

int StartServer(map * users_map) {
    int serverSocket = 0;
    struct sockaddr_in server_address;
    // Record the time the server started.
    time(&server.time_started);
    // Get a socket file pointer associated with ipv4 internet protocols that represents a two-way connection based byte stream.
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    server_address.sin_family = AF_INET;
    // Set the address to bind to all available interfaces.
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    // Set the port.
    server_address.sin_port = server.port;
    // Assign a name to the socket.
    int bind_error = bind(serverSocket, (struct sockaddr*)&server_address, sizeof(server_address));
    if(bind_error) {
        printRed("Error binding the server to port %d.\n", ntohs(server.port));
        perror("Bind Error:");
        return 0;
    }
    // Initialized a shared space that will be used across threads.
    ClientShared * shared = InitializeShared(users_map, server.send_buffer_size, server.receive_buffer_size);
    // The update thread is responsible for checking if there is 'dirty' data that should be saved to the registered user's file.
    pthread_t registered_update_thread;
    pthread_create(&registered_update_thread, NULL, StartUpdateThread, NULL);
    printBlue("Server listening on port: %d\n", ntohs(server.port));
    // begin listening according to the socket settings
    listen(serverSocket, server.backlog);
    while(!shared->shutting_down) {
        // Get an available connection.
        Connection * next_client = NextAvailableConnection();
        if(next_client == NULL) {
            printYellow("Server connections are maxxed.\n");
            sleep(1);
            continue;
        }
        // Accept a connection.
        next_client->address_length = sizeof(next_client->address);
        next_client->socket = accept(serverSocket, (struct sockaddr *)&(next_client->address), &(next_client->address_length));
        if(next_client->socket < 0)
        {
            printRed("Failed to accept() client!\n");
            sleep(1);
            continue;
        }
        printBlue("New client connection from IP: %s\n", inet_ntoa(next_client->address.sin_addr));
        next_client->status = ConnectionStatus_ACTIVE;
        // Start a thread to handle communication from that connection.
        pthread_create(&(next_client->thread_id), NULL, StartConnectionThread, next_client);
    }

    return 1;
}

Connection * NextAvailableConnection()
{
    int i;
    for(i = 0; i < server.max_connections; i++) {
        if(connections[i].status == ConnectionStatus_CLOSED)
        {
            return &(connections[i]);
        }
    }
    return NULL;
}


```

## 8.17 Server.h

```c
#ifndef Server_h
#define Server_h
#include <stdint.h>
#include <time.h>
#include "map.h"
#include "Connection.h"

typedef struct {
    uint16_t port;
    size_t send_buffer_size;
    size_t receive_buffer_size;
    int backlog;
    int active_connections;
    int max_connections;
    time_t time_started;
} ServerProperties;

int InitializeServer();

int StartServer(map * users_map);

Connection * NextAvailableConnection();

#endif
```

## 8.18 Util.c

```c

#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include "Util.h"

void printRed(const char * format, ...) {
    printf(COLOR_RED);
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    printf(COLOR_RESET);
}

void printGreen(const char * format, ...) {
    printf(COLOR_GREEN);
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    printf(COLOR_RESET);
}

void printYellow(const char * format, ...) {
    printf(COLOR_YELLOW);
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    printf(COLOR_RESET);
}

void printBlue(const char * format, ...) {
    printf(COLOR_BLUE);
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    printf(COLOR_RESET);
}

int RandomInteger(int min, int max)
{
    int r_add = rand() % (max - min + 1);
    return r_add + min;
}

float RandomFloat(float min, float max)
{
    float dif = max - min;
    int rand_int = rand() % (int)(dif * 10000);
    return min + (float)rand_int / 10000.0;
}

short RandomFlag(float percentage_chance)
{
    float random_value = (float)rand() / RAND_MAX;
    if (random_value < percentage_chance)
    {
        return 1;
    }
    return 0;
}

```

## 8.19 Util.h

```
#ifndef Util_h
#define Util_h
#define COLOR_RED "\e[38;2;255;75;75m"
#define COLOR_GREEN "\e[38;2;0;240;0m"
#define COLOR_YELLOW "\e[38;2;255;255;0m"
#define COLOR_BLUE "\e[38;2;0;240;240m"
#define COLOR_RESET "\e[0m"

void printRed(const char * format, ...);

void printGreen(const char * format, ...);

void printYellow(const char * format, ...);

void printBlue(const char * format, ...);


int RandomInteger(int min, int max);

float RandomFloat(float min, float max);

short RandomFlag(float percentage_chance);

#endif
```

# Index