ecet4640-lab4

1.0

Generated by Doxygen 1.9.6

Chapter 1

ecet4640-lab4

1.1 Intro

This program reads user information using the `who` and `ac -p` commands and publishes that information as an array of Student structures to shared virtual memory for client processes to read. It updates every second.

The main.c page is a good starting point for following the program control flow.

1.2 Contributions

- On 9/14 all group members collaborated on VSCode LiveShare to implement the program skeleton, including the testing framework.
- On 9/17 Karl started the client and worked on memshare.
- On 9/18 all group members collaborated to start Build, Data, and memShare.
- On 9/20 Christian worked on functions to randomize and create the initial data and started the processing of the who pipe.
- On 9/21 all group members collaborated to fix up memshare and read files.
- On 9/22 Karl added the map and worked to populate data structures from files.
- On 9/23 Karl finished the reading who process and the control flow.
- On 9/24 Paul worked on handling command line arguments.
- On 9/25 Karl implemented the ac -p cumulative times and headless running.
- On 9/26 Karl and Christian started documentation.
- On 9/27 all group members collaborated to finish the documentation.

1.3 Overview

The first time the program runs, it generates files containing static user data and the cumulative login times for each user. As the server runs, it will recheck the result of 'who' and 'ac -p' to update the student's cumulative login times, determine which students are actively logged in, and what time they last logged in. This information is stored in a read-shared memory segment so clients can access it using the shared memory key. If necessary, it also updates student information in the file.

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.

1.4 Arguments for program

Argument	Description	Calls
help	Prints usage of program.	HelpCommand()
reset	Resets and re-randomizes the static user data and restarts the cumulative time tracking.	ResetCommand()
stop	Stops an existing server process if it is running.	StopCommand()
headless	Runs the program headlessly in the background if it is not already running.	RunHeadless()
run	Runs the server in the current program if it is not already running.	RunCommand()

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 output the help for the server command. Executing `make server` will make the server binary without running it.

Second is for making the test client process with `make client`. This uses the files from `src/client`. The client process is not documented as it was not part of the program objective, and to avoid further documentation inflation.

Third 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 run` to run the server in the shell.
6. Press ctrl-c to stop the server.
7. Run `./server headless` to run the server headlessly using nohup.
8. Run `./server reset` to re-randomize the user data and reset the login times.
9. Run `./server stop` to shut down the server. (You may want to leave it running so clients can connect to it)

2.3 Screenshot of Compilation

```
[mil7233@draco1 ecet4640-lab4]$ make server
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -Isrc/client -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 -Isrc/client -c src/server/memShare.c -o bin/src/server/memShare.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -Isrc/client -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 -Isrc/client -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 -Isrc/client -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 -Isrc/client -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 -Isrc/client -c src/server/Build.c -o bin/src/server/Build.c.o
mkdir -p bin/src/server/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -Isrc/client -c src/server/Files.c -o bin/src/server/Files.c.o
mkdir -p bin/src/client/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -Isrc/client -c src/client/Print.c -o bin/src/client/Print.c.o
mkdir -p bin/src/client/
gcc  -Wall -Itests -Itests/lib -Isrc -Isrc/server -Isrc/client -c src/client/GetData.c -o bin/src/client/GetData.c.o
gcc  bin/src/server/util.c.o bin/src/server/memShare.c.o bin/src/server/Data.c.o bin/src/server/map.c.o bin/src/server/main.c.o bin/
/Files.c.o bin/src/client/Print.c.o bin/src/client/GetData.c.o -o server
[mil7233@draco1 ecet4640-lab4]$ ./server run

Running server.
static-user-data.txt does not exist. Creating.
static-user-cumulative-start.txt does not exist. Creating.
Student data retrieved from file.
Shared memory allocated.
Server started.
^CReceived shutdown signal.
Server shutting down.
Server terminated.
[mil7233@draco1 ecet4640-lab4]$ ./server headless
Executing:  nohup ./server run & exit
Server running headlessly.
[mil7233@draco1 ecet4640-lab4]$ nohup: appending output to 'nohup.out'

[mil7233@draco1 ecet4640-lab4]$ ./server run

Running server.

Server is already running. Run 'server stop' to shut it down first.
[mil7233@draco1 ecet4640-lab4]$ ./server stop

Stopping server...
Server terminated.
[mil7233@draco1 ecet4640-lab4]$
```

Figure 2.1 Compiling on draco1

2.4 Cleaning

There are two clean commands.

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

`make cleanf` will also remove the files generated on server initialization, such as the cumulative login file and user data file.

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

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 files with brief descriptions:

Chapter 6

Module Documentation

6.1 Build

Functions that populate data structures.

Functions

- void PopulateStudents (char **studentIDs, char **studentNames, int arsize)
- void BuildStudentMap (map *stmap, Student *studentArr, int studentArrLength)
- int PipeWhoToStudentMap (map *stmap)
- int ProcessWhoLine (map *stmap, char *whoLine, int whoLineLength)
- void SetAllStudentsInactive (Student *stud_arr, int arr_len)
- void WriteStudentsToMemory (void *mem_ptr, Student *stud_arr, int arr_len)
- int ReadInitialCumulative (map *time_map, char *filename)
- int PipeAcpToStudentMap (map *st_map)
- void ReadCumulativeFileLine (map *cum_map, char *acp_line)
- int ReadAcpPipeLine (map *stmap, char *acp_line)
- void CalculateCumulative (Student *stud_arr, int stud_arr_len, map *cum_map)

Variables

- Student * students
- short dirty = 1

6.1.1 Detailed Description

These functions perform actions that involve populating maps and arrays.

6.1.2 Function Documentation

6.1.2.1 PopulateStudents()

```
void PopulateStudents (
            char ** studentIDs,
            char ** studentNames,
            int arsize )
```

Allocate and populate the Students array with data.

Parameters

studentIDs	An array of student IDs.
studentNames	An array of student names.
arsize	The size of the array to allocate.

Warning

studentIDs and studentNames must both be arsize in length.

Definition at line 17 of file Build.c.

Here is the caller graph for this function:

6.1.2.2 BuildStudentMap()

```
void BuildStudentMap (
            map * stmap,
            Student * studentArr,
            int studentArrLength )
```

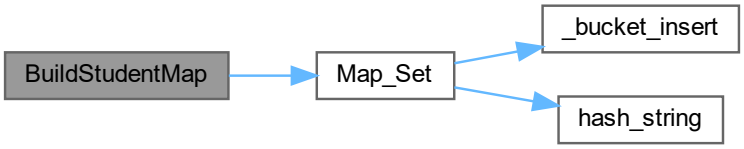
Given a student array, populates a student map, where the student IDs are the key, and the values are pointers to items in the array.

Parameters

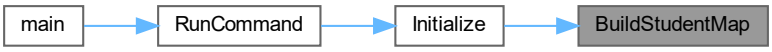
map	The map structure to populate.
studentArr	An array of student structures.
studentArrLength	The length of the students array.

Definition at line 28 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



6.1.2.3 PipeWhoToStudentMap()

```
int PipeWhoToStudentMap (
            map * stmap )
```

Executes the 'who' command by reading from a file pipe. Calls ProcessWhoLine for each line, to realize updates in the user data from the who command.

Parameters

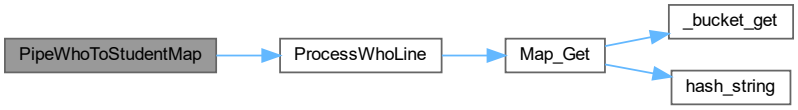
stmap	The student map.

Returns

0 if succesful, otherwise nonzero.

Definition at line 41 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.2.4 ProcessWhoLine()

```
int ProcessWhoLine (
            map * stmap,
            char * whoLine,
            int whoLineLength )
```

Processes a single line as read from the 'who' shell command. Uses that data to update the relevant student by retrieving them from the student map. Updates that students last login time. Also sets 'active' to 1 for the found student.

Attention

May set dirty to 1.

Parameters

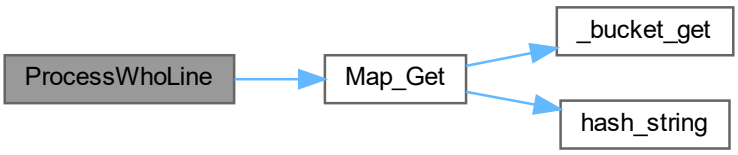
stmap	The student map.
whoLine	The line of text, such as returned from fgets
whoLineLength	The length of that text.

Returns

0 if success, -1 if the student was not found in the map.

Definition at line 61 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.2.5 SetAllStudentsInactive()

```
void SetAllStudentsInactive (
            Student * stud_arr,
            int arr_len )
```

Sets the 'active' property on all students in the students array to 0.

Parameters

stud_arr	The students array.
arr_len	The length of the students array.

Definition at line 109 of file Build.c.

Here is the caller graph for this function:

6.1.2.6 WriteStudentsToMemory()

```
void WriteStudentsToMemory (
            void * mem_ptr,
            Student * stud_arr,
            int arr_len )
```

Writes the students array to the location specified by mem_ptr (eg. the shared memory segment).

Parameters

mem_ptr	The address to write at.
stud_arr	The students array to write.
arr_len	The length of the students array.

Definition at line 118 of file Build.c.

Here is the caller graph for this function:

6.1.2.7 ReadInitialCumulative()

```
int ReadInitialCumulative (
            map * time_map,
            char * filename )
```

Populates the cumulative map by reading from the initial cumulative file. The map will be of the form [userID] -> long seconds

The map will contain users who we don't care about, but it doesn't matter.

Parameters

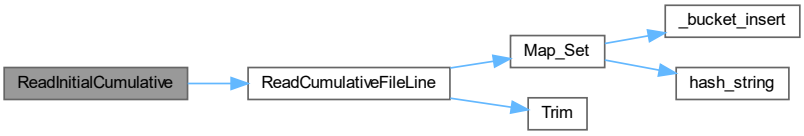
time_map	A map of cumulative times. Different from the students map.
filename	The filename where the initial cumulative times are located.

Returns

0 if success. -1 if it failed to find the file.

Definition at line 136 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.2.8 PipeAcpToStudentMap()

```
int PipeAcpToStudentMap (
            map * st_map )
```

Pipes ac -p, then calls ReadCumulativeLine to update the student map.

Note

After this runs, the student map cumulative will be their total login time in the system. This total time must be subtracted from the cumulative map time to find the time they have been logged in since the program started.

Parameters

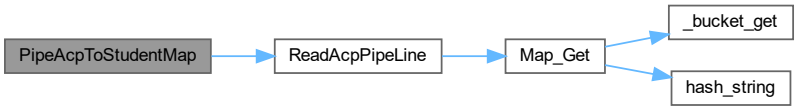
st_map	The students map.

Returns

0 on success. -1 if the pipe could not be opened. Otherwise an error from ReadAcpPipeLine().

Definition at line 153 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.2.9 ReadCumulativeFileLine()

```
void ReadCumulativeFileLine (
            map * time_map,
            char * acp_line )
```

Reads a single line from the initial cumulative file and updates the map so that userID maps to a long seconds value in the initial file.

Note

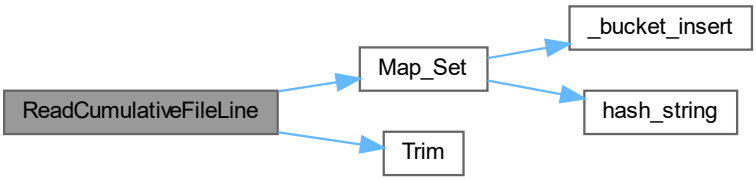
A line is structured like this: `mes08346 10.06` It finishes with a line starting with `total`; this line should be disregarded.

Parameters

time_map	The cumulative map.
acp_line	A single line from ac -p.

Definition at line 177 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.2.10 ReadAcpPipeLine()

```
int ReadAcpPipeLine (
            map * stmap,
            char * acp_line )
```

Reads a single line from the result of ac -p into the students map.

Parameters

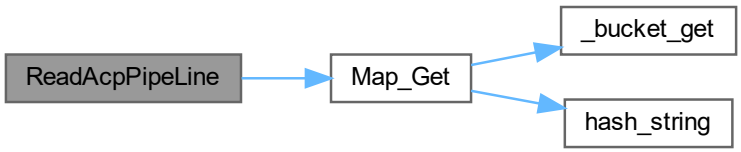
stmap	A map of students.
acp_line	A string representing 1 line result from ac -p.

Returns

-1 if acp_line is NULL or length is less than 1, otherwise 0.

Definition at line 193 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.2.11 CalculateCumulative()

```
void CalculateCumulative (
            Student * stud_arr,
            int stud_arr_len,
            map * time_map )
```

Calculates the cumulative time for each student by subtracting cum_map[studentID] from student.loginDuration.

Warning

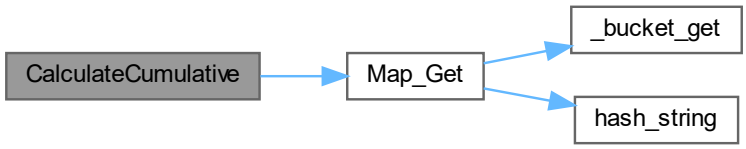
each student.loginDuration must have already been set to the total cumulative time logged in.

Parameters

stud_arr	The student's array.
arr_len	The length of students array.
time_map	A map mapping studentIds to their cumulative login time when the server was started.

Definition at line 217 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.1.3 Variable Documentation

6.1.3.1 students

```
Student* students
```

A pointer to the students array. It is heap allocated with malloc, when PopulateStudents() is called.

Note

This array and its length are passed around via parameters, to decouple as much as possible and enable simple testing and dummy data, even though it is globally available.

Definition at line 15 of file Build.c.

6.1.3.2 dirty

```
short dirty = 1
```

Set to '1' if there are changes that should be written to a file.

Definition at line 39 of file Build.c.

6.2 Data

Declarations of types and macros.

Data Structures

- struct Student

  The student data type.

Macros

- #define DATA_NUM_RECORDS 17
- #define DATA_ID_MAX_LENGTH 9
- #define DATA_NAME_MAX_LENGTH 21
- #define DATA_SIZE 56

Variables

- char * Data_IDs [DATA_NUM_RECORDS]
- char * Data_Names [DATA_NUM_RECORDS]
- char * Data_IDs [ ]
- char * Data_Names [ ]

6.2.1 Detailed Description

This module implements the data types required by the project specifications. The contents of this file should be shared with clients.

6.2.2 Macro Definition Documentation

6.2.2.1 DATA_NUM_RECORDS

```
#define DATA_NUM_RECORDS 17
```

The total count of records.

Definition at line 15 of file Data.h.

6.2.2.2 DATA_ID_MAX_LENGTH

```
#define DATA_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 20 of file Data.h.

6.2.2.3 DATA_NAME_MAX_LENGTH

```
#define DATA_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 25 of file Data.h.

6.2.2.4 DATA_SIZE

```
#define DATA_SIZE 56
```

The size of one student record; the result of sizeof(Student).

Definition at line 36 of file Data.h.

6.2.3 Variable Documentation

6.2.3.1 Data_IDs [1/2]

```
char* Data_IDs[DATA_NUM_RECORDS]
```

Initial value:

```
= {
    "chen",
    "beal389",
    "bol4559",
    "cal6258",
    "kre5277",
    "lon1150",
    "mas9309",
    "mes08346",
    "mil7233",
    "nef9476",
    "nov7488",
    "pan9725",
    "rac3146",
    "rub4133",
    "shr5683",
    "vay3083",
    "yos2327"}
```

Definition at line 7 of file Data.c.

6.2.3.2 Data_Names [1/2]

```
char* Data_Names[DATA_NUM_RECORDS]
```

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"}
```

Constant, all user's names.

Definition at line 26 of file Data.c.

6.2.3.3 Data_IDs [2/2]

```
char* Data_IDs[]  [extern]
```

Definition at line 7 of file Data.c.

6.2.3.4 Data_Names [2/2]

```
char* Data_Names[]  [extern]
```

Constant, all user's names.

Definition at line 26 of file Data.c.

6.3 Files

The Files module contains functions which operate on files.

Macros

- #define STATIC_USER_DATA_FILE "static-user-data.txt"
- #define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"
- #define LOCKFILE "/tmp/ecet-server.lock"

Functions

- short FileExists (char *file_name_to_check)

  Determines whether a file exists.
- int CreateInitialUserDataFile (char *file_name, char **id_list, int id_list_len)

  Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.
- int FillStudentMapFromFile (map *student_map, char *file_name, char **id_list, int id_list_len)

  Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.
- int WriteStudentArrayToFile (Student *students, int arr_len, char *file_name)

  Writes the student array to the file.
- int CreateInitialCumulativeFile (char *file_name)
- short DoesLockfileExist ()
- int CreateLockfile ()
- int DeleteLockfile ()

6.3.1 Detailed Description

Some program data needs to be stored in files, to preserve it in the case of early termination.

There are three files that are created if they don't exist when the program is first run.

- STATIC_USER_DATA_FILE contains a list of userIDs, ages, gpa, and last login time. Age and gpa are randomly generated on server start and when "reset" is run. The login time is updated when it changes as per the dirty flag.
- STATIC_USER_CUMULATIVE_FILE contains the results of 'ac -p' run when the server first starts. These values will be subtracted from later pipes of "ac -p" to determine the cumulative time since the server started.
- LOCKFILE contains a flag, 0 or 1, that indicates whether the STATIC_USER_DATA_FILE has been re-randomized and should be re-read. It contains the process ID of the running server process. It serves as an indicator to the process as to whether a server is already running and, when "close" is passed as a command line argument, which process to kill.

6.3.2 Macro Definition Documentation

6.3.2.1 STATIC_USER_DATA_FILE

```
#define STATIC_USER_DATA_FILE "static-user-data.txt"
```

File name for the text file that will store user data, namely, the age, gpa, and last login time.

Note

Each line contain in the created file contains:

(1) The ID from the students array, where the `line # - 1` == the index of the students array

(2) A tab character

(3) A random int between 18 and 22, for the age.

(4) A tab character

(5) A random float between 2.5 and 4.0, for the GPA.

(6) A tab character.

(7) A 0 (representing the last login time)

(8) A newline.

The order of entries in the file is the same as the order in the Data_IDs array from Data.c.

Definition at line 33 of file Files.h.

6.3.2.2 STATIC_USER_CUMULATIVE_FILE

```
#define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"
```

File name for the text file that will store the cumulative login time for each user at the point in time when it was created.

The values in this file are subtracted from the result of running 'ac -p' later to get the cumulative time each user was logged in since the server started.

Note

Each line contains the following.

(1) A user ID

(2) An integer representing the minutes the user has been logged in.

Definition at line 43 of file Files.h.

6.3.2.3 LOCKFILE

```
#define LOCKFILE "/tmp/ecet-server.lock"
```

The lockfile serves as a signal to subsequent processes as to whether or not server is already running.

Note

File contains the following

(1) a 1 or a 0 indicating whether the data has been reset and must be re-read

(2) an integer correcponding to the PID of the process so that server close can end that process

Definition at line 53 of file Files.h.

6.3.3 Function Documentation

6.3.3.1 FileExists()

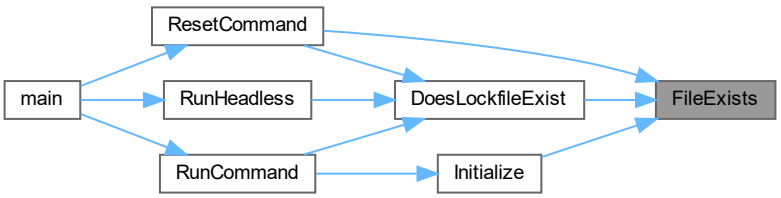
```
short FileExists (
            char * file_name_to_check )
```

Returns

1 if it exists. 0 if it does not.

Definition at line 12 of file Files.c.

Here is the caller graph for this function:



6.3.3.2 CreateInitialUserDataFile()

```
int CreateInitialUserDataFile (
            char * file_name,
            char ** id_list,
            int id_list_len )
```

Parameters

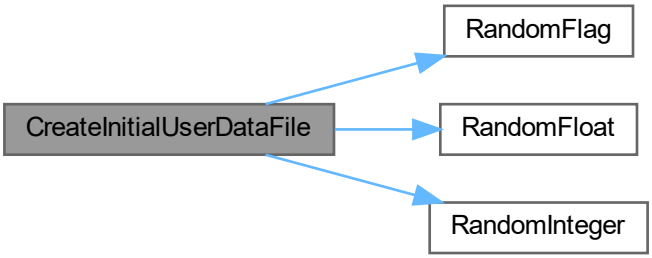
file_name	The file name to create.
id_list	An array containing the IDs. Eg. "Data_IDs" from Data.h
id_list_len	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

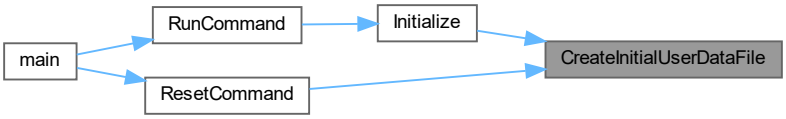
A 0 if the operation was succesful, otherwise nonzero.

Definition at line 27 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.3.3 FillStudentMapFromFile()

```
int FillStudentMapFromFile (
            map * student_map,
            char * file_name,
            char ** id_list,
            int id_list_len )
```

Parameters

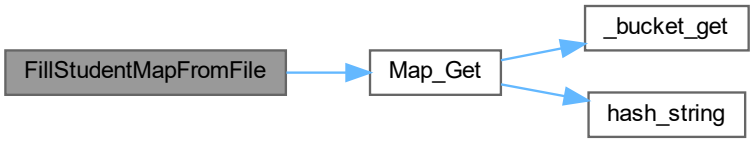
student_map	The map of student structs to be populated from the login.txt file
file_name	The name of the login.txt file.
id_list	An array containing the IDs. Eg. "Data_IDs" from Data.h
id_list_len	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

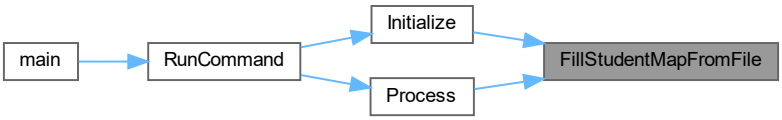
0 if succesful, 1 if there was an error.

Definition at line 53 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.3.4 WriteStudentArrayToFile()

```
int WriteStudentArrayToFile (
            Student * students,
            int arr_len,
            char * file_name )
```

Parameters

students	A pointer to the student array that will be read into the file.
arr_len	The length of the students array. e.g. DATA_NUM_RECORDS from Data.h.
file_name	The file name to write.

Returns

A 0 if the operation was succesful, otherwise a nonzero.

Definition at line 80 of file Files.c.

Here is the caller graph for this function:

6.3.3.5 CreateInitialCumulativeFile()

```
int CreateInitialCumulativeFile (
            char * file_name )
```

Creates the initial cumulative login time file.

It will hold the result of running 'ac -p'.

Parameters

file_name	The name of the file to created. EG STATIC_USER_CUMULATIVE_FILE

Warning

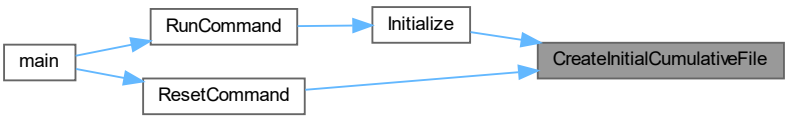
This file should already be validated to not exist.

Returns

0 if succesful, -1 if the file couldn't be opened, -2 if the pipe couldn't be opened, otherwise an error code.

Definition at line 96 of file Files.c.

Here is the caller graph for this function:



6.3.3.6 DoesLockfileExist()

```
short DoesLockfileExist ( )
```

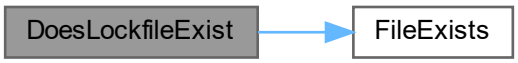
Determines if lockfile exists, which indicates that a server process is already running.

Returns

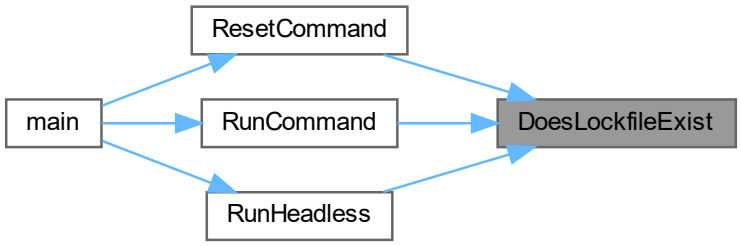
0 if lockfile does not exist, 1 if it does.

Definition at line 122 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.3.7 CreateLockfile()

```
int CreateLockfile ( )
```

Creates a lockfile.

Warning

This should only be called by a running server process when a lockfile does not already exist.

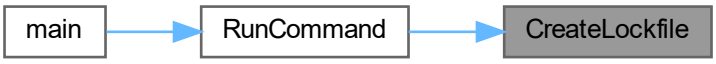
The lockfile will carry a 'data reset' signal and a process ID. CreateLockfile will write the current processes PID.

Returns

-1 if fopen failed, otherwise 0.

Definition at line 127 of file Files.c.

Here is the caller graph for this function:



6.3.3.8 DeleteLockfile()

```
int DeleteLockfile ( )
```

Deletes the lockfile.

Returns

0 on success, -1 on failure.

Definition at line 139 of file Files.c.

Here is the caller graph for this function:

6.4 Map

Functions that implement a hash map data structure.

Data Structures

- 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

- int hash_log2 (int number_to_log)

  Get's a log2 ceiling. Eg, hash_log2(5) == 3.
- int hash_upperLimit (int bitsize)

  This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.
- int hash_string (int hash_table_size, char ∗string, int strlen)
- map ∗ NewMap (int capacity)
- void _bucket_insert (struct _map_bucket ∗bucket, char ∗key, void ∗value)
- void Map_Set (map ∗a_map, char ∗key, void ∗value)

  Sets a value in the map.
- void _bucket_get (struct _map_bucket ∗bucket, char ∗key, map_result ∗result)
- 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.
- void _bucket_delete (struct _map_bucket ∗bucket, char ∗key, short free_it, map_result ∗result)
- 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.

Variables

- char ∗ key

  The key associated with this bucket.
- void ∗ data

  The data this bucket holds.
- struct _map_bucket ∗ next

  The next node in this linked list, or NULL if it is a leaf.
- int size

  The number of base buckets in this map.
- struct _map_bucket ∗ buckets

  The buckets for this map.
- short found

  1 if succesfully found. 0 if not found.
- void ∗ data

  The data linked with that key; indeterminate if found == 0.

6.4.1 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.

6.4.2 Function Documentation

6.4.2.1 hash_log2()

```
int hash_log2 (
            int number_to_log )
```

Parameters

number_to_log	The number to calculate the log of.

Returns

The log ceiling; eg, the lowest exponent to raise 2 with which would yield a number greater or equal to number_to_log.

Definition at line 10 of file map.c.

Here is the caller graph for this function:

6.4.2.2 hash_upperLimit()

```
int hash_upperLimit (
            int bitsize )
```

Parameters

bitsize	The number of bits to calculate the max from.

Returns

The max value that number of bits can hold.

Definition at line 23 of file map.c.

Here is the caller graph for this function:

6.4.2.3 hash_string()

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

Uses some clever, prime-number-multiplication, ORing, and bitwise operations to generate a number than, when modulused with the hash_table_size, will produce numbers ('buckets') of even distribution, to minimize the number of collisions. This function contains the meat of the hashing algorithm; it converts a key-string to an array index.

See also

http://isthe.com/chongo/tech/comp/fnv/

Parameters

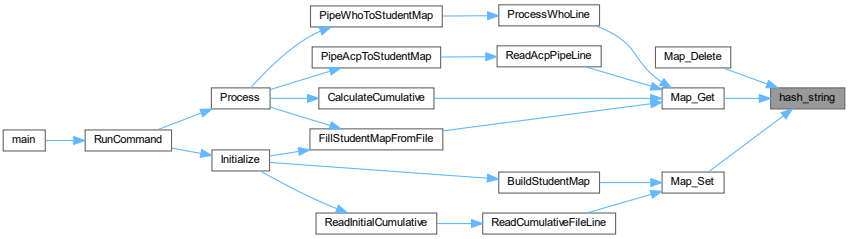
hash_table_capacity	The number of buckets this table holds.
string	The key to hash.
strlen	The length of the key.

Returns

The index of the bucket that should be used.

Definition at line 32 of file map.c.

Here is the caller graph for this function:



6.4.2.4 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

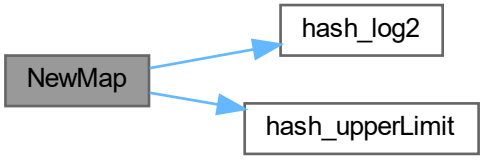
capacity	The estimated required capacity of the map.

Returns

A pointer to the heap allocated map.

Definition at line 47 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.4.2.5 _bucket_insert()

```
void _bucket_insert (
            struct _map_bucket * bucket,
            char * key,
            void * value )
```

Definition at line 65 of file map.c.

Here is the caller graph for this function:

6.4.2.6 Map_Set()

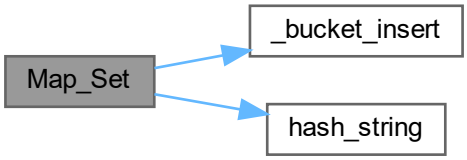
```
void Map_Set (
            map * a_map,
            char * key,
            void * value )
```

Parameters

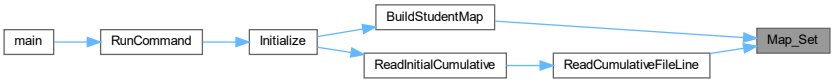
map	The map to set a key in.
key	The key to use.
keylen	The length of the key.
value	The pointer to the data stored at that location.

Definition at line 86 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:

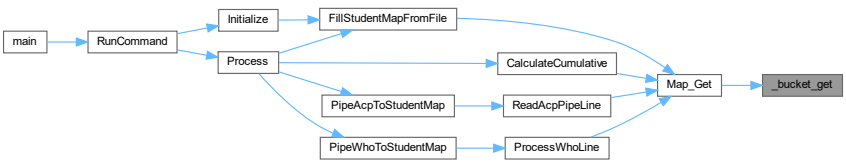


6.4.2.7 _bucket_get()

```
void _bucket_get (
            struct _map_bucket * bucket,
            char * key,
            map_result * result )
```

Definition at line 93 of file map.c.

Here is the caller graph for this function:



6.4.2.8 Map_Get()

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

Parameters

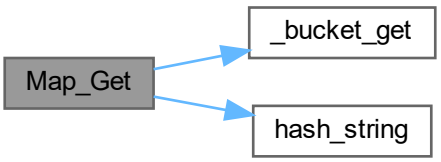
map	The map to retrieve from.
key	The key of the item.

Returns

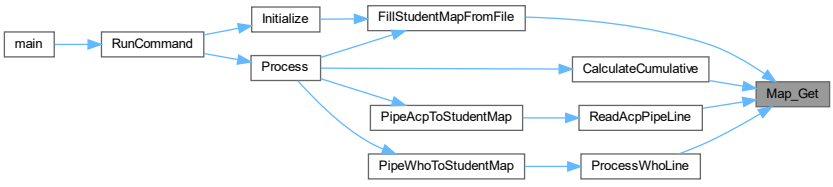
A map_get_result containing the sought data.

Definition at line 116 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:

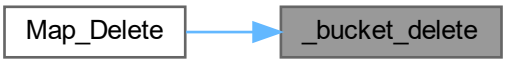


6.4.2.9 _bucket_delete()

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

Definition at line 125 of file map.c.

Here is the caller graph for this function:



6.4.2.10 Map_Delete()

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

Parameters

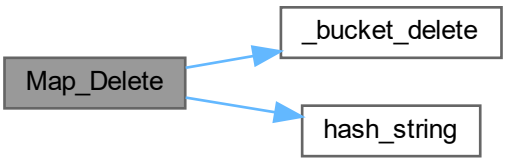
map	The map to delete the key from.
key	The key to delete.
free_it	Whether to call free() on the underlying data.

Returns

A map_get_result with the data that was removed.

Definition at line 151 of file map.c.

Here is the call graph for this function:



6.4.3 Variable Documentation

6.4.3.1 key

```
char* key
```

Definition at line 84 of file map.h.

6.4.3.2 data [1/2]

```
void* data
```

Definition at line 86 of file map.h.

6.4.3.3 next

```
struct _map_bucket* next
```

Definition at line 88 of file map.h.

6.4.3.4 size

```
int size
```

Definition at line 103 of file map.h.

6.4.3.5 buckets

```
struct _map_bucket* buckets
```

Definition at line 105 of file map.h.

6.4.3.6 found

```
short found
```

Definition at line 114 of file map.h.

6.4.3.7 data [2/2]

```
void* data
```

Definition at line 116 of file map.h.

6.5 MemShare

Functions that operate on a shared memory segment.

Macros

- #define MEM_KEY 0x727
- #define MEM_PERMISSIONS 0664
- #define MEM_SIZE DATA_SIZE ∗DATA_NUM_RECORDS

Functions

- int CreateSharedMemory ()
- int DestroySharedMemory ()
- void ∗ GetMemoryPointer (int shared_mem_id)
- int ReleaseMemoryPointer (void ∗shmaddr)

6.5.1 Detailed Description

To share data to clients program uses shared memory

- MEM_KEY is the key to access the shared memory and clients must have this info
- MEM_PERMISSIONS who has read, write permissions of the shared memory segment
- MEM_SIZE the total size of the shared memory allocation

6.5.2 Macro Definition Documentation

6.5.2.1 MEM_KEY

```
#define MEM_KEY 0x727
```

The shared memory key that clients and servers will use to identify the segment.

Definition at line 21 of file memShare.h.

6.5.2.2 MEM_PERMISSIONS

```
#define MEM_PERMISSIONS 0664
```

Memory permissions are: Self: RW 110 = 6 Group: R 100 = 4 Others: R 100 = 4

- All groups can read.
- Self can write.
- None can execute.

Definition at line 32 of file memShare.h.

6.5.2.3 MEM_SIZE

```
#define MEM_SIZE DATA_SIZE *DATA_NUM_RECORDS
```

The memory allocation must as large as the data size times the number of records.

Definition at line 37 of file memShare.h.

6.5.3 Function Documentation

6.5.3.1 CreateSharedMemory()

```
int CreateSharedMemory ( )
```

CreateSharedMemory retrieves a shared memory ID that can be used to access or delete shared memory.

Returns

A shared memory ID that can be used with other 'shm' commands to access shared memory, -1 if an error has occured

Definition at line 9 of file memShare.c.

Here is the caller graph for this function:

6.5.3.2 DestroySharedMemory()

```
int DestroySharedMemory ( )
```

Flags the shared memory segment for deallocation. Returns the result of that operation.

Returns

0 if succesful. 1 if not succesful. Errno will be set.

Definition at line 14 of file memShare.c.

Here is the caller graph for this function:

6.5.3.3 GetMemoryPointer()

```
void * GetMemoryPointer (
            int shared_mem_id )
```

"Attaches" to the shared memory, returning a memory pointer to the shared memory.

Calls 'shmat(shared_mem_id, NULL, 0)`;

Parameters

shared_mem↩ _id	The id of the shared memory

Returns

A pointer to the shared memory, or -1 if it fails.

Definition at line 23 of file memShare.c.

Here is the caller graph for this function:

6.5.3.4 ReleaseMemoryPointer()

```
int ReleaseMemoryPointer (
            void * shmaddr )
```

Release a shm memory pointer.

Parameters

shmaddr	The memory pointer to release.

Returns

Whether the operation was succesful.

Definition at line 28 of file memShare.c.

Here is the caller graph for this function:

6.6 Process

Functions that manage control flow.

Functions

- int TerminateExistingServer ()
- int IndicateRereadNeeded ()
- int IndicateRereadDone ()
- short IsRereadNeeded ()
- void SignalHandle (int signo)
- int Initialize ()
- void Process (int shm_id)
- void HelpCommand ()
- void RunCommand ()

  Runs the server if it doesn't already exist.
- void StopCommand ()
- void ResetCommand ()
- void RunHeadless (char *processName)

Variables

- map * student_map
- map * initial_cumulative_times

  A map of userIDs to integer seconds. These values are subtracted from the current total cumulative time for each user to calculate their cumulative time since the server process started.
- short is_stopping = 0

6.6.1 Detailed Description

This module handles the processes that this server might execute. It calls functions from the other modules to realize program changes.

It contains the main update loop for a running server, Process(), as well as functions for implementing the different command line argument driven procedures.

6.6.2 Function Documentation

6.6.2.1 TerminateExistingServer()

```
int TerminateExistingServer ( )
```

Reads the lockfile to get the ID of the process that created it.

Sends a SIGTERM signal to that process.

Warning

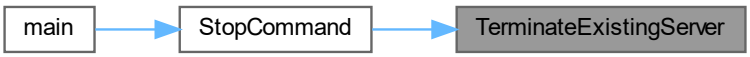
lockfile should be confirmed to exist

Returns

-1 if file doesn't exist, -2 if no valid process ID existed in the file, 1 if sending the kill signal failed.

Definition at line 22 of file Process.c.

Here is the caller graph for this function:



6.6.2.2 IndicateRereadNeeded()

```
int IndicateRereadNeeded ( )
```

If we reset the user data, we need to indicate to the running process that a re-read is needed. This changes the flag in the lockfile to 1, but keeps the same process ID as before there.

Warning

should only be called by non main processes

Returns

-1 if lockfile not found, 0 if success, or an error number if some other error

Definition at line 40 of file Process.c.

Here is the caller graph for this function:

6.6.2.3 IndicateRereadDone()

```
int IndicateRereadDone ( )
```

If we re-read the users file, we can indicate that we have done so by setting the re-read flag back to 0.

Warning

should only be called by main process.

Returns

0 on success, -1 if the file was not found, otherwise an error number produced by fclose.

Definition at line 57 of file Process.c.

Here is the caller graph for this function:

6.6.2.4 IsRereadNeeded()

```
short IsRereadNeeded ( )
```

Reads the lockfile for the re-read flag.

Warning

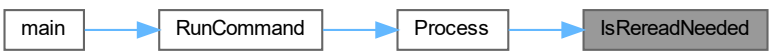
Lockfile should exist - should be called by the server in the main process loop

Returns

0 if the Lockfile starts with '0', 1 if the Lockfile starts with '1'.

Definition at line 74 of file Process.c.

Here is the caller graph for this function:



6.6.2.5 SignalHandle()

```
void SignalHandle (
            int signo )
```

Called by a new server process, telling this server process to shut down. This sets 'is_stopping' to true, which shuts down the server gracefully, writing any necessary data to the user data file, then deleting the lockfile.

Parameters

signo	The signal number will be SIGTERM from the other server process or SIGINT if interrupted from the console.

Definition at line 82 of file Process.c.

Here is the caller graph for this function:

6.6.2.6 Initialize()

```
int Initialize ( )
```

Run once at program start. Calls functions from other modules to do the following:

Note

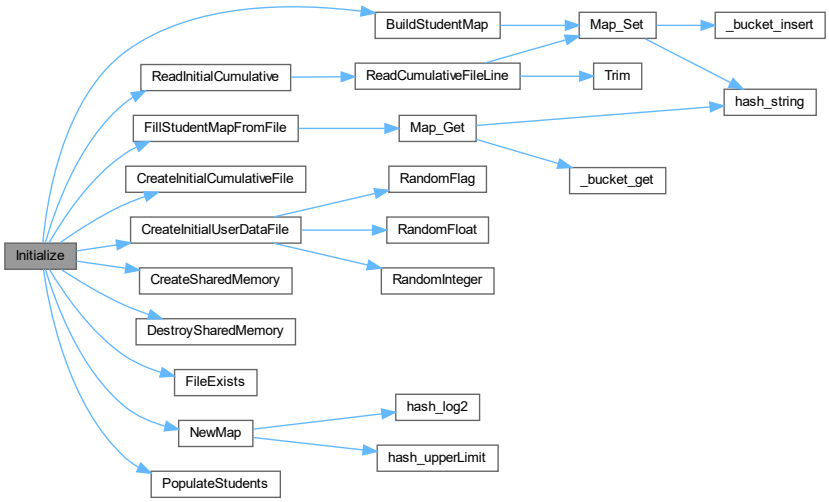
(1) - Create an initial user data file if it doesn't exist.
(2) - Initialize the students array.
(3) - Initialize the students map.
(4) - Read the data from the user data file into the map/array.
(5) - Initializes the shared memory segment.

Returns

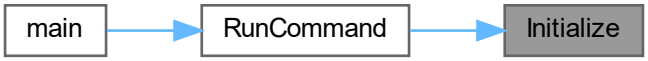
The ID of the shared memory segment or -1 if an error has occured.

Definition at line 97 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



6.6.2.7 Process()

```
void Process (
            int shm_id )
```

Called repeatedly with a delay.

Note

(1) - Sets all users to inactive.

(2) - Reads the result of the `who` command, setting some users to active, and possibly changing 'dirty' and last login times.

(3) - Overwrites the user data file if we are dirty.

(4) - Sets dirty to false.

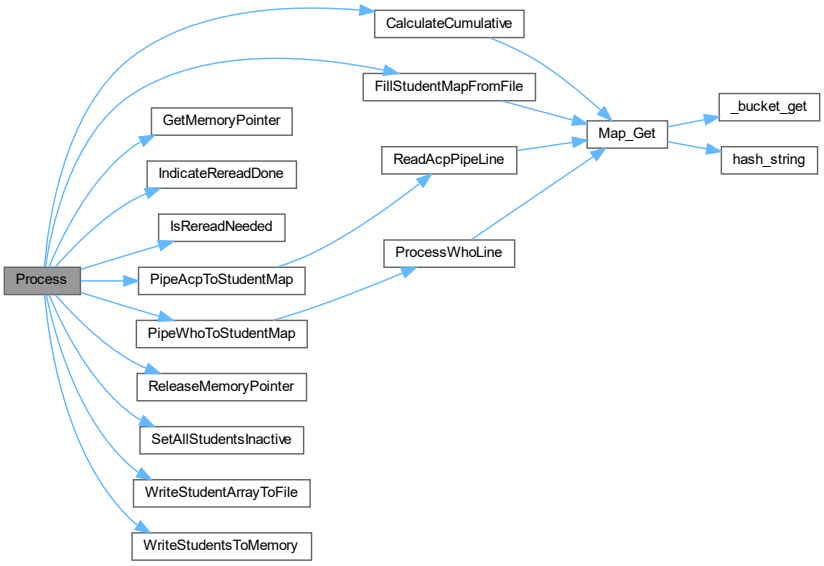
(5) - Rewrites the shared memory.

Parameters

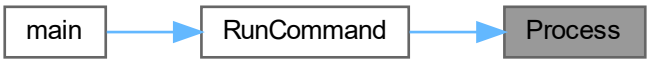
shm↩ _id	The ID of the shared memory segment.

Definition at line 147 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



6.6.2.8 HelpCommand()

```
void HelpCommand ( )
```

Displays the commands available to the user.

Note

To execute the command, pass "help" as an argument to the program.

This command will also run if arg num is incorrect or if invalid option is entered.

Definition at line 194 of file Process.c.

Here is the caller graph for this function:

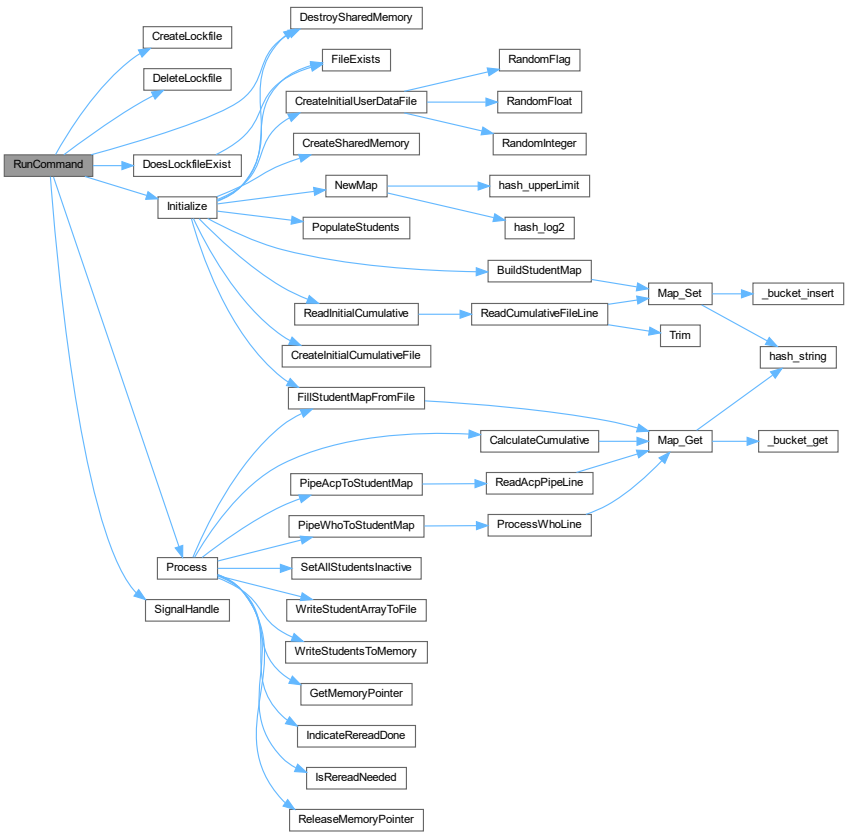
6.6.2.9 RunCommand()

```
void RunCommand ( )
```

This function begins the Process() loop. It is ultimately called via two cli arguments; "run" and "headless".

Definition at line 205 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.6.2.10 StopCommand()

```
void StopCommand ( )
```

Stops an existing server process if it is running by calling `kill` on the pid stored in the Lockfile.

Note

To execute the command, pass "stop" as an argument to the program.

Definition at line 243 of file Process.c.

Here is the call graph for this function:

Here is the caller graph for this function:

6.6.2.11 ResetCommand()

```
void ResetCommand ( )
```

Deletes and recreates the static-user-data file and cumulative login file.

Note

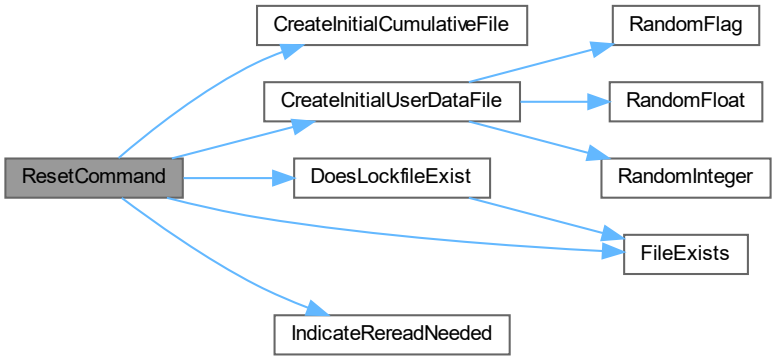
To execute the command, pass "reset" as an argument to the program.

Warning

This will clear login times.

Definition at line 268 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:

6.6.2.12 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 307 of file Process.c.

Here is the call graph for this function:

Here is the caller graph for this function:

6.6.3 Variable Documentation

6.6.3.1 student_map

```
map* student_map
```

Definition at line 18 of file Process.c.

6.6.3.2 initial_cumulative_times

```
map* initial_cumulative_times
```

Definition at line 20 of file Process.c.

6.6.3.3 is_stopping

```
short is_stopping = 0
```

If 0, the server is running and looping, re-reading and writing every second. If 1, it is stopping and shutting down.

Definition at line 93 of file Process.c.

6.7 Util

Helper functions.

Functions

- int RandomInteger (int min, int max)
- float RandomFloat (float min, float max)
- short RandomFlag (float percentage_chance)
- void Trim (char *string)

6.7.1 Detailed Description

Contains utility functions that are not coupled to any other data or structures in the program. Contains randomization functions.

6.7.2 Function Documentation

6.7.2.1 RandomInteger()

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

Returns an integer between min and max.

Parameters

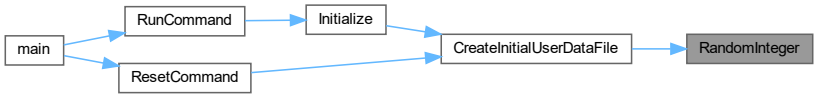
min	The minimum, inclusive.
max	The maximum, inclusive.

Returns

A random integer between min and max.

Definition at line 10 of file util.c.

Here is the caller graph for this function:



6.7.2.2 RandomFloat()

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

Returns a float between min and max.

Parameters

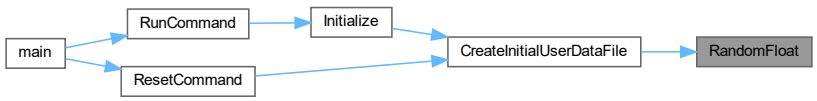
min	The minimum, inclusive.
max	The maximum, inclusive.

Returns

A random integer between min and max.

Definition at line 16 of file util.c.

Here is the caller graph for this function:



6.7.2.3 RandomFlag()

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

Returns 1, percentage_chance of the time.

Parameters

percentage_chance	The chance to return 1.

Note

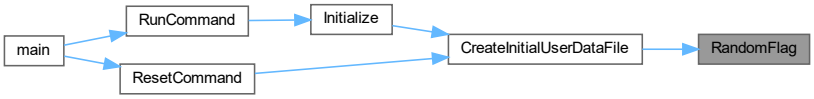
If percentage_chance $>$ 1, this will always return true.

Returns

1 or 0

Definition at line 23 of file util.c.

Here is the caller graph for this function:



6.7.2.4 Trim()

```
void Trim (
            char * string )
```

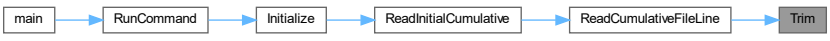
Trims a string by setting the first whitespace character found to the null-terminator.

Parameters

string	The string to trim.

Definition at line 33 of file util.c.

Here is the caller graph for this function:



Chapter 7

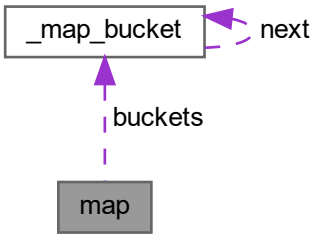
Data Structure Documentation

7.1 map Struct Reference

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

`#include <map.h>`

Collaboration diagram for map:



Data Fields

- int size

  The number of base buckets in this map.
- struct _map_bucket * buckets

  The buckets for this map.

7.1.1 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 100 of file map.h.

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

- src/server/map.h

7.2 map_result Struct Reference

The result of a map retrieval.

`#include <map.h>`

Data Fields

- short found

  1 if succesfully found. 0 if not found.
- void * data

  The data linked with that key; indeterminate if found == 0.

7.2.1 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

7.3 Student Struct Reference

The student data type.

`#include <Data.h>`

Data Fields

- char userID [DATA_ID_MAX_LENGTH]

  The unique user ID.
- char fullName [DATA_NAME_MAX_LENGTH]

  The user's full name.
- short age

  The user's age (randomized).
- float gpa

  The user's gpa (randomized).
- short active

  Whether the user is currently logged in (1) or not (0).
- time_t lastLogin

  The last time the user logged in.
- int loginDuration

  The cumulative time the user has been logged in since the server process started.

7.3.1 Detailed Description

Definition at line 41 of file Data.h.

7.3.2 Field Documentation

7.3.2.1 userID

```
char userID[DATA_ID_MAX_LENGTH]
```

Definition at line 44 of file Data.h.

7.3.2.2 fullName

```
char fullName[DATA_NAME_MAX_LENGTH]
```

Definition at line 46 of file Data.h.

7.3.2.3 age

```
short age
```

Definition at line 48 of file Data.h.

7.3.2.4 gpa

```
float gpa
```

Definition at line 50 of file Data.h.

7.3.2.5 active

```
short active
```

Definition at line 52 of file Data.h.

7.3.2.6 lastLogin

```
time_t lastLogin
```

Definition at line 54 of file Data.h.

7.3.2.7 loginDuration

```
int loginDuration
```

Definition at line 56 of file Data.h.

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

- src/server/Data.h

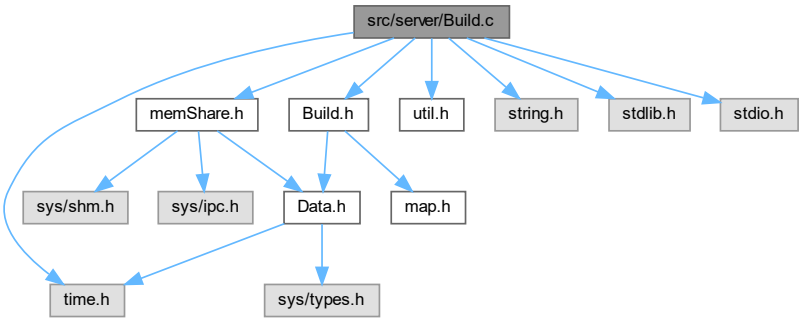
Chapter 8

File Documentation

8.1 src/server/Build.c File Reference

```
#include "Build.h"
#include "memShare.h"
#include "util.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

Include dependency graph for Build.c:



Functions

- void PopulateStudents (char **studentIDs, char **studentNames, int arsize)
- void BuildStudentMap (map *stmap, Student *studentArr, int studentArrLength)
- int PipeWhoToStudentMap (map *stmap)
- int ProcessWhoLine (map *stmap, char *whoLine, int whoLineLength)
- void SetAllStudentsInactive (Student *stud_arr, int arr_len)
- void WriteStudentsToMemory (void *mem_ptr, Student *stud_arr, int arr_len)
- int ReadInitialCumulative (map *time_map, char *filename)
- int PipeAcpToStudentMap (map *st_map)
- void ReadCumulativeFileLine (map *cum_map, char *acp_line)
- int ReadAcpPipeLine (map *stmap, char *acp_line)
- void CalculateCumulative (Student *stud_arr, int stud_arr_len, map *cum_map)

Variables

- Student * students
- short dirty = 1

8.2 Build.c

Go to the documentation of this file.

```
00001
00005 #include "Build.h"
00006 #include "memShare.h"
00007 #include "util.h"
00008 #include <string.h>
00009 #include <stdlib.h>
00010 #include <stdio.h>
00011 #include <time.h>
00012
00013 // ~~~~~~~~  Data Structures ~~~~~~~~~
00014
00015 Student *students;
00016
00017 void PopulateStudents(char **studentIDs, char **studentNames, int arsize)
00018 {
00019     students = malloc(sizeof(Student) * arsize);
00020     int i;
00021     for (i = 0; i < arsize; i++)
00022     {
00023         strcpy(students[i].userID, studentIDs[i]);
00024         strcpy(students[i].fullName, studentNames[i]);
00025     }
00026 }
00027
00028 void BuildStudentMap(map *stmap, Student *studentArr, int studentArrLength)
00029 {
00030     int i;
00031     for (i = 0; i < studentArrLength; i++)
00032     {
00033         Map_Set(stmap, studentArr[i].userID, (void *)(&studentArr[i]));
00034     }
00035 }
00036
00037 // ~~~~~~~~  Processing ~~~~~~~~~
00038
00039 short dirty = 1; // start dirty
00040
00041 int PipeWhoToStudentMap(map *stmap)
00042 {
00043     char command[4] = "who";
00044     char line[100];
00045     FILE *fpipe;
00046     fpipe = popen(command, "r");
00047     if (fpipe == NULL)
00048     {
00049         return -1;
00050     }
00051
00052     while (fgets(line, sizeof(line), fpipe) != NULL)
00053     {
00054         ProcessWhoLine(stmap, line, strlen(line));
00055     }
00056     pclose(fpipe);
00057
00058     return 0;
00059 }
00060
00061 int ProcessWhoLine(map *stmap, char *whoLine, int whoLineLength)
00062 {
00063     char userId[20];
00064     char dateString[50];
00065     char timeString[20];
00066     int read_total = 0;
00067     int read;
00068     sscanf(whoLine, " %s %n", userId, &read);
00069     read_total += read;
00070
00071     map_result mr = Map_Get(stmap, userId);
00072     if (!mr.found)
00073     { // if we can't find that person in the map, return early
00074         return -1;
00075     }
```

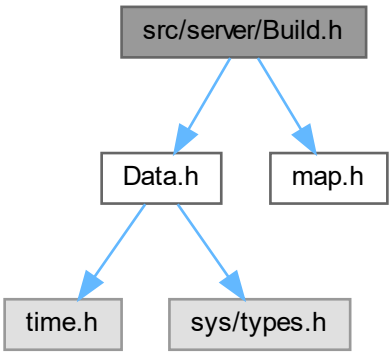
```
00076     Student *student = (Student *)mr.data;
00077
00078     sscanf(whoLine + read_total, " %s %n", dateString, &read); // will be thrown away. eg 'pts/1'
00079     read_total += read;
00080     sscanf(whoLine + read_total, " %s %n", dateString, &read); // read the date string
00081     read_total += read;
00082     sscanf(whoLine + read_total, " %s %n", timeString, &read); // read the time string
00083     strcat(dateString, " ");
00084     strcat(dateString, timeString); // catenate the time string back to the date string
00085
00086     time_t now = time(NULL);
00087     struct tm dtime = *localtime(&now);
00088     dtime.tm_sec = 0;
00089
00090     memset(&dtime, 0, sizeof(struct tm));
00091
00092     sscanf(dateString, "%d-%d-%d %d:%d", &(dtime.tm_year), &(dtime.tm_mon), &(dtime.tm_mday),
      &(dtime.tm_hour), &(dtime.tm_min));
00093
00094     dtime.tm_year -= 1900;
00095     dtime.tm_mon -= 1;
00096     dtime.tm_hour -= 1;
00097
00098     time_t parsed_time = mktime(&dtime);
00099
00100     if (student->lastLogin != parsed_time)
00101     {
00102         student->lastLogin = parsed_time;
00103         dirty = 1;
00104     }
00105     student->active = 1;
00106     return 0;
00107 }
00108
00109 void SetAllStudentsInactive(Student *stud_arr, int arr_len)
00110 {
00111     int i;
00112     for (i = 0; i < arr_len; i++)
00113     {
00114         stud_arr[i].active = 0;
00115     }
00116 }
00117
00118 void WriteStudentsToMemory(void *mem_ptr, Student *stud_arr, int arr_len)
00119 {
00120     Student *memloc = (Student *)mem_ptr;
00121     int i;
00122     for (i = 0; i < arr_len; i++)
00123     {
00124         strcpy(memloc[i].userID, stud_arr[i].userID);
00125         strcpy(memloc[i].fullName, stud_arr[i].fullName);
00126         memloc[i].age = stud_arr[i].age;
00127         memloc[i].gpa = stud_arr[i].gpa;
00128         memloc[i].active = stud_arr[i].active;
00129         memloc[i].lastLogin = stud_arr[i].lastLogin;
00130         memloc[i].loginDuration = stud_arr[i].loginDuration;
00131     }
00132 }
00133
00134 // ~~~~~~~~  Cumulative Processing ~~~~~~~~~
00135
00136 int ReadInitialCumulative(map *time_map, char *filename)
00137 {
00138     FILE *file = fopen(filename, "r");
00139     char line[100];
00140     if (file == NULL)
00141     {
00142         return -1;
00143     }
00144     while (fgets(line, sizeof(line), file) != NULL)
00145     {
00146         ReadCumulativeFileLine(time_map, line);
00147     }
00148
00149     fclose(file);
00150     return 0;
00151 }
00152
00153 int PipeAcpToStudentMap(map *st_map)
00154 {
00155     char command[6] = "ac -p";
00156     char line[300];
00157     FILE *fpipe;
00158     fpipe = popen(command, "r");
00159     if (fpipe == NULL)
00160     {
00161         return -1;
```

```
00162     }
00163     int err;
00164     while (fgets(line, sizeof(line), fpipe) != NULL)
00165     {
00166         err = ReadAcpPipeLine(st_map, line);
00167         if (err)
00168         {
00169             printf("\nError %d reading acp pipeline.", err);
00170             break;
00171         }
00172     }
00173     pclose(fpipe);
00174     return 0;
00175 }
00176
00177 void ReadCumulativeFileLine(map *cum_map, char *acp_line)
00178 {
00179     char userId[20];
00180     float hours;
00181     sscanf(acp_line, " %s %f ", userId, &hours);
00182     // int seconds = (int) (minutes * 60)
00183     long seconds = (long)(hours * 60 *60);
00184     // if(strcmp(userId, "mil7233") == 0) {
00185     //     printf("Cum file line for %s seconds = %ld\n", userId, seconds);
00186     // }
00187     Trim(userId);
00188     char* key = malloc( (strlen(userId)+1) * sizeof(char));
00189     strcpy(key, userId);
00190     Map_Set(cum_map, key, (void *)seconds);
00191 }
00192
00193 int ReadAcpPipeLine(map *stmap, char *acp_line)
00194 {
00195     if (acp_line == NULL || strlen(acp_line) < 1)
00196     {
00197         return -1;
00198     }
00199     char userId[40];
00200     float hours;
00201     sscanf(acp_line, "%s %f", userId, &hours);
00202     map_result result = Map_Get(stmap, userId);
00203     if (result.found)
00204     {
00205         Student *student = (Student *)result.data;
00206         int seconds = (int)(hours * 60*60);
00207         student->loginDuration = seconds;
00208         // if(strcmp(userId, "mil7233")==0) {
00209         //     printf("ACP pipe for %s student quant = %f\n", userId, hours);
00210         //     printf(" --- int seconds = %d\n", seconds);
00211         // }
00212     }
00213     return 0;
00214 }
00215
00216
00217 void CalculateCumulative(Student *stud_arr, int stud_arr_len, map *cum_map)
00218 {
00219     int i;
00220     for (i = 0; i < stud_arr_len; i++)
00221     {
00222         map_result result = Map_Get(cum_map, stud_arr[i].userID);
00223         if (result.found)
00224         {
00225
00226             long time_at_server_start = (long)result.data;
00227             stud_arr[i].loginDuration = stud_arr[i].loginDuration - time_at_server_start;
00228             // if(strcmp("mil7233", stud_arr->userID) == 0) {
00229             //     printf("calc cum: found user %s.\n", stud_arr[i].userID);
00230             //     printf(" tot time now: %ld\n", stud_arr[i].loginDuration);
00231             //     printf(" time at server start: %ld\n", time_at_server_start);
00232             //     printf(" new duration: %ld\n", stud_arr[i].loginDuration);
00233             // }
00234         }
00235     }
00236 }
```

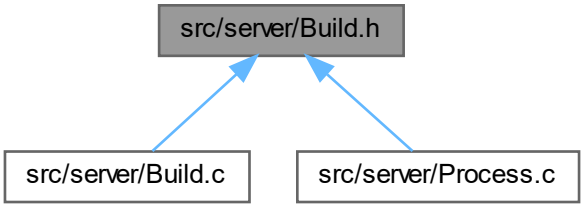
8.3 src/server/Build.h File Reference

```
#include "Data.h"
#include "map.h"
```

Include dependency graph for Build.h:



This graph shows which files directly or indirectly include this file:



Functions

- void PopulateStudents (char **studentIDs, char **studentNames, int arsize)
- void BuildStudentMap (map *stmap, Student *studentArr, int studentArrLength)
- int PipeWhoToStudentMap (map *stmap)
- int ProcessWhoLine (map *stmap, char *whoLine, int whoLineLength)
- void SetAllStudentsInactive (Student *stud_arr, int arr_len)
- void WriteStudentsToMemory (void *mem_ptr, Student *stud_arr, int arr_len)
- int ReadInitialCumulative (map *time_map, char *filename)
- int PipeAcpToStudentMap (map *st_map)
- void ReadCumulativeFileLine (map *time_map, char *acp_line)
- int ReadAcpPipeLine (map *stmap, char *acp_line)
- void CalculateCumulative (Student *stud_arr, int stud_arr_len, map *time_map)

Variables

- Student * students
- short dirty

8.4 Build.h

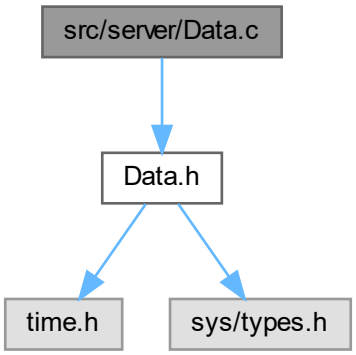
Go to the documentation of this file.

```
00001 #ifndef BUILD_H
00002 #define BUILD_H
00009 #include "Data.h"
00010 #include "map.h"
00011
00012 // ~~~~~~~~  Data Structures ~~~~~~~~~
00013
00019 extern Student *students;
00020
00028 void PopulateStudents(char **studentIDs, char **studentNames, int arsize);
00029
00036 void BuildStudentMap(map *stmap, Student *studentArr, int studentArrLength);
00037
00038 // ~~~~~~~~  Processing ~~~~~~~~~
00039
00041 extern short dirty;
00042
00049 int PipeWhoToStudentMap(map *stmap);
00050
00064 int ProcessWhoLine(map *stmap, char *whoLine, int whoLineLength);
00065
00072 void SetAllStudentsInactive(Student *stud_arr, int arr_len);
00073
00082 void WriteStudentsToMemory(void *mem_ptr, Student *stud_arr, int arr_len);
00083
00084 // ~~~~~~~~  Cumulative Processing ~~~~~~~~~
00085
00095 int ReadInitialCumulative(map *time_map, char *filename);
00096
00105 int PipeAcpToStudentMap(map *st_map);
00106
00115 void ReadCumulativeFileLine(map *time_map, char *acp_line);
00116
00124 int ReadAcpPipeLine(map *stmap, char *acp_line);
00125
00135 void CalculateCumulative(Student *stud_arr, int stud_arr_len, map *time_map);
00136
00141 #endif
```

8.5 src/server/Data.c File Reference

`#include "Data.h"`

Include dependency graph for Data.c:



Variables

- char * Data_IDs [DATA_NUM_RECORDS]
- char * Data_Names [DATA_NUM_RECORDS]

8.6 Data.c

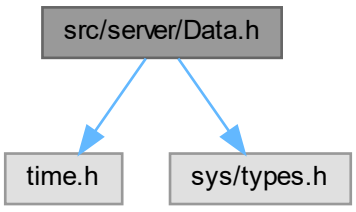
Go to the documentation of this file.

```
00001
00005 #include "Data.h"
00006
00007 char *Data_IDs[DATA_NUM_RECORDS] = {
00008     "chen",
00009     "bea1389",
00010     "bol4559",
00011     "cal6258",
00012     "kre5277",
00013     "lon1150",
00014     "mas9309",
00015     "mes08346",
00016     "mil7233",
00017     "nef9476",
00018     "nov7488",
00019     "pan9725",
00020     "rac3146",
00021     "rub4133",
00022     "shr5683",
00023     "vay3083",
00024     "yos2327"};
00025
00026 char *Data_Names[DATA_NUM_RECORDS] = {
00027     "Weifeng Chen",
00028     "Christian Beatty",
00029     "Emily Bolles",
00030     "Cameron Calhoun",
00031     "Ty Kress",
00032     "Cody Long",
00033     "Caleb Massey",
00034     "Christian Messmer",
00035     "Karl Miller",
00036     "Jeremiah Neff",
00037     "Kaitlyn Novacek",
00038     "Joshua Panaro",
00039     "Caleb Rachocki",
00040     "Caleb Ruby",
00041     "Paul Shriner",
00042     "Alan Vayansky",
00043     "Assefa Ayalew Yoseph"};
00044
```

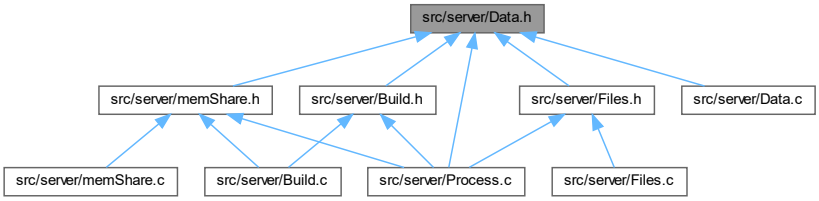
8.7 src/server/Data.h File Reference

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

Include dependency graph for Data.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct Student

  The student data type.

Macros

- #define DATA_NUM_RECORDS 17
- #define DATA_ID_MAX_LENGTH 9
- #define DATA_NAME_MAX_LENGTH 21
- #define DATA_SIZE 56

Variables

- char * Data_IDs [ ]
- char * Data_Names [ ]

8.8 Data.h

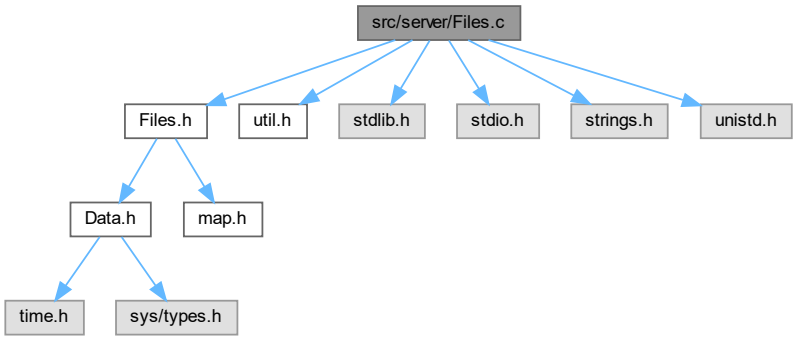
Go to the documentation of this file.

```
00001 #ifndef Data_h
00002 #define Data_h
00009 #include <time.h>
00010 #include <sys/types.h>
00011
00015 #define DATA_NUM_RECORDS 17
00020 #define DATA_ID_MAX_LENGTH 9
00025 #define DATA_NAME_MAX_LENGTH 21
00026
00027 /* Constant, all user IDs. */
00028 extern char *Data_IDs[];
00029
00031 extern char *Data_Names[];
00032
00036 #define DATA_SIZE 56
00037
00041 typedef struct
00042 {
00044     char userID[DATA_ID_MAX_LENGTH];
00046     char fullName[DATA_NAME_MAX_LENGTH];
00048     short age;
00050     float gpa;
00052     short active;
00054     time_t lastLogin;
00056     int loginDuration;
00057 } Student;
00061 #endif
```

8.9 src/server/Files.c File Reference

```
#include "Files.h"
#include "util.h"
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <unistd.h>
```

Include dependency graph for Files.c:



Functions

- short FileExists (char *file_name_to_check)

  Determines whether a file exists.
- int CreateInitialUserDataFile (char *file_name, char **id_list, int id_list_len)

  Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.
- int FillStudentMapFromFile (map *student_map, char *file_name, char **id_list, int id_list_len)

  Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.
- int WriteStudentArrayToFile (Student *students, int arr_len, char *file_name)

  Writes the student array to the file.
- int CreateInitialCumulativeFile (char *file_name)
- short DoesLockfileExist ()
- int CreateLockfile ()
- int DeleteLockfile ()

8.10 Files.c

Go to the documentation of this file.

```
00001
00005 #include "Files.h"
00006 #include "util.h"
00007 #include <stdlib.h>
00008 #include <stdio.h>
00009 #include <strings.h>
00010 #include <unistd.h>
00011
00012 short FileExists(char *file_name_to_check)
00013 {
00014     FILE *file = fopen(file_name_to_check, "r");
```

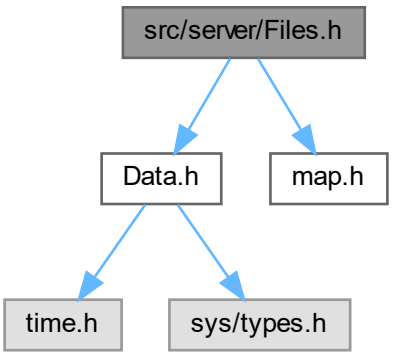
```
00015     short result = 1;
00016     if (file == NULL)
00017     {
00018         result = 0;
00019     }
00020     else
00021     {
00022         fclose(file);
00023     }
00024     return result;
00025 }
00026
00027 int CreateInitialUserDataFile(char *file_name, char **id_list, int id_list_len)
00028 {
00029     FILE *file = fopen(file_name, "w");
00030     if (file == NULL)
00031     {
00032         return -1;
00033     }
00034     int i;
00035     for (i = 0; i < id_list_len; i++)
00036     {
00037         int rand_age = RandomInteger(18, 22);
00038         float gpa;
00039         if (RandomFlag(0.42))
00040         {
00041             gpa = 4.0; // 42% of the time, make the GPA 4.0
00042         }
00043         else
00044         {
00045             gpa = RandomFloat(2.5, 4.0);
00046         }
00047         fprintf(file, "%s\t%d\t%.2f\t%d\n", id_list[i], rand_age, gpa, 0);
00048     }
00049     fclose(file);
00050     return 0;
00051 }
00052
00053 int FillStudentMapFromFile(map *student_map, char *file_name, char **id_list, int id_list_len)
00054 {
00055     FILE *file = fopen(file_name, "r");
00056     if (file == NULL)
00057     {
00058         return -1;
00059     }
00060     // id buffer
00061     char user_id[9];
00062     int age;
00063     float gpa;
00064     long time;
00065     while (fscanf(file, "%9s\t%d\t%f\t%ld", user_id, &age, &gpa, &time) == 4)
00066     {
00067         map_result result = Map_Get(student_map, user_id);
00068         if (result.found == 0)
00069         {
00070             continue;
00071         }
00072         ((Student *)result.data)->age = age;
00073         ((Student *)result.data)->gpa = gpa;
00074         ((Student *)result.data)->lastLogin = time;
00075     }
00076     fclose(file);
00077     return 0;
00078 }
00079
00080 int WriteStudentArrayToFile(Student *students, int arr_len, char *file_name)
00081 {
00082     FILE *file = fopen(file_name, "w");
00083     if (file == NULL)
00084     {
00085         return -1;
00086     }
00087     int i;
00088     for (i = 0; i < arr_len; i++)
00089     {
00090         fprintf(file, "%s\t%d\t%.2f\t%ld\n", students[i].userID, students[i].age, students[i].gpa,
      students[i].lastLogin);
00091     }
00092     fclose(file);
00093     return 0;
00094 }
00095
00096 int CreateInitialCumulativeFile(char *file_name)
00097 {
00098     FILE *file = fopen(file_name, "w");
00099     if (file == NULL)
00100     {
```

```
00101         return -1;
00102     }
00103     FILE *pipe = popen("ac -p", "r");
00104     if (pipe == NULL)
00105     {
00106         fclose(file);
00107         return -2;
00108     }
00109
00110     char line[100];
00111     while (fgets(line, sizeof(line), pipe) != NULL)
00112     {
00113         fputs(line, file);
00114     }
00115     pclose(pipe);
00116     fclose(file);
00117     return 0;
00118 }
00119
00120 // ~~~~~~~~~~~~~~~ Lockfile Commands ~~~~~~~~~~~~~~~~~~~~
00121
00122 short DoesLockfileExist()
00123 {
00124     return FileExists(LOCKFILE);
00125 }
00126
00127 int CreateLockfile()
00128 {
00129     FILE *file = fopen(LOCKFILE, "w");
00130     if (file == NULL)
00131     {
00132         return -1;
00133     }
00134     fprintf(file, "0 %d", getpid());
00135     fclose(file);
00136     return 0;
00137 }
00138
00139 int DeleteLockfile()
00140 {
00141     return remove(LOCKFILE);
00142 }
```

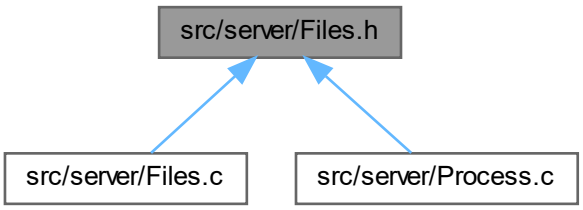
8.11 src/server/Files.h File Reference

```
#include "Data.h"
#include "map.h"
```

Include dependency graph for Files.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define STATIC_USER_DATA_FILE "static-user-data.txt"
- #define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"
- #define LOCKFILE "/tmp/ecet-server.lock"

Functions

- short FileExists (char *file_name_to_check)

  Determines whether a file exists.
- int CreateInitialUserDataFile (char *file_name, char **id_list, int id_list_len)

  Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.
- int WriteStudentArrayToFile (Student *students, int arr_len, char *file_name)

  Writes the student array to the file.
- int FillStudentMapFromFile (map *student_map, char *file_name, char **id_list, int id_list_len)

  Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.
- int CreateInitialCumulativeFile (char *file_name)
- short DoesLockfileExist ()
- int CreateLockfile ()
- int DeleteLockfile ()

8.12 Files.h

Go to the documentation of this file.

```
00001 #ifndef Files_H
00002 #define Files_H
00015 #include "Data.h"
00016 #include "map.h"
00017
00033 #define STATIC_USER_DATA_FILE "static-user-data.txt"
00034
00043 #define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"
00044
00053 #define LOCKFILE "/tmp/ecet-server.lock"
00054
00059 short FileExists(char *file_name_to_check);
00060
00070 int CreateInitialUserDataFile(char *file_name, char **id_list, int id_list_len);
00071
00081 int WriteStudentArrayToFile(Student *students, int arr_len, char *file_name);
```

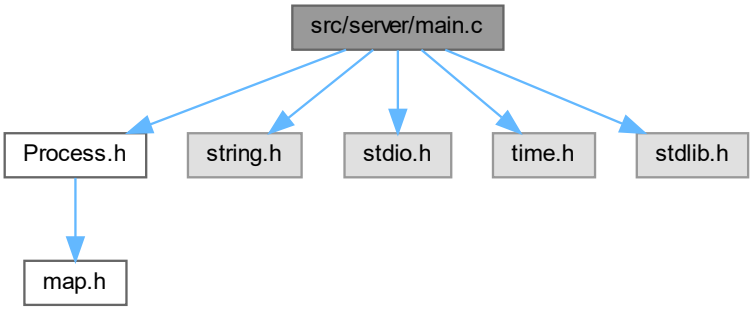
```
00082
00093 int FillStudentMapFromFile(map *student_map, char *file_name, char **id_list, int id_list_len);
00094
00104 int CreateInitialCumulativeFile(char *file_name);
00105
00106
00107
00108 // ~~~~~~~~~~~~~~~ Lockfile Commands ~~~~~~~~~~~~~~~~~~~
00109
00115 short DoesLockfileExist();
00116
00125 int CreateLockfile();
00126
00131 int DeleteLockfile();
00136 #endif
```

8.13 src/server/main.c File Reference

Program entry point.

```
#include "Process.h"
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
```

Include dependency graph for main.c:



Functions

- int main (int argc, char **argv)
  Program entry.

8.13.1 Function Documentation

8.13.1.1 main()

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

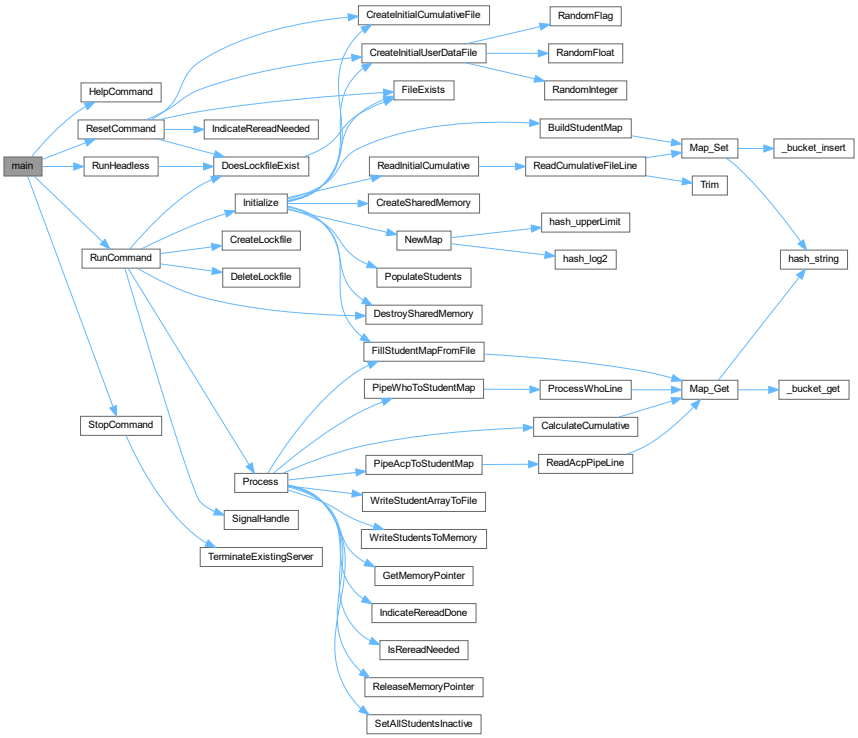
Parses arguments and calls the appropriate Process.h function.

Parameters

argc	The argument count.
argv	The argument values.

Definition at line 104 of file main.c.

Here is the call graph for this function:



8.14 main.c

Go to the documentation of this file.

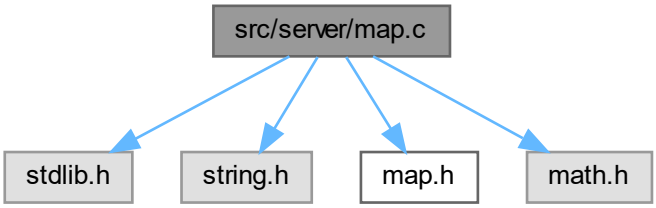
```
00001
00005 #include "Process.h"
00006 #include <string.h>
00007 #include <stdio.h>
00008 #include <time.h>
00009 #include <stdlib.h>
00104 int main(int argc, char **argv)
00105 {
00106     srand(time(NULL)); // seed the randomizer
00107
00108     if (argc <= 1 || argc >= 3)
00109     {
00110         printf("Too few or many options!\n");
00111         HelpCommand();
00112     }
00113     else if (strcmp(argv[1], "help") == 0)
00114     {
00115         HelpCommand();
00116     }
00117     else if (strcmp(argv[1], "reset") == 0)
00118     {
00119         ResetCommand();
```

```
00120     }
00121     else if (strcmp(argv[1], "stop") == 0 || strcmp(argv[1], "end") == 0 || strcmp(argv[1], "close")
      == 0 || strcmp(argv[1], "exit") == 0)
00122     {
00123         StopCommand();
00124     }
00125     else if (strcmp(argv[1], "headless") == 0)
00126     {
00127         RunHeadless(argv[0]);
00128     }
00129     else if (strcmp(argv[1], "run") == 0 || strcmp(argv[1], "start") == 0)
00130     {
00131         RunCommand();
00132     }
00133     else
00134     {
00135         printf("Unknown option!\n");
00136         HelpCommand();
00137     }
00138     return 0;
00139 }
```

8.15 src/server/map.c File Reference

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

Include dependency graph for map.c:



Functions

- int hash_log2 (int number_to_log)

  Get's a log2 ceiling. Eg, hash_log2(5) == 3.
- int hash_upperLimit (int bitsize)

  This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.
- int hash_string (int hash_table_size, char *string, int strlen)
- map * NewMap (int capacity)
- void _bucket_insert (struct _map_bucket *bucket, char *key, void *value)
- void Map_Set (map *a_map, char *key, void *value)

  Sets a value in the map.
- void _bucket_get (struct _map_bucket *bucket, char *key, map_result *result)
- 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.

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

8.16 map.c

Go to the documentation of this file.

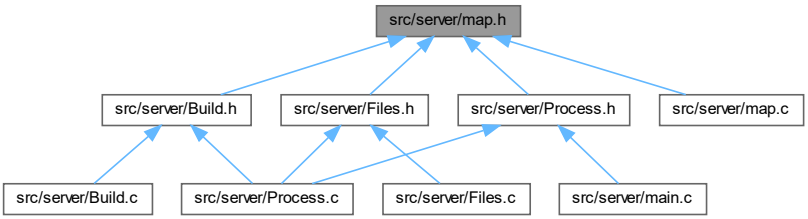
```
00001
00005 #include "stdlib.h"
00006 #include "string.h"
00007 #include "map.h"
00008 #include "math.h"
00009
00010 int hash_log2(int num_to_log)
00011 {
00012     int t = 1;
00013     int i = 0;
00014     do
00015     {
00016         num_to_log = num_to_log & ~t;
00017         t = t « 1;
00018         i++;
00019     } while (num_to_log > 0);
00020     return i;
00021 }
00022
00023 int hash_upperLimit(int bitsize)
00024 {
00025     return 1 « bitsize;
00026 }
00027
00029 int char_ratio = (int)(sizeof(int) / sizeof(char));
00030
00031 // Modified some stuff from : http://isthe.com/chongo/tech/comp/fnv/
00032 int hash_string(int hash_table_size, char *string, int strlen)
00033 {
00034     int i, hash = 2166136261;
00035     for (i = 0; i < strlen; i += 1)
00036     {
00037         hash *= 16777619;
00038         hash ^= string[i];
00039     }
00040     if (hash < 0)
00041     {
00042         hash *= -1;
00043     }
00044     return hash % hash_table_size;
00045 }
00046
00047 map *NewMap(int capacity)
00048 {
00049     int log2 = hash_log2(capacity);
00050     int capac = hash_upperLimit(log2);
00051     int sz = sizeof(struct _map_bucket) * capac;
00052     struct _map_bucket *buckets = malloc(sz);
00053     memset(buckets, 0, sz);
00054     int i;
00055     for (i = 0; i < capac; i++)
00056     {
00057         buckets[i] = (struct _map_bucket){NULL, NULL, NULL};
00058     }
00059     map newm = (map){capac, buckets};
00060     map *map_p = malloc(sizeof(map));
00061     *map_p = newm;
00062     return map_p;
00063 }
00064
00065 void _bucket_insert(struct _map_bucket *bucket, char *key, void *value)
00066 {
00067     struct _map_bucket *check = bucket;
00068     while (check->key != NULL)
00069     {
00070         if (strcmp(check->key, key) == 0)
00071         {
00072             check->data = value;
```

```
00073             return;
00074         }
00075         if (check->next == NULL)
00076         {
00077             check->next = malloc(sizeof(struct _map_bucket));
00078             *(check->next) = (struct _map_bucket){NULL, NULL, NULL};
00079         }
00080         check = check->next;
00081     }
00082     check->key = key;
00083     check->data = value;
00084 }
00085
00086 void Map_Set(map *a_map, char *key, void *value)
00087 {
00088     int keyl = (int)strlen(key);
00089     int hash = hash_string(a_map->size, key, keyl);
00090     _bucket_insert(&(a_map->buckets[hash]), key, value);
00091 }
00092
00093 void _bucket_get(struct _map_bucket *bucket, char *key, map_result *result)
00094 {
00095     struct _map_bucket *check = bucket;
00096     while (check->key != NULL)
00097     {
00098         if (strcmp(check->key, key) == 0)
00099         {
00100             result->found = 1;
00101             result->data = check->data;
00102             return;
00103         }
00104         else if (check->next != NULL)
00105         {
00106             check = check->next;
00107         }
00108         else
00109         {
00110             result->found = 0;
00111             break;
00112         }
00113     }
00114 }
00115
00116 map_result Map_Get(map *a_map, char *key)
00117 {
00118     map_result res = (map_result){0, NULL};
00119     int keyl = (int)strlen(key);
00120     int hash = hash_string(a_map->size, key, keyl);
00121     _bucket_get(&(a_map->buckets[hash]), key, &res);
00122     return res;
00123 }
00124
00125 void _bucket_delete(struct _map_bucket *bucket, char *key, short free_it, map_result *result)
00126 {
00127     struct _map_bucket *last = bucket;
00128     struct _map_bucket *next = bucket->next;
00129     while (next != NULL)
00130     {
00131         if (strcmp(next->key, key) == 0)
00132         {
00133             result->found = 1;
00134             result->data = next->data;
00135             if (free_it)
00136             {
00137                 free(next->data);
00138                 result->data = NULL;
00139             }
00140             last->next = next->next;
00141             free(next);
00142         }
00143         else
00144         {
00145             last = next;
00146             next = next->next;
00147         }
00148     }
00149 }
00150
00151 map_result Map_Delete(map *a_map, char *key, short free_it)
00152 {
00153     map_result res = (map_result){0, NULL};
00154     int keyl = (int)strlen(key);
00155     int hash = hash_string(a_map->size, key, keyl);
00156
00157     struct _map_bucket top = a_map->buckets[hash];
00158     if (top.key == NULL)
00159     {
```

```
00160         return res;
00161     }
00162     if (strcmp(top.key, key) == 0)
00163     {
00164         res.found = 1;
00165         res.data = top.data;
00166         if (free_it)
00167         {
00168             free(top.data);
00169             res.data = NULL;
00170         }
00171         if (top.next != NULL)
00172         {
00173             a_map->buckets[hash] = *(top.next);
00174             free(top.next);
00175         }
00176         else
00177         {
00178             a_map->buckets[hash] = (struct _map_bucket){NULL, NULL, NULL};
00179         }
00180         return res;
00181     }
00182     if (top.next == NULL)
00183     {
00184         return res;
00185     }
00186     _bucket_delete(&(a_map->buckets[hash]), key, free_it, &res);
00187
00188     return res;
00189 }
```

8.17 src/server/map.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- 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

- int hash_log2 (int number_to_log)

  Get's a log2 ceiling. Eg, hash_log2(5) == 3.
- int hash_string (int hash_table_size, char *string, int strlen)
- int hash_upperLimit (int bitsize)

This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.

- 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.

8.18 map.h

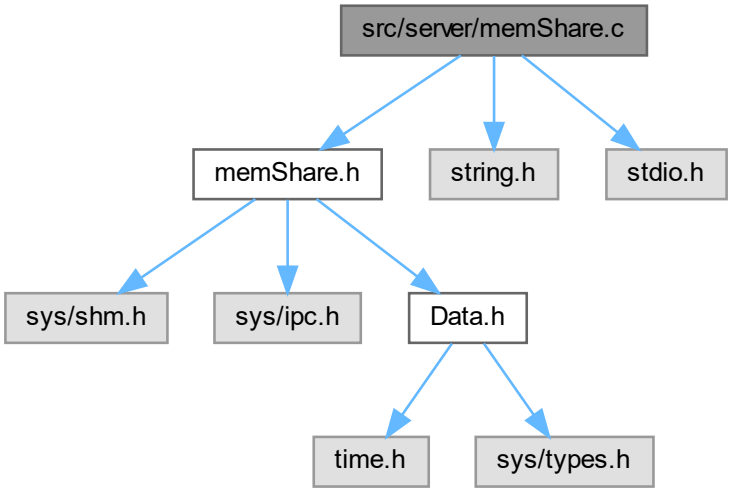
Go to the documentation of this file.

```
00001 #ifndef map_h
00002 #define map_h
00003
00041 // ------------------------------
00042 //        Hashing Math
00043 // ------------------------------
00044
00051 int hash_log2(int number_to_log);
00052
00062 int hash_string(int hash_table_capacity, char *string, int strlen);
00063
00070 int hash_upperLimit(int bitsize);
00071
00072 // -----------------------------------
00073 //        General Map Operations
00074 // -----------------------------------
00075
00081 struct _map_bucket
00082 {
00084     char *key;
00086     void *data;
00088     struct _map_bucket *next;
00089 };
00090
00100 typedef struct
00101 {
00103     int size;
00105     struct _map_bucket *buckets;
00106 } map;
00107
00111 typedef struct
00112 {
00114     short found;
00116     void *data;
00117 } map_result;
00118
00125 map *NewMap(int capacity);
00126
00134 void Map_Set(map *a_map, char *key, void *value);
00135
00142 map_result Map_Get(map *a_map, char *key);
00143
00151 map_result Map_Delete(map *a_map, char *key, short free_it);
00152
00153 #endif
```

8.19 src/server/memShare.c File Reference

```
#include "memShare.h"
#include <string.h>
```

`#include <stdio.h>`
Include dependency graph for memShare.c:



Functions

- int CreateSharedMemory ()
- int DestroySharedMemory ()
- void * GetMemoryPointer (int shared_mem_id)
- int ReleaseMemoryPointer (void *shmaddr)

8.20 memShare.c

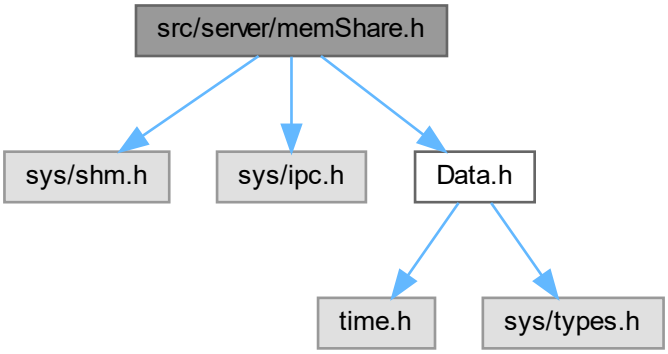
Go to the documentation of this file.

```
00001
00005 #include "memShare.h"
00006 #include <string.h>
00007 #include <stdio.h>
00008
00009 int CreateSharedMemory()
00010 {
00011     return shmget(MEM_KEY, MEM_SIZE, IPC_CREAT | MEM_PERMISSIONS);
00012 }
00013
00014 int DestroySharedMemory()
00015 {
00016     int shm_id = shmget(MEM_KEY, MEM_SIZE, 0);
00017     int control_result = shmctl(shm_id, IPC_RMID, 0);
00018     if (control_result != -1)
00019         return 0;
00020     return control_result;
00021 }
00022
00023 void *GetMemoryPointer(int shared_mem_id)
00024 {
00025     return shmat(shared_mem_id, NULL, 0);
00026 }
00027
00028 int ReleaseMemoryPointer(void *shmaddr)
00029 {
00030     return shmdt(shmaddr);
00031 }
```

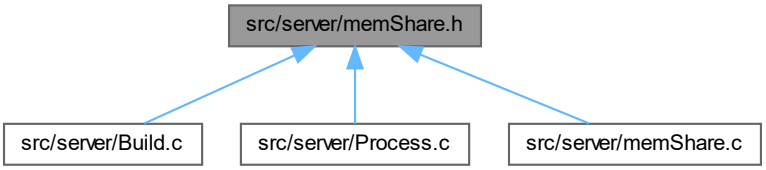
8.21 src/server/memShare.h File Reference

```
#include <sys/shm.h>
#include <sys/ipc.h>
#include "Data.h"
```

Include dependency graph for memShare.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define MEM_KEY 0x727
- #define MEM_PERMISSIONS 0664
- #define MEM_SIZE DATA_SIZE ∗DATA_NUM_RECORDS

Functions

- int CreateSharedMemory ()
- int DestroySharedMemory ()
- void ∗ GetMemoryPointer (int shared_mem_id)
- int ReleaseMemoryPointer (void ∗shmaddr)

8.22 memShare.h

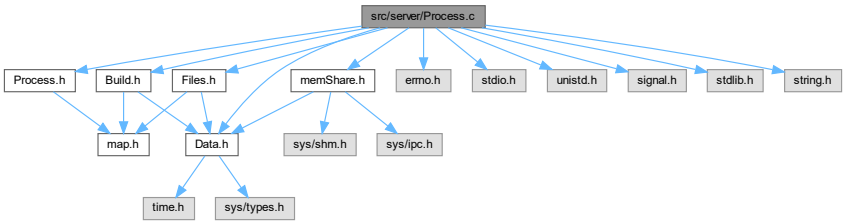
Go to the documentation of this file.

```
00001 #ifndef MEM_SHARE_H
00002 #define MEM_SHARE_H
00014 #include <sys/shm.h>
00015 #include <sys/ipc.h>
00016 #include "Data.h"
00017
00021 #define MEM_KEY 0x727
00022
00032 #define MEM_PERMISSIONS 0664
00033
00037 #define MEM_SIZE DATA_SIZE *DATA_NUM_RECORDS
00038
00044 int CreateSharedMemory();
00045
00051 int DestroySharedMemory();
00052
00062 void *GetMemoryPointer(int shared_mem_id);
00063
00069 int ReleaseMemoryPointer(void *shmaddr);
00073 #endif
```

8.23 src/server/Process.c File Reference

```
#include "Process.h"
#include "Files.h"
#include "Data.h"
#include "Build.h"
#include "memShare.h"
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
```

Include dependency graph for Process.c:



Functions

- int TerminateExistingServer ()
- int IndicateRereadNeeded ()
- int IndicateRereadDone ()
- short IsRereadNeeded ()
- void SignalHandle (int signo)
- int Initialize ()
- void Process (int shm_id)

- void HelpCommand ()
- void RunCommand ()

  Runs the server if it doesn't already exist.
- void StopCommand ()
- void ResetCommand ()
- void RunHeadless (char *processName)

Variables

- map * student_map
- map * initial_cumulative_times

  A map of userIDs to integer seconds. These values are subtracted from the current total cumulative time for each user to calculate their cumulative time since the server process started.
- short is_stopping = 0

8.24 Process.c

Go to the documentation of this file.

```
00001
00006 #include "Process.h"
00007 #include "Files.h"
00008 #include "Data.h"
00009 #include "Build.h"
00010 #include "memShare.h"
00011 #include <errno.h>
00012 #include <stdio.h>
00013 #include <unistd.h>
00014 #include <signal.h>
00015 #include <stdlib.h>
00016 #include <string.h>
00017
00018 map *student_map;
00020 map *initial_cumulative_times;
00021
00022 int TerminateExistingServer()
00023 {
00024     FILE *file = fopen(LOCKFILE, "r");
00025     if (file == NULL)
00026     {
00027         return -1;
00028     }
00029     int need_rewrite;
00030     int pid = 0;
00031     fscanf(file, "%d %d", &need_rewrite, &pid);
00032     fclose(file);
00033     if (pid > 0)
00034     {
00035         return kill(pid, SIGTERM);
00036     }
00037     return -2;
00038 }
00039
00040 int IndicateRereadNeeded()
00041 {
00042     FILE *file = fopen(LOCKFILE, "r+");
00043     if (file == NULL)
00044     {
00045         return -1;
00046     }
00047     int err = 0;
00048     err = fseek(file, 0, SEEK_SET);
00049     if (!err)
00050     {
00051         fputc('1', file);
00052     }
00053     err = fclose(file);
00054     return err;
00055 }
00056
00057 int IndicateRereadDone()
00058 {
```

```
00059     FILE *file = fopen(LOCKFILE, "r+");
00060     if (file == NULL)
00061     {
00062         return -1;
00063     }
00064     int err = 0;
00065     err = fseek(file, 0, SEEK_SET);
00066     if (!err)
00067     {
00068         fputc('0', file);
00069     }
00070     err = fclose(file);
00071     return err;
00072 }
00073
00074 short IsRereadNeeded()
00075 {
00076     FILE *file = fopen(LOCKFILE, "r");
00077     char firstc = fgetc(file);
00078     fclose(file);
00079     return firstc == '1';
00080 }
00081
00082 void SignalHandle(int signo)
00083 {
00084     printf("Received shutdown signal.\n");
00085     if (signo == SIGINT || signo == SIGTERM)
00086     {
00087         is_stopping = 1;
00088     }
00089     // possible feature: add a timeout terminate emergency exit (with graceful shutdown)
00090
00091 }
00092
00093 short is_stopping = 0;
00094
00095 // ~~~~~~~~~~~~~~~ CLI Commands ~~~~~~~~~~~~~~~~~~~~~~~~~
00096
00097 int Initialize()
00098 {
00099     int err;
00100     if (!FileExists(STATIC_USER_DATA_FILE))
00101     {
00102         printf("%s does not exist. Creating.\n", STATIC_USER_DATA_FILE);
00103         err = CreateInitialUserDataFile(STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00104         if (err)
00105         {
00106             printf("Problem creating %s!\n", STATIC_USER_DATA_FILE);
00107         }
00108     }
00109     if (!FileExists(STATIC_USER_CUMULATIVE_FILE))
00110     {
00111         printf("%s does not exist. Creating.\n", STATIC_USER_CUMULATIVE_FILE);
00112         err = CreateInitialCumulativeFile(STATIC_USER_CUMULATIVE_FILE);
00113         if (err)
00114         {
00115             printf("Problem creating %s!\n", STATIC_USER_CUMULATIVE_FILE);
00116         }
00117     }
00118     PopulateStudents(Data_IDs, Data_Names, DATA_NUM_RECORDS);
00119     student_map = NewMap(50);
00120     BuildStudentMap(student_map, students, DATA_NUM_RECORDS);
00121     err = FillStudentMapFromFile(student_map, STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00122     if (err)
00123     {
00124         printf("Problem filling student map from %s!\n", STATIC_USER_DATA_FILE);
00125     }
00126     printf("Student data retrieved from file.\n");
00127
00128     initial_cumulative_times = NewMap(50);
00129     err = ReadInitialCumulative(initial_cumulative_times, STATIC_USER_CUMULATIVE_FILE);
00130     if (err)
00131     {
00132         printf("Failed to read %s. Cumulative times may be wrong!", STATIC_USER_CUMULATIVE_FILE);
00133     }
00134
00135     dirty = 0;
00136
00137     int shmid = CreateSharedMemory();
00138     if (shmid == -1)
00139     {
00140         DestroySharedMemory();
00141         shmid = CreateSharedMemory();
00142     }
00143     printf("Shared memory allocated.\n");
00144     return shmid;
00145 }
```

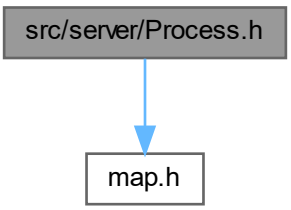
```
00146
00147 void Process(int shm_id)
00148 {
00149     if (IsRereadNeeded())
00150     {
00151         printf("\nReread indicated - rechecking user data file.");
00152         FillStudentMapFromFile(student_map, STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00153         IndicateRereadDone();
00154     }
00155     SetAllStudentsInactive(students, DATA_NUM_RECORDS);
00156     int err = PipeAcpToStudentMap(student_map);
00157     if (err)
00158     {
00159         printf("Error piping ac -p command! \n");
00160     }
00161     else
00162     {
00163         CalculateCumulative(students, DATA_NUM_RECORDS, initial_cumulative_times);
00164     }
00165     err = PipeWhoToStudentMap(student_map);
00166     if (err)
00167     {
00168         perror("Error updating from who!");
00169     }
00170     if (dirty)
00171     {
00172         err = WriteStudentArrayToFile(students, DATA_NUM_RECORDS, STATIC_USER_DATA_FILE);
00173         if (err)
00174         {
00175             printf("\nError updating %s!", STATIC_USER_DATA_FILE);
00176         }
00177         else
00178         {
00179             dirty = 0;
00180         }
00181     }
00182     void *ptr = GetMemoryPointer(shm_id);
00183     if (ptr == (void *)-1)
00184     {
00185         perror("Error attaching to shared memory");
00186     }
00187     else
00188     {
00189         WriteStudentsToMemory(ptr, students, DATA_NUM_RECORDS);
00190         ReleaseMemoryPointer(ptr);
00191     }
00192 }
00193
00194 void HelpCommand()
00195 {
00196     printf("\nUsage: server [OPTION]\n\n");
00197     printf("Options: \n");
00198     printf("\thelp\t\t\tShows the possible program commands\n");
00199     printf("\treset\t\t\tRegenerates the user data file\n");
00200     printf("\tstop\t\t\tStops an existing server process if it is running\n");
00201     printf("\trun\t\t\tCreates a new server with output to the shell if a server isn't already
    running.\n");
00202     printf("\theadless\t\tCreates a new headless server if a server isn't already running.\n\n");
00203 }
00204
00205 void RunCommand()
00206 {
00207     printf("\nRunning server.\n");
00208     if (DoesLockfileExist())
00209     {
00210         printf("\nServer is already running. Run 'server stop' to shut it down first.\n");
00211         return;
00212     }
00213     int err = CreateLockfile();
00214     if (err)
00215     {
00216         printf("\nFailed to create lockfile! Exiting.\n");
00217         return;
00218     }
00219     int shm_id = Initialize();
00220     signal(SIGTERM, SignalHandle);
00221     signal(SIGINT, SignalHandle);
00222     printf("Server started.\n");
00223     fflush(stdout);
00224     while (!is_stopping)
00225     {
00226         Process(shm_id);
00227         sleep(1);
00228     }
00229     printf("Server shutting down.\n");
00230     err = DeleteLockfile();
00231     if (err)
```

```
00232     {
00233         printf("Failed to delete lockfile!\n");
00234     }
00235     err = DestroySharedMemory();
00236     if (err)
00237     {
00238         printf("Failed to destroy shared memory!\n");
00239     }
00240     printf("Server terminated.\n");
00241 }
00242
00243 void StopCommand()
00244 {
00245     printf("\nStopping server...\n");
00246     int err = TerminateExistingServer();
00247     if (err != 0)
00248     {
00249         if (err == -1)
00250         {
00251             printf("Server isn't running.\n");
00252         }
00253         else if (err == -2)
00254         {
00255             printf("Lockfile did not contain a valid process id!\n");
00256         }
00257         else
00258         {
00259             printf("Sending terminate signal failed!\n");
00260         }
00261     }
00262     else
00263     {
00264         printf("Server terminated.\n");
00265     }
00266 }
00267
00268 void ResetCommand()
00269 {
00270     int err;
00271
00272     if (FileExists(STATIC_USER_DATA_FILE))
00273     {
00274         printf("User data file exists. Deleting...\n");
00275         remove(STATIC_USER_DATA_FILE);
00276     }
00277
00278     printf("Creating new data file.\n");
00279     err = CreateInitialUserDataFile(STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00280     if (err)
00281     {
00282         printf("Problem creating %s!\n", STATIC_USER_DATA_FILE);
00283     }
00284     else
00285     {
00286         printf("%s created.\n", STATIC_USER_DATA_FILE);
00287     }
00288
00289     printf("Creating new cumulative file.\n");
00290     err = CreateInitialCumulativeFile(STATIC_USER_CUMULATIVE_FILE);
00291     if (err)
00292     {
00293         printf("Problem creating %s!\n", STATIC_USER_CUMULATIVE_FILE);
00294     }
00295     else
00296     {
00297         printf("%s created.\n", STATIC_USER_CUMULATIVE_FILE);
00298     }
00299
00300     if (DoesLockfileExist())
00301     {
00302         printf("Indicated re-read to running server process.\n");
00303         IndicateRereadNeeded();
00304     }
00305 }
00306
00307 void RunHeadless(char *processName)
00308 {
00309     if (DoesLockfileExist())
00310     {
00311         printf("Server process already running.\n");
00312         return;
00313     }
00314     char commandFront[] = " nohup ";
00315     char commandEnd[] = " run & exit";
00316     size_t comm_length = strlen(commandFront) + strlen(commandEnd) + strlen(processName) + 1;
00317     char *commandFull = malloc(comm_length * sizeof(char));
00318     memset(commandFull, 0, comm_length * sizeof(char));
```

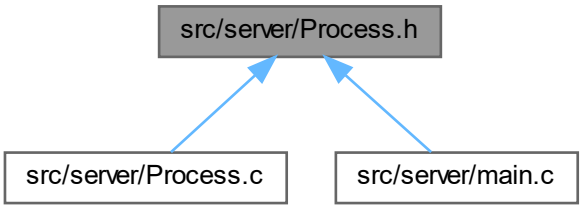
```
00319     strcpy(commandFull, commandFront);
00320     strcat(commandFull, processName);
00321     strcat(commandFull, commandEnd);
00322
00323     printf("Executing: %s\n", commandFull);
00324     popen(commandFull, "we");
00325     printf("Server running headlessly.\n");
00326 }
```

8.25 src/server/Process.h File Reference

`#include "map.h"`
Include dependency graph for Process.h:



This graph shows which files directly or indirectly include this file:



Functions

- int TerminateExistingServer ()
- int IndicateRereadNeeded ()
- int IndicateRereadDone ()
- short IsRereadNeeded ()
- void SignalHandle (int signo)
- int Initialize ()
- void Process (int shm_id)

- void ResetCommand ()
- void StopCommand ()
- void RunCommand ()

  Runs the server if it doesn't already exist.
- void HelpCommand ()
- void RunHeadless (char *processName)

Variables

- short is_stopping
- map * student_map

8.26 Process.h

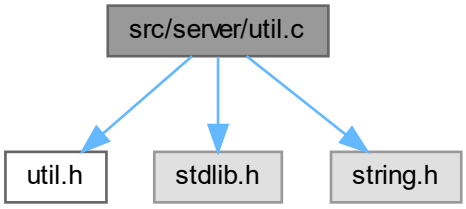
Go to the documentation of this file.

```
00001 #ifndef Process_h
00002 #define Process_h
00010 #include "map.h"
00011
00020 int TerminateExistingServer();
00021
00029 int IndicateRereadNeeded();
00030
00037 int IndicateRereadDone();
00038
00045 short IsRereadNeeded();
00046
00052 void SignalHandle(int signo);
00053
00057 extern short is_stopping;
00058
00059 // ~~~~~~~~~~~~~~~ CLI Commands ~~~~~~~~~~~~~~~~~~~~~~~~~
00060
00061 extern map *student_map;
00062
00073 int Initialize();
00074
00087 void Process(int shm_id);
00088
00095 void ResetCommand();
00096
00102 void StopCommand();
00103
00108 void RunCommand();
00109
00116 void HelpCommand();
00117
00122 void RunHeadless(char *processName);
00127 #endif
```

8.27 src/server/util.c File Reference

```
#include "util.h"
#include <stdlib.h>
```

`#include <string.h>`
Include dependency graph for util.c:



Functions

- int RandomInteger (int min, int max)
- float RandomFloat (float min, float max)
- short RandomFlag (float percentage_chance)
- void Trim (char *string)

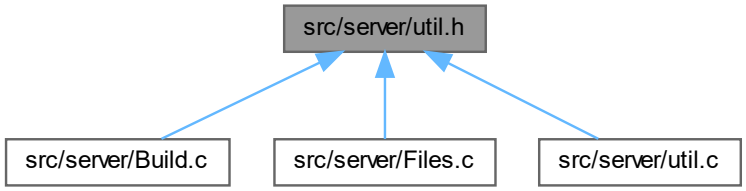
8.28 util.c

Go to the documentation of this file.

```
00001
00005 #include "util.h"
00006
00007 #include <stdlib.h>
00008 #include <string.h>
00009
00010 int RandomInteger(int min, int max)
00011 {
00012     int r_add = rand() % (max - min + 1);
00013     return r_add + min;
00014 }
00015
00016 float RandomFloat(float min, float max)
00017 {
00018     float dif = max - min;
00019     int rand_int = rand() % (int)(dif * 10000);
00020     return min + (float)rand_int / 10000.0;
00021 }
00022
00023 short RandomFlag(float percentage_chance)
00024 {
00025     float random_value = (float)rand() / RAND_MAX;
00026     if (random_value < percentage_chance)
00027     {
00028         return 1;
00029     }
00030     return 0;
00031 }
00032
00033 void Trim(char * string)
00034 {
00035     size_t len = strlen(string);
00036     int i;
00037     for(i = 0; i < len; i++)
00038     {
00039         if(string[i] == ' ' || string[i] == '\t' || string[i] == '\n')
00040         {
00041             string[i] = '\0';
00042             break;
00043         }
00044     }
00045 }
00046
```

8.29 src/server/util.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- int RandomInteger (int min, int max)
- float RandomFloat (float min, float max)
- short RandomFlag (float percentage_chance)
- void Trim (char ∗string)

8.30 util.h

Go to the documentation of this file.

```
00001 #ifndef util_h
00002 #define util_h
00017 int RandomInteger(int min, int max);
00018
00025 float RandomFloat(float min, float max);
00026
00033 short RandomFlag(float percentage_chance);
00034
00039 void Trim(char * string);
00044 #endif
```

Index