Quash Shell

0.1

Generated by Doxygen 1.8.11

Contents

Chapter 1

Main Page

EECS 678 - Project 1 - Quash Shell

Introduction

In this assignment, you must complete the quash shell. A simple skeleton has been provided, but lacks any of the core functionality.

Installation

To build Quash and this documentation in HTML use:

```
make
```

To only build Quash use:

```
make quash
```

To generate this documentation in HTML use:

```
make doc
```

To clean quash use:

```
make clean
```

Usage

To run Quash use:

```
./quash
```

or

```
make test
```

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

CDCommand
: Command to change directories . . . . . ??

Command
: Make all command types the same size and interchangeable . . . . . ??

CommandHolder
: Contains information about the properties of the command . . . . . ??

Example
: A data structure generated by IMPLEMENT_DEQUE_STRUCT() to store the state of a deque ??

ExportCommand
: Command to set environment variables . . . . . ??

GenericCommand
: Commands that take any number of arguments and are not built into Quash . . . . . ??

KillCommand
: Command to kill a process based on it's job id . . . . . ??

MemoryPool
: Holds a block of memory that can be used for allocations . . . . . ??

QuashState
: Holds information about the state and environment Quash is running in . . . . . ??

Redirect
: Intermediate parsing structure used to determine the final configuration of the redirects in a command . . . . . ??

SimpleCommand
: A command which has no arguments . . . . . ??

Chapter 3

File Index

3.1 File List

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

src/command.c
: Implements functions used to generate and manage commands . . . . . ??

src/command.h
: Command structures and functions for defining and managing commands . . . . . ??

src/debug.h
: This file holds useful macros for debugging purposes . . . . . ??

src/deque.h
: Double ended queue generators specialized to any given type . . . . . ??

src/execute.c
: Implements interface functions between Quash and the environment and functions that interpret an execute commands . . . . . ??

src/execute.h
: Functions for interpreting and running commands . . . . . ??

src/quash.c . . . . . ??

src/quash.h . . . . . ??

src/parsing/memory_pool.h
: An abstraction of malloc that allows for all allocations in the pool to be free'd with a single call to destroy_memory_pool(). Allocations to the memory pool should NOT be manually free'd with a call to free() . . . . . ??

src/parsing/parsing_interface.h
: Defines an interface between c and the parser. This file is also responsible for defining many of the structures used by the parser . . . . . ??

Chapter 4

Class Documentation

4.1 CDCommand Struct Reference

Command to change directories.

`#include <command.h>`

Public Attributes

- CommandType type
- char * dir

4.1.1 Detailed Description

Command to change directories.

See also

realpath(), Command

4.1.2 Member Data Documentation

4.1.2.1 char* CDCommand::dir

Path to the directory we wish to change to

4.1.2.2 CommandType CDCommand::type

Type of command

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

- src/command.h

4.2 Command Union Reference

Make all command types the same size and interchangeable.

`#include <command.h>`

Public Attributes

- SimpleCommand simple
- GenericCommand generic
- EchoCommand echo
- ExportCommand export
- CDCommand cd
- KillCommand kill
- PWDCommand pwd
- JobsCommand jobs
- ExitCommand exit
- EOCCommand eoc

4.2.1 Detailed Description

Make all command types the same size and interchangeable.

This is useful for arrays or making a common entry point for all command types. The exact type information can be recovered later with the *get_command_type()* function.

See also

get_command_type, SimpleCommand, GenericCommand, EchoCommand, ExportCommand, CDCommand, KillCommand, PWDCommand, JobsCommand, ExitCommand, EOCCommand

4.2.2 Member Data Documentation

4.2.2.1 CDCommand Command::cd

Read structure as a *CDCommand*

4.2.2.2 EchoCommand Command::echo

Read structure as a *ExportCommand*

4.2.2.3 EOCCommand Command::eoc

Read structure as a *EOCCommand*

4.2.2.4 ExitCommand Command::exit

Read structure as a *ExitCommand*

4.2.2.5 ExportCommand Command::export

Read structure as a *ExportCommand*

4.2.2.6 GenericCommand Command::generic

Read structure as a *GenericCommand*

4.2.2.7 JobsCommand Command::jobs

Read structure as a *JobsCommand*

4.2.2.8 KillCommand Command::kill

Read structure as a *KillCommand*

4.2.2.9 PWDCommand Command::pwd

Read structure as a *PWDCommand*

4.2.2.10 SimpleCommand Command::simple

Read structure as a *SimpleCommand*

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

- src/command.h

4.3 CommandHolder Struct Reference

Contains information about the properties of the command.

`#include <command.h>`

Public Attributes

- char * redirect_in
- char * redirect_out
- char flags
- Command cmd

4.3.1 Detailed Description

Contains information about the properties of the command.

See also

REDIRECT_IN, REDIRECT_OUT, REDIRECT_APPEND, PIPE_IN, PIPE_OUT, BACKGROUND, Command

4.3.2 Member Data Documentation

4.3.2.1 Command CommandHolder::cmd

A *Command* to hold

4.3.2.2 char CommandHolder::flags

A set of bits that hold information about how to execute the command. The properties can be extracted from the flags field by using a bit-wise & (i.e. `flags` & *PIPE_IN*) are macro defined as:

- *REDIRECT_IN*
- *REDIRECT_OUT*
- *REDIRECT_APPEND*
- *PIPE_IN*
- *PIPE_OUT*
- *BACKGROUND*

4.3.2.3 char* CommandHolder::redirect_in

Redirect standard in of this command to a file name *redirect_in*

4.3.2.4 char* CommandHolder::redirect_out

Redirect standard out of this command to a file name *redirect_out*

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

- src/command.h

4.4 Example Struct Reference

A data structure generated by IMPLEMENT_DEQUE_STRUCT() to store the state of a deque.

```
#include <deque.h>
```

Public Attributes

- Type * data
- size_t cap
- size_t front
- size_t back
- void(* destructor)(Type)

4.4.1 Detailed Description

A data structure generated by IMPLEMENT_DEQUE_STRUCT() to store the state of a deque.

Note

The members of this struct should not be accessed or modified directly. For modification please use the functions generated by the IMPLEMENT_DEQUE() macro.

See also

IMPLEMENT_DEQUE()

4.4.2 Member Data Documentation

4.4.2.1 size_t Example::back

The index one greater than the last element of the queue

4.4.2.2 size_t Example::cap

The current capacity of the deque

4.4.2.3 Type* Example::data

The array holding the deque

4.4.2.4 void(* Example::destructor) (Type)

Optional destructor function pointer for the data type. This is called on every element in the queue when *destroy_Example()* is called.

Note

We will have a lab over function pointers later in the semester. If this doesn't make sense now then skip it or come to your TA for assistance if you really want to use it.

4.4.2.5 size_t Example::front

The index of the element at the front of the deque

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

- src/deque.h

4.5 ExportCommand Struct Reference

Command to set environment variables.

`#include <command.h>`

Public Attributes

- CommandType type
- char * env_var
- char * val

4.5.1 Detailed Description

Command to set environment variables.

See also

CommandType, lookup_env(), write_env(), Command

4.5.2 Member Data Documentation

4.5.2.1 char* ExportCommand::env_var

Name of environment variable to set

4.5.2.2 CommandType ExportCommand::type

Type of command

4.5.2.3 char* ExportCommand::val

String that should be stored in *env_var* environment variable

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

- src/command.h

4.6 GenericCommand Struct Reference

Commands that take any number of arguments and are not built into Quash.

`#include <command.h>`

Public Attributes

- CommandType type
- char ** args

4.6.1 Detailed Description

Commands that take any number of arguments and are not built into Quash.

See also

Command

4.6.2 Member Data Documentation

4.6.2.1 char** GenericCommand::args

A NULL terminated array of c-strings ready to pass to *exec* functions

4.6.2.2 CommandType GenericCommand::type

Type of command

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

- src/command.h

4.7 KillCommand Struct Reference

Command to kill a process based on it's job id.

`#include <command.h>`

Public Attributes

- CommandType type
- int sig
- int job
- char * sig_str
- char * job_str

4.7.1 Detailed Description

Command to kill a process based on it's job id.

See also

CommandType, Command

4.7.2 Member Data Documentation

4.7.2.1 int KillCommand::job

Job id number

4.7.2.2 char* KillCommand::job_str

String holding the job id number (used for printing)

4.7.2.3 int KillCommand::sig

Signal to send to the job

4.7.2.4 char* KillCommand::sig_str

String holding the signal number (used for printing)

4.7.2.5 CommandType KillCommand::type

Type of command

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

- src/command.h

4.8 MemoryPool Struct Reference

Holds a block of memory that can be used for allocations.

Public Attributes

- void * pool
- size_t size
- void * next

4.8.1 Detailed Description

Holds a block of memory that can be used for allocations.

The advantage of a MemoryPool is that we can free the entire block at once. This can help prevent memory leaks in code that creates structures from the bottom up (such as the parser) and can fail leaving the top most portion of the structure uninitialized.

Note

A single memory pool can run out of space fast or will take up quite a bit of unnecessary space in memory. To combat this problem the *MemoryPoolDeque* allows the creation of a larger additional MemoryPool to handle the allocation.

4.8.2 Member Data Documentation

4.8.2.1 void* MemoryPool::next

The next pointer to be returned from an allocation

4.8.2.2 void* MemoryPool::pool

Pointer to the top of the memory pool

4.8.2.3 size_t MemoryPool::size

Size of the memory pool in bytes

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

- src/parsing/memory_pool.c

4.9 QuashState Struct Reference

Holds information about the state and environment Quash is running in.

`#include <quash.h>`

Public Attributes

- bool running
- bool is_a_tty
- char * parsed_str

4.9.1 Detailed Description

Holds information about the state and environment Quash is running in.

4.9.2 Member Data Documentation

4.9.2.1 bool QuashState::is_a_tty

Indicates if the shell is receiving input from a file or the command line

4.9.2.2 char* QuashState::parsed_str

Holds a string representing the parsed structure of the command input from the command line

4.9.2.3 bool QuashState::running

Indicates if Quash should keep accept more input

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

- src/quash.h

4.10 Redirect Struct Reference

Intermediate parsing structure used to determine the final configuration of the redirects in a command.

`#include <parsing_interface.h>`

Public Attributes

- char * in
- char * out
- bool append

4.10.1 Detailed Description

Intermediate parsing structure used to determine the final configuration of the redirects in a command.

4.10.2 Member Data Documentation

4.10.2.1 bool Redirect::append

Flag indicating that the redirect out should actually append to the end of a file rather than truncating it

4.10.2.2 char* Redirect::in

File name for redirect in.

4.10.2.3 char* Redirect::out

File name for redirect out.

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

- src/parsing/parsing_interface.h

4.11 SimpleCommand Struct Reference

A command which has no arguments.

`#include <command.h>`

Public Attributes

- CommandType type

4.11.1 Detailed Description

A command which has no arguments.

All command structures can be correctly read as a *SimpleCommand* since they all have the *CommandType* field as the first field.

See also

Command, CommandType

4.11.2 Member Data Documentation

4.11.2.1 CommandType SimpleCommand::type

Type of command

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

- src/command.h

Chapter 5

File Documentation

5.1 src/command.c File Reference

Implements functions used to generate and manage commands.

```
#include "command.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

Functions

- CommandHolder mk_command_holder (char *redirect_in, char *redirect_out, char flags, Command cmd)
  Create a CommandHolder structure and return a copy.
- Command mk_generic_command (char **args)
  Create a GenericCommand structure and return a copy.
- Command mk_echo_command (char **strs)
  Create a EchoCommand structure and return a copy.
- Command mk_export_command (char *env_var, char *val)
  Create a ExportCommand structure and return a copy.
- Command mk_cd_command (char *dir)
  Create a CDCommand structure and return a copy.
- Command mk_kill_command (char *sig, char *job)
  Create a KillCommand structure and return a copy.
- Command mk_pwd_command ()
  Create a PWDCommand structure and return a copy.
- Command mk_jobs_command ()
  Create a JobsCommand structure and return a copy.
- Command mk_exit_command ()
  Create a ExitCommand structure and return a copy.
- Command mk_eoc ()
  Create a EOCCommand structure and return a copy.
- CommandType get_command_type (Command cmd)
  Get the type of the command.
- CommandType get_command_holder_type (CommandHolder holder)
  Get the type of the Command in the CommandHolder.
- void debug_print_script (const CommandHolder *holders)
  Print all commands in the script with print_command()

5.1.1 Detailed Description

Implements functions used to generate and manage commands.

5.1.2 Function Documentation

5.1.2.1 void debug_print_script (const CommandHolder * *holders*)

Print all commands in the script with *print_command()*

Note

This only works when the *DEBUG* macro is defined

Parameters

holders	*CommandHolder* array to print

See also

CommandHolder

5.1.2.2 CommandType get_command_holder_type (CommandHolder *holder*)

Get the type of the *Command* in the *CommandHolder*.

Uses the property that all Command variants can be cast to *SimpleCommand* to extract the *CommandType* of the Command.

Parameters

holder	CommandHolder from which this function extracts the *CommandType*

Returns

The resulting CommandType of the `cmd` parameter

See also

CommandType, CommandHolder, SimpleCommand

5.1.2.3 CommandType get_command_type (Command *cmd*)

Get the type of the command.

Uses the property that all *Command* variants can be cast to *SimpleCommand* to extract the *CommandType* of the Command.

Parameters

cmd	Command from which this function extracts the *CommandType*

Returns

The resulting *CommandType* of the `cmd` parameter

See also

CommandType, Command, SimpleCommand

5.1.2.4 Command mk_cd_command (char * *dir*)

Create a *CDCommand* structure and return a copy.

Parameters

dir	Path to the directory we wish to change to

Returns

Copy of constructed CDCommand as a *Command*

See also

realpath(), Command, CDCommand

5.1.2.5 CommandHolder mk_command_holder (char * *redirect_in*, char * *redirect_out*, char *flags*, Command *cmd*)

Create a *CommandHolder* structure and return a copy.

Parameters

redirect_in	If the *REDIRECT_IN* flag is set, Quash should redirect the standard input stream of the command to read from a file stored in this string
redirect_out	If the *REDIRECT_OUT* flag is set, Quash should redirect the standard output stream of the command to write to a file stored in this string
flags	A set of bits that hold information about how to execute the command. The properties can be extracted from the flags field by using a bit-wise & (i.e. `flags` & *PIPE_IN*) are macro defined as: • *REDIRECT_IN* • *REDIRECT_OUT* • *REDIRECT_APPEND* • *PIPE_IN* • *PIPE_OUT* • *BACKGROUND*
cmd	The *Command* the CommandHolder should copy and hold on to

Returns

Copy of constructed CommandHolder

See also

CommandType, REDIRECT_IN, REDIRECT_OUT, REDIRECT_APPEND, PIPE_IN, PIPE_OUT, BACKGROUND, Command, CommandHolder

5.1.2.6 Command mk_echo_command (char ** *args*)

Create a *EchoCommand* structure and return a copy.

Parameters

args	A NULL terminated array of strings containing the strings passed to echo

Returns

Copy of constructed EchoCommand as a *Command*

See also

Command, EchoCommand

5.1.2.7 Command mk_eoc ( )

Create a *EOCCommand* structure and return a copy.

Returns

Copy of constructed EOCCommand as a *Command*

See also

Command, EOCCommand

5.1.2.8 Command mk_exit_command ( )

Create a *ExitCommand* structure and return a copy.

Returns

Copy of constructed ExitCommand as a *Command*

See also

Command, ExitCommand

5.1.2.9 Command mk_export_command (char * *env_var*, char * *val*)

Create a *ExportCommand* structure and return a copy.

Parameters

env_var	Name of environment variable to set
val	String that should be stored in *env_var* environment variable

Returns

Copy of constructed ExportCommand as a *Command*

See also

lookup_env(), write_env(), Command, ExportCommand

5.1.2.10 Command mk_generic_command (char ** *args*)

Create a *GenericCommand* structure and return a copy.

Parameters

args	A NULL terminated array of strings ready to pass to the exec family of functions

Returns

Copy of constructed GenericCommand as a *Command*

See also

Command, GenericCommand

5.1.2.11 Command mk_jobs_command ( )

Create a *JobsCommand* structure and return a copy.

Returns

Copy of constructed JobsCommand as a *Command*

See also

Command, JobsCommand

5.1.2.12 Command mk_kill_command (char * *sig*, char * *job*)

Create a *KillCommand* structure and return a copy.

Parameters

sig	Signal to send to the job
job	Job id number

Returns

Copy of constructed KillCommand as a *Command*

See also

Command, KillCommand

5.1.2.13 Command mk_pwd_command ( )

Create a *PWDCommand* structure and return a copy.

Returns

Copy of constructed PWDCommand as a *Command*

See also

Command, PWDCommand

5.2 src/command.h File Reference

Command structures and functions for defining and managing commands.

```
#include <stdbool.h>
```

Classes

- struct SimpleCommand

  A command which has no arguments.
- struct GenericCommand

  Commands that take any number of arguments and are not built into Quash.
- struct ExportCommand

  Command to set environment variables.
- struct CDCommand

  Command to change directories.
- struct KillCommand

  Command to kill a process based on it's job id.
- union Command

  Make all command types the same size and interchangeable.
- struct CommandHolder

  Contains information about the properties of the command.

Macros

- #define REDIRECT_IN (0x01)

  Flag bit indicating whether a GenericCommand should read from standard in.
- #define REDIRECT_OUT (0x04)

  Flag bit indicating whether a GenericCommand should write to standard out truncating the original file.
- #define REDIRECT_APPEND (0x08)

  Flag bit indicating whether a GenericCommand should write to standard out appending its output.
- #define PIPE_IN (0x10)

  Flag bit indicating whether a GenericCommand should read from a pipe.
- #define PIPE_OUT (0x20)

  Flag bit indicating whether a GenericCommand should write to a pipe.
- #define BACKGROUND (0x40)

  Flag bit indicating whether a GenericCommand should be run in the background.

Typedefs

- typedef enum CommandType CommandType

  All possible types of commands.
- typedef struct SimpleCommand SimpleCommand

  A command which has no arguments.
- typedef struct GenericCommand GenericCommand

  Commands that take any number of arguments and are not built into Quash.
- typedef GenericCommand EchoCommand

  Alias for GenericCommand to denote a command to print strings.
- typedef struct ExportCommand ExportCommand

  Command to set environment variables.
- typedef struct CDCommand CDCommand

  Command to change directories.
- typedef struct KillCommand KillCommand

  Command to kill a process based on it's job id.
- typedef SimpleCommand PWDCommand

  Alias for SimpleCommand to denote a print working directory command.
- typedef SimpleCommand JobsCommand

  Alias for SimpleCommand to denote a print jobs list.
- typedef SimpleCommand ExitCommand

  Alias for SimpleCommand to denote a termination of the program.
- typedef SimpleCommand EOCCommand

  Alias for SimpleCommand to denote the end of a command.
- typedef union Command Command

  Make all command types the same size and interchangeable.
- typedef struct CommandHolder CommandHolder

  Contains information about the properties of the command.

Enumerations

- enum CommandType {
  EOC = 0, **GENERIC**, **ECHO**, **EXPORT**,
  KILL, **CD**, **PWD**, **JOBS**,
  EXIT }

  All possible types of commands.

Functions

- CommandHolder mk_command_holder (char *redirect_in, char *redirect_out, char flags, Command cmd)

  Create a CommandHolder structure and return a copy.
- Command mk_generic_command (char **args)

  Create a GenericCommand structure and return a copy.
- Command mk_echo_command (char **args)

  Create a EchoCommand structure and return a copy.
- Command mk_export_command (char *env_var, char *val)

  Create a ExportCommand structure and return a copy.
- Command mk_cd_command (char *dir)

  Create a CDCommand structure and return a copy.
- Command mk_kill_command (char *sig, char *job)

  Create a KillCommand structure and return a copy.
- Command mk_pwd_command ()

  Create a PWDCommand structure and return a copy.
- Command mk_jobs_command ()

  Create a JobsCommand structure and return a copy.
- Command mk_exit_command ()

  Create a ExitCommand structure and return a copy.
- Command mk_eoc ()

  Create a EOCCommand structure and return a copy.
- CommandType get_command_type (Command cmd)

  Get the type of the command.
- CommandType get_command_holder_type (CommandHolder holder)

  Get the type of the Command in the CommandHolder.
- void debug_print_script (const CommandHolder *holders)

  Print all commands in the script with print_command()

5.2.1 Detailed Description

Command structures and functions for defining and managing commands.

5.2.2 Typedef Documentation

5.2.2.1 typedef struct CDCommand CDCommand

Command to change directories.

See also

realpath(), Command

5.2.2.2 typedef union Command Command

Make all command types the same size and interchangeable.

This is useful for arrays or making a common entry point for all command types. The exact type information can be recovered later with the *get_command_type()* function.

See also

get_command_type, SimpleCommand, GenericCommand, EchoCommand, ExportCommand, CDCommand, KillCommand, PWDCommand, JobsCommand, ExitCommand, EOCCommand

5.2.2.3 typedef struct CommandHolder CommandHolder

Contains information about the properties of the command.

See also

REDIRECT_IN, REDIRECT_OUT, REDIRECT_APPEND, PIPE_IN, PIPE_OUT, BACKGROUND, Command

5.2.2.4 typedef enum CommandType CommandType

All possible types of commands.

These are useful for recovering what the actual type of a command is if the command is placed in a *Command* union

See also

Command

5.2.2.5 typedef GenericCommand EchoCommand

Alias for *GenericCommand* to denote a command to print strings.

Note

EchoCommand is similar to a generic command but is a builtin command

See also

GenericCommand, Command

5.2.2.6 typedef SimpleCommand EOCCommand

Alias for *SimpleCommand* to denote the end of a command.

See also

SimpleCommand, Command

5.2.2.7 typedef SimpleCommand ExitCommand

Alias for *SimpleCommand* to denote a termination of the program.

See also

end_main_loop(), SimpleCommand, Command

5.2.2.8 typedef struct ExportCommand ExportCommand

Command to set environment variables.

See also

CommandType, lookup_env(), write_env(), Command

5.2.2.9 typedef struct GenericCommand GenericCommand

Commands that take any number of arguments and are not built into Quash.

See also

Command

5.2.2.10 typedef SimpleCommand JobsCommand

Alias for *SimpleCommand* to denote a print jobs list.

See also

SimpleCommand, Command, Job

5.2.2.11 typedef struct KillCommand KillCommand

Command to kill a process based on it's job id.

See also

CommandType, Command

5.2.2.12 typedef SimpleCommand PWDCommand

Alias for *SimpleCommand* to denote a print working directory command.

See also

SimpleCommand, Command

5.2.2.13 typedef struct SimpleCommand SimpleCommand

A command which has no arguments.

All command structures can be correctly read as a *SimpleCommand* since they all have the *CommandType* field as the first field.

See also

Command, CommandType

5.2.3 Enumeration Type Documentation

5.2.3.1 enum CommandType

All possible types of commands.

These are useful for recovering what the actual type of a command is if the command is placed in a *Command* union

See also

Command

5.2.4 Function Documentation

5.2.4.1 void debug_print_script (const CommandHolder * *holders*)

Print all commands in the script with *print_command()*

Note

This only works when the *DEBUG* macro is defined

Parameters

holders	*CommandHolder* array to print

See also

CommandHolder

5.2.4.2 CommandType get_command_holder_type (CommandHolder *holder*)

Get the type of the *Command* in the *CommandHolder*.

Uses the property that all Command variants can be cast to *SimpleCommand* to extract the *CommandType* of the Command.

Parameters

holder	CommandHolder from which this function extracts the *CommandType*

Returns

The resulting CommandType of the `cmd` parameter

See also

CommandType, CommandHolder, SimpleCommand

5.2.4.3 CommandType get_command_type (Command *cmd*)

Get the type of the command.

Uses the property that all *Command* variants can be cast to *SimpleCommand* to extract the *CommandType* of the Command.

Parameters

cmd	Command from which this function extracts the *CommandType*

Returns

The resulting *CommandType* of the `cmd` parameter

See also

CommandType, Command, SimpleCommand

5.2.4.4 Command mk_cd_command (char * *dir*)

Create a *CDCommand* structure and return a copy.

Parameters

dir	Path to the directory we wish to change to

Returns

Copy of constructed CDCommand as a *Command*

See also

realpath(), Command, CDCommand

5.2.4.5 CommandHolder mk_command_holder (char * *redirect_in*, char * *redirect_out*, char *flags*, Command *cmd*)

Create a *CommandHolder* structure and return a copy.

Parameters

redirect_in	If the *REDIRECT_IN* flag is set, Quash should redirect the standard input stream of the command to read from a file stored in this string
redirect_out	If the *REDIRECT_OUT* flag is set, Quash should redirect the standard output stream of the command to write to a file stored in this string
flags	A set of bits that hold information about how to execute the command. The properties can be extracted from the flags field by using a bit-wise & (i.e. `flags` & *PIPE_IN*) are macro defined as: • *REDIRECT_IN* • *REDIRECT_OUT* • *REDIRECT_APPEND* • *PIPE_IN* • *PIPE_OUT* • *BACKGROUND*
cmd	The *Command* the CommandHolder should copy and hold on to

Returns

Copy of constructed CommandHolder

See also

CommandType, REDIRECT_IN, REDIRECT_OUT, REDIRECT_APPEND, PIPE_IN, PIPE_OUT, BACKGROUND, Command, CommandHolder

5.2.4.6 Command mk_echo_command (char ** *args*)

Create a *EchoCommand* structure and return a copy.

Parameters

args	A NULL terminated array of strings containing the strings passed to echo

Returns

Copy of constructed EchoCommand as a *Command*

See also

Command, EchoCommand

5.2.4.7 Command mk_eoc ( )

Create a *EOCCommand* structure and return a copy.

Returns

Copy of constructed EOCCommand as a *Command*

See also

Command, EOCCommand

5.2.4.8 Command mk_exit_command ( )

Create a *ExitCommand* structure and return a copy.

Returns

Copy of constructed ExitCommand as a *Command*

See also

Command, ExitCommand

5.2.4.9 Command mk_export_command (char * *env_var*, char * *val*)

Create a *ExportCommand* structure and return a copy.

Parameters

env_var	Name of environment variable to set
val	String that should be stored in *env_var* environment variable

Returns

Copy of constructed ExportCommand as a *Command*

See also

lookup_env(), write_env(), Command, ExportCommand

5.2.4.10 Command mk_generic_command (char ** *args*)

Create a *GenericCommand* structure and return a copy.

Parameters

args	A NULL terminated array of strings ready to pass to the exec family of functions

Returns

Copy of constructed GenericCommand as a *Command*

See also

Command, GenericCommand

5.2.4.11 Command mk_jobs_command ( )

Create a *JobsCommand* structure and return a copy.

Returns

Copy of constructed JobsCommand as a *Command*

See also

Command, JobsCommand

5.2.4.12 Command mk_kill_command (char * *sig*, char * *job*)

Create a *KillCommand* structure and return a copy.

Parameters

sig	Signal to send to the job
job	Job id number

Returns

Copy of constructed KillCommand as a *Command*

See also

Command, KillCommand

5.2.4.13 Command mk_pwd_command ( )

Create a *PWDCommand* structure and return a copy.

Returns

Copy of constructed PWDCommand as a *Command*

See also

Command, PWDCommand

5.3 src/debug.h File Reference

This file holds useful macros for debugging purposes.

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

Macros

- #define PRINT_DEBUG(fmt, ...)

  Print statement that only triggers if DEBUG is defined.
- #define IFDEBUG(x)

  Insert code that is only compiled if DEBUG is defined.

5.3.1 Detailed Description

This file holds useful macros for debugging purposes.

5.3.2 Macro Definition Documentation

5.3.2.1 #define IFDEBUG(*x*)

Insert code that is only compiled if DEBUG is defined.

Parameters

x	Code to insert

5.3.2.2 #define PRINT_DEBUG(*fmt, ...*)

Print statement that only triggers if DEBUG is defined.

Parameters

fmt	Format for the fprintf statement. This format is appended to the information refering to the location in the code the macro is expanded at.
...	Arguments to be substituted into the format string

5.4 src/deque.h File Reference

Double ended queue generators specialized to any given type.

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

Classes

- struct Example

  A data structure generated by IMPLEMENT_DEQUE_STRUCT() to store the state of a deque.

Macros

- #define IMPLEMENT_DEQUE_STRUCT(struct_name, type)

  Generates a structure for use with Double Ended Queues.
- #define PROTOTYPE_DEQUE(struct_name, type)

  Generates prototypes for functions that manipulate Double Ended Queue structures.
- #define IMPLEMENT_DEQUE(struct_name, type)

  Generates a malloc based set of functions for use with a structure generated by IMPLEMENT_DEQUE_STRUCT()

Typedefs

- typedef char Type

  An example type used for example purposes only.
- typedef struct Example Example

  This way you do not have to type "struct Example" each time you wish to refer to an Example structure.

Functions

- Example new_Example (size_t)

  Create a new, fully initialized deque structure.
- Example new_destructable_Example (size_t, void(∗)(Type))

  Create a new, fully initialized deque structure with a destructor that is applied to every element when the destroy_Example() function is called.
- void destroy_Example (Example ∗)

  Destroy the deque structure freeing memory if necessary.
- void empty_Example (Example ∗)

  Quickly empties the deque structure.
- bool is_empty_Example (Example ∗)

  Checks if the deque is empty.
- size_t length_Example (Example ∗)

  Query the number of elements in the deque.
- Type ∗ as_array_Example (Example ∗, size_t ∗)

  Extract an array based off the deque.
- void apply_Example (Example ∗, void(∗)(Type))

  Calls the function func on every element in the deque.
- void push_front_Example (Example ∗, Type)

  Insert an element to the front of the deque.
- void push_back_Example (Example ∗, Type)

  Insert an element to the back of the deque.
- Type pop_front_Example (Example ∗)

  Remove an element from the front of the deque.
- Type pop_back_Example (Example ∗)

  Remove an element from the back of the deque.
- Type peek_front_Example (Example ∗)

  Get a copy of the element at the front of the deque.
- Type peek_back_Example (Example ∗)

  Remove an element from the back of the deque.
- void update_front_Example (Example ∗, Type)

  Change the element at the front of the deque to be a copy of element.
- void update_back_Example (Example ∗, Type)

  Change the element at the front of the deque to be a copy of element.

5.4.1 Detailed Description

Double ended queue generators specialized to any given type.

5.4.2 Macro Definition Documentation

5.4.2.1 #define IMPLEMENT_DEQUE(*struct_name*, *type*)

Generates a *malloc* based set of functions for use with a structure generated by *IMPLEMENT_DEQUE_STRUCT()*

Parameters

struct_name	The name of the structure
type	The name of the type of elements stored in the *struct_name* structure

See also

IMPLEMENT_DEQUE_STRUCT(), PROTOTYPE_DEQUE()

5.4.2.2 #define IMPLEMENT_DEQUE_STRUCT(*struct_name*, *type*)

Value:

```
typedef struct struct_name {                                        \
    type* data;                                                      \
    size_t cap;                                                      \
    size_t front;                                                    \
    size_t back;                                                     \
                                                                     \
    void (*destructor)(type);                                        \
  } struct_name;
```

Generates a structure for use with Double Ended Queues.

Follow this call with either *PROTOTYPE_DEQUE()* (if in a header file) or *IMPLEMENT_DEQUE()* to generate the functions that correspond to this structure. The structure fields should not be manually changed at any time. Instead use one of the generated functions from the aforementioned macros.

Parameters

struct_name	The name of the structure
type	The name of the type of elements stored in the *struct_name* structure

See also

PROTOTYPE_DEQUE, IMPLEMENT_DEQUE

5.4.2.3 #define PROTOTYPE_DEQUE(*struct_name*, *type*)

Value:

```
struct_name new_##struct_name(size_t);                              \
  struct_name new_destructable_##struct_name(size_t, void (*)(type));  \
  void destroy_##struct_name(struct_name*);                         \
  void empty_##struct_name(struct_name*);                           \
  bool is_empty_##struct_name(struct_name*);                        \
  size_t length_##struct_name(struct_name*);                        \
  type* as_array_##struct_name(struct_name*, size_t*);              \
  void apply_##struct_name(struct_name*, void (*)(type));           \
  void push_front_##struct_name(struct_name*, type);                \
  void push_back_##struct_name(struct_name*, type);                 \
  type pop_front_##struct_name(struct_name*);                       \
  type pop_back_##struct_name(struct_name*);                        \
  type peek_front_##struct_name(struct_name*);                      \
  type peek_back_##struct_name(struct_name*);                       \
  void update_front_##struct_name(struct_name*, type);              \
  void update_back_##struct_name(struct_name*, type);
```

Generates prototypes for functions that manipulate Double Ended Queue structures.

This is intended for use in a header file or anywhere you need a forward declaration of these functions. This does not actually implement these functions.

Parameters

struct_name	The name of the structure
type	The name of the type of elements stored in the *struct_name* structure

See also

IMPLEMENT_DEQUE_STRUCT(), IMPLEMENT_DEQUE()

5.4.3 Function Documentation

5.4.3.1 void apply_Example (Example * *deq*, void(*)(Type) *func*)

Calls the function func on every element in the deque.

Note

We will have a lab over function pointers later in the semester. If this doesn't make sense now then skip it or come to your TA for assistance if you really want to use it.

Parameters

deq	A pointer to the deque to apply the function func to
func	A pointer to a function that takes an element of type

See also

Example, Type

5.4.3.2 Type * as_array_Example (Example * *deq*, size_t * *len*)

Extract an array based off the deque.

Note

Calling this function on a deque will invalidate the deque. This means no further deque functions can be called on the deq until it is reinitialized with *new_Example()*. If this function was created with the *IMPLEMENT_DEQUE()* macro, then the destructor is never called and you will be responsible for freeing the memory of the array.

Parameters

	deq	in A pointer to the deque to extract an array from
`out`	*len*	A pointer to an size_t value. This value will be set to the current length of the deque returned from the length_Example() function. If knowing the length of the array is unnecessary then NULL may be passed as this parameter.

Returns

An array containing the data from the deque

See also

Example, Type

5.4.3.3 void destroy_Example (Example * *deq*)

Destroy the deque structure freeing memory if necessary.

Parameters

deq	A pointer to the deque to destory

See also

Example

5.4.3.4 void empty_Example (Example * *deq*)

Quickly empties the deque structure.

Parameters

deq	A pointer to the deque to empty

See also

Example

5.4.3.5 bool is_empty_Example (Example * *deq*)

Checks if the deque is empty.

Parameters

deq	A pointer to the deque to check if empty

Returns

Returns true if empty and false if not empty

See also

Example

5.4.3.6 size_t length_Example (Example * *deq*)

Query the number of elements in the deque.

Parameters

deq	A pointer to the deque to calculate the length of

Returns

The number of elements in the deque

See also

Example

5.4.3.7 Example new_destructable_Example (size_t *init_cap*, void(*)(Type) *destructor*)

Create a new, fully initialized deque structure with a destructor that is applied to every element when the *destroy_Example()* function is called.

Specifying the destructor is useful to not have to manually iterate over the deque destroying any malloc'd memory in each element.

Parameters

init_cap	Initial capacity of the deque
destructor	A function that is run on each element in the deque when *destroy_Example()* is called

Returns

A copy of the fully initialized struct

See also

Example

5.4.3.8 Example new_Example (size_t *init_cap*)

Create a new, fully initialized deque structure.

Parameters

init_cap	Initial capacity of the deque

Returns

A copy of the fully initialized struct

See also

Example

5.4.3.9 Type peek_back_Example (Example * *deq*)

Remove an element from the back of the deque.

Parameters

deq	A pointer to the deque to view the first element from

Returns

A copy of the element removed from the back of the queue

See also

Example, Type

5.4.3.10 Type peek_front_Example (Example * *deq*)

Get a copy of the element at the front of the deque.

Parameters

deq	A pointer to the deque to view the first element from

Returns

A copy of the element at the front of the deque

See also

Example, Type

5.4.3.11 Type pop_back_Example (Example * *deq*)

Remove an element from the back of the deque.

Parameters

deq	A pointer to the deque to remove an element from

Returns

A copy of the element removed from the back of the deque

See also

Example, Type

5.4.3.12 Type pop_front_Example (Example * *deq*)

Remove an element from the front of the deque.

Parameters

deq	A pointer to the deque to remove an element from

Returns

A copy of the element removed from the front of the deque

See also

Example, Type

5.4.3.13 void push_back_Example (Example * *deq*, Type *element*)

Insert an element to the back of the deque.

Parameters

deq	A pointer to the deque to insert the element
element	The element to copy into the deque

See also

Example, Type

5.4.3.14 void push_front_Example (Example * *deq*, Type *element*)

Insert an element to the front of the deque.

Parameters

deq	A pointer to the deque to insert the element
element	The element to copy into the deque

See also

Example, Type

5.4.3.15 void update_back_Example (Example * *deq,* Type *element*)

Change the element at the front of the deque to be a copy of element.

Parameters

deq	A pointer to the deque to update
element	The element to copy into the current last element in the deque

See also

Example, Type

5.4.3.16 void update_front_Example (Example * *deq,* Type *element*)

Change the element at the front of the deque to be a copy of element.

Parameters

deq	A pointer to the deque to update
element	The element to update the first element to

See also

Example, Type

5.5 src/execute.c File Reference

Implements interface functions between Quash and the environment and functions that interpret an execute commands.

```
#include "execute.h"
#include <stdio.h>
#include "quash.h"
```

Macros

- #define IMPLEMENT_ME() printf("IMPLEMENT ME: %s(line %d): %s()\n", __FILE__, __LINE__, __FUN↩ CTION__)

  Note calls to any function that requires implementation.

Functions

- char * get_current_directory (bool *should_free)

  Get the real current working directory.
- const char * lookup_env (const char *env_var)

  Function to get environment variable values.
- void write_env (const char *env_var, const char *val)

  Function to set and define environment variable values.
- void check_jobs_status ()

  Check on background jobs to see if they have exited.
- void print_job (int job_id, pid_t pid, const char *cmd)

  Print a job to standard out.
- void print_job_complete (int job_id, pid_t pid, const char *cmd)

  Print the completion of a job to standard out.
- void run_generic (GenericCommand cmd)

  Run a generic (non-builtin) command.
- void run_echo (EchoCommand cmd)

  Run the builtin echo command.
- void run_export (ExportCommand cmd)

  Run the builtin export command.
- void run_cd (CDCommand cmd)

  Run the builtin cd (change directory) command.
- void run_kill (KillCommand cmd)

  Run the builtin kill command.
- void run_pwd ()

  Run the builtin pwd (print working directory) command.
- void run_jobs ()

  Run the builtin jobs command to show the jobs list.
- bool run_child_process_command (Command cmd)

  Run a Command that should not impact the quash process in any way.
- bool run_quash_process_command (Command cmd)

  Run a Command that should impact the quash process in some way.
- void create_process (CommandHolder holder)

  Create a process centered around the Command in the CommandHolder setting up redirects and pipes where needed.
- void run_script (CommandHolder *holders)

  Common entry point for all commands.

5.5.1 Detailed Description

Implements interface functions between Quash and the environment and functions that interpret an execute commands.

Note

As you add things to this file you may want to change the method signature

5.5.2 Function Documentation

5.5.2.1 void create_process (CommandHolder *holder*)

Create a process centered around the *Command* in the *CommandHolder* setting up redirects and pipes where needed.

Note

Not all commands should be run in the child process. A few need to change the quash process in some way

Parameters

holder	The CommandHolder to try to run

See also

Command CommandHolder

5.5.2.2 char* get_current_directory (bool * *should_free*)

Get the real current working directory.

This is not necessarily the same as the PWD environment variable and setting PWD does not actually change the current working directory.

Parameters

out	*should_free*	Set this to true if the returned string should be free'd by the caller and false otherwise.

Returns

A string representing the current working directory

5.5.2.3 const char* lookup_env (const char * *env_var*)

Function to get environment variable values.

Parameters

env_var	Environment variable to lookup

Returns

String containing the value of the environment variable env_var

5.5.2.4 void print_job (int *job_id,* pid_t *pid,* const char * *cmd*)

Print a job to standard out.

We use the minimum of what a Job structure should contain to pass to this function.

Parameters

job_id	Job identifier number.
pid	Process id of a process belonging to this job.
cmd	String holding an approximation of what the user typed in for the command.

5.5.2.5 void print_job_complete (int *job_id,* pid_t *pid,* const char * *cmd*)

Print the completion of a job to standard out.

We use the minimum of what a Job should contain to pass to this function.

Parameters

job_id	Job identifier number.
pid	Process id of a process belonging to this job.
cmd	String holding an aproximation of what the user typed in for the command.

5.5.2.6 void run_cd (CDCommand *cmd*)

Run the builtin cd (change directory) command.

Parameters

cmd	An *CDCommand*

See also

CDCommand

5.5.2.7 bool run_child_process_command (Command *cmd*)

Run a *Command* that should not impact the quash process in any way.

Parameters

cmd	The Command to try to run

See also

Command

5.5.2.8 void run_echo (EchoCommand *cmd*)

Run the builtin echo command.

Parameters

cmd	An *EchoCommand*

See also

EchoCommand

5.5.2.9 void run_export (ExportCommand *cmd*)

Run the builtin export command.

Parameters

cmd	An *ExportCommand*

See also

ExportCommand

5.5.2.10 void run_generic (GenericCommand *cmd*)

Run a generic (non-builtin) command.

Parameters

cmd	A *GenericCommand* command

See also

GenericCommand

5.5.2.11 void run_jobs ( )

Run the builtin jobs command to show the jobs list.

See also

PWDCommand

5.5.2.12 void run_kill (KillCommand *cmd*)

Run the builtin kill command.

Parameters

cmd	A *KillCommand*

See also

KillCommand

5.5.2.13 void run_pwd ( )

Run the builtin pwd (print working directory) command.

See also

PWDCommand

5.5.2.14 bool run_quash_process_command (Command *cmd*)

Run a *Command* that should impact the quash process in some way.

Parameters

cmd	The Command to try to run

Returns

True if command was run and false otherwise.

See also

Command

5.5.2.15 void run_script (CommandHolder * *holders*)

Common entry point for all commands.

This function resolves the type of the command and calls the relevant run function

Parameters

holders	An array of command holders

See also

Command

5.5.2.16 void write_env (const char * *env_var,* const char * *val*)

Function to set and define environment variable values.

Parameters

env_var	Environment variable to set
val	String with the value to set the environment variable env_var

5.6 src/execute.h File Reference

Functions for interpreting and running commands.

```
#include <stdbool.h>
#include <unistd.h>
#include "command.h"
```

Functions

- const char * lookup_env (const char *env_var)
  Function to get environment variable values.
- void write_env (const char *env_var, const char *val)
  Function to set and define environment variable values.
- char * get_current_directory (bool *should_free)
  Get the real current working directory.
- void check_jobs_status ()
  Check on background jobs to see if they have exited.
- void print_job (int job_id, pid_t pid, const char *cmd)
  Print a job to standard out.
- void print_job_complete (int job_id, pid_t pid, const char *cmd)
  Print the completion of a job to standard out.
- void run_generic (GenericCommand cmd)
  Run a generic (non-builtin) command.
- void run_echo (EchoCommand cmd)
  Run the builtin echo command.
- void run_export (ExportCommand cmd)
  Run the builtin export command.
- void run_cd (CDCommand cmd)
  Run the builtin cd (change directory) command.
- void run_kill (KillCommand cmd)
  Run the builtin kill command.
- void run_pwd ()
  Run the builtin pwd (print working directory) command.
- void run_jobs ()
  Run the builtin jobs command to show the jobs list.
- void run_script (CommandHolder *holders)
  Common entry point for all commands.

5.6.1 Detailed Description

Functions for interpreting and running commands.

5.6.2 Function Documentation

5.6.2.1 char* get_current_directory (bool * *should_free*)

Get the real current working directory.

This is not necessarily the same as the PWD environment variable and setting PWD does not actually change the current working directory.

Parameters

out	*should_free*	Set this to true if the returned string should be free'd by the caller and false otherwise.

Returns

A string representing the current working directory

5.6.2.2 const char* lookup_env (const char * *env_var*)

Function to get environment variable values.

Parameters

env_var	Environment variable to lookup

Returns

String containing the value of the environment variable env_var

5.6.2.3 void print_job (int *job_id,* pid_t *pid,* const char * *cmd*)

Print a job to standard out.

We use the minimum of what a Job structure should contain to pass to this function.

Parameters

job_id	Job identifier number.
pid	Process id of a process belonging to this job.
cmd	String holding an approximation of what the user typed in for the command.

5.6.2.4 void print_job_complete (int *job_id,* pid_t *pid,* const char * *cmd*)

Print the completion of a job to standard out.

We use the minimum of what a Job should contain to pass to this function.

Parameters

job_id	Job identifier number.
pid	Process id of a process belonging to this job.
cmd	String holding an aproximation of what the user typed in for the command.

5.6.2.5 void run_cd (CDCommand *cmd*)

Run the builtin cd (change directory) command.

Parameters

cmd	An *CDCommand*

See also

CDCommand

5.6.2.6 void run_echo (EchoCommand *cmd*)

Run the builtin echo command.

Parameters

cmd	An *EchoCommand*

See also

EchoCommand

5.6.2.7 void run_export (ExportCommand *cmd*)

Run the builtin export command.

Parameters

cmd	An *ExportCommand*

See also

ExportCommand

5.6.2.8 void run_generic (GenericCommand *cmd*)

Run a generic (non-builtin) command.

Parameters

cmd	A *GenericCommand* command

See also

GenericCommand

5.6.2.9 void run_jobs ( )

Run the builtin jobs command to show the jobs list.

See also

PWDCommand

5.6.2.10 void run_kill (KillCommand *cmd*)

Run the builtin kill command.

Parameters

cmd	A *KillCommand*

See also

KillCommand

5.6.2.11 void run_pwd ( )

Run the builtin pwd (print working directory) command.

See also

PWDCommand

5.6.2.12 void run_script (CommandHolder * *holders*)

Common entry point for all commands.

This function resolves the type of the command and calls the relevant run function

Parameters

holders	An array of command holders

See also

Command

5.6.2.13 void write_env (const char * *env_var*, const char * *val*)

Function to set and define environment variable values.

Parameters

env_var	Environment variable to set
val	String with the value to set the environment variable env_var

5.7 src/parsing/memory_pool.h File Reference

An abstraction of malloc that allows for all allocations in the pool to be free'd with a single call to destroy_memory_pool(). Allocations to the memory pool should NOT be manually free'd with a call to free().

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

Macros

- #define IMPLEMENT_DEQUE_MEMORY_POOL(struct_name, type)

  Generates a memory_pool_alloc() based set of functions for use with a structure generated by IMPLEMENT_DEQUE_STRUCT.

Functions

- void initialize_memory_pool (size_t size)

  Allocate the memory pool.
- void * memory_pool_alloc (size_t size)

  Reserve some space in the memory pool and returns a unique address that can be written to and read from. This can be thought of exactly like malloc() without the requirement of calling free() directly on the returned pointer.
- void destroy_memory_pool ()

  Free all memory allocated in the memory pool.
- char * memory_pool_strdup (const char *str)

  A version of strdup() that allocates the duplicate to the memory pool rather than with malloc directly.

5.7.1 Detailed Description

An abstraction of malloc that allows for all allocations in the pool to be free'd with a single call to destroy_memory_pool(). Allocations to the memory pool should NOT be manually free'd with a call to free().

Warning

FOR THE QUASH PROJECT, DO NOT USE OR COPY ANY OF THESE FUNCTIONS YOURSELF. This is a simple garbage collector to fix memory leak problems resulting from the parser, generated by bison, encountering syntax errors in the input that would be very difficult or impossible to fix without this mechanism. Generally, hand crafted c code should be able to manage it's heap allocations directly with malloc and free. We would like for you to become comfortable with malloc based memory management. YOU WILL BE PENALIZED FOR USING ANYTHING IN THIS FILE TO HIDE MEMORY LEAKS.
The memory pool allocations are not thread safe

5.7.2 Macro Definition Documentation

5.7.2.1 #define IMPLEMENT_DEQUE_MEMORY_POOL(*struct_name, type*)

Generates a *memory_pool_alloc()* based set of functions for use with a structure generated by *IMPLEMENT_DEQUE_STRUCT*.

Parameters

struct_name	The name of the structure
type	The name of the type of elements stored in the *struct_name* structure

See also

IMPLEMENT_DEQUE_STRUCT, PROTOTYPE_DEQUE, IMPLEMENT_DEQUE_MEMORY_POOL, memory_pool_alloc()

5.7.3 Function Documentation

5.7.3.1 void initialize_memory_pool (size_t *size*)

Allocate the memory pool.

Parameters

size	The initial size of the memory pool. If this value is zero then a default size of one is used.

5.7.3.2 void* memory_pool_alloc (size_t *size*)

Reserve some space in the memory pool and returns a unique address that can be written to and read from. This can be thought of exactly like malloc() without the requirement of calling free() directly on the returned pointer.

Parameters

size	Size in bytes of the requested reserved space

Returns

A pointer to a unique array of size bytes

5.7.3.3 char* memory_pool_strdup (const char * *str*)

A version of strdup() that allocates the duplicate to the memory pool rather than with malloc directly.

Parameters

str	Pointer to the string to duplicate

Returns

A copy of str allocated in the memory pool

5.8 src/parsing/parsing_interface.h File Reference

Defines an interface between c and the parser. This file is also responsible for defining many of the structures used by the parser.

```
#include <stdbool.h>
#include "command.h"
#include "deque.h"
#include "quash.h"
```

Classes

- struct Redirect

  Intermediate parsing structure used to determine the final configuration of the redirects in a command.

Typedefs

- typedef struct Redirect Redirect

  Intermediate parsing structure used to determine the final configuration of the redirects in a command.

Functions

- Redirect mk_redirect (char *in, char *out, bool append)

  Creates an intermediate Redirect structure and returns a copy.
- char * interpret_complex_string_token (const char *str)

  Clean up a string by removing escape symbols and unescaped single quotes. Also expands any environment variables.
- CommandHolder * parse (QuashState *state)

  Handles the call to the parser and provides a string equivalent of the Command structure to QuashState.
- void destroy_parser ()

  Cleanup memory dynamically allocated by the parser.

5.8.1 Detailed Description

Defines an interface between c and the parser. This file is also responsible for defining many of the structures used by the parser.

5.8.2 Function Documentation

5.8.2.1 char* interpret_complex_string_token (const char * *str*)

Clean up a string by removing escape symbols and unescaped single quotes. Also expands any environment variables.

Parameters

str	The string to clean up

Returns

The cleaned up and expanded string allocated on the *MemoryPool*

See also

MemoryPool

5.8.2.2 Redirect mk_redirect (char * *in*, char * *out*, bool *append*)

Creates an intermediate *Redirect* structure and returns a copy.

Doxygen_Suppress

Parameters

in	A string containing the name of a file to redirect the intput to a command from. This should be NULL if there are not any redirections in.
out	A string containing the name of a file to redirect the output of a command to. This should be NULL if there are not any redirections out.
append	Should the redirect out append or truncate a file. True if append, false if truncate.

Returns

A copy of the constructed Redirect structure

See also

Redirect

5.8.2.3 CommandHolder* parse (QuashState * *state*)

Handles the call to the parser and provides a string equivalent of the *Command* structure to *QuashState*.

Parameters

out	*state*	The state of the quash shell. The parsed_str member of QuashState is set to the stringified command structure.

Returns

A pointer to the parsed command structure

See also

CommandHolder, QuashState

5.9 src/quash.c File Reference

```
#include "quash.h"
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include "command.h"
#include "execute.h"
#include "parsing_interface.h"
#include "memory_pool.h"
```

Functions

- bool is_running ()
  Query if quash should accept more input or not.
- char * get_command_string ()
  Get a deep copy of the current command string.
- bool is_tty ()
  Check if Quash is receiving input from the command line (TTY)
- void end_main_loop ()
  Causes the execution loop to end.
- int main (int argc, char **argv)
  Quash entry point.

5.9.1 Detailed Description

Quash's main file

5.9.2 Function Documentation

5.9.2.1 char* get_command_string ( )

Get a deep copy of the current command string.

Note

The free function must be called on the result eventually

Returns

A copy of the command string

5.9.2.2 bool is_running ( )

Query if quash should accept more input or not.

Returns

True if Quash should accept more input and false otherwise

5.9.2.3 bool is_tty ( )

Check if Quash is receiving input from the command line (TTY)

Returns

True if stdin is a TTY false if stdin is not a TTY

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

Quash entry point.

Parameters

argc	argument count from the command line
argv	argument vector from the command line

Returns

program exit status

5.10 src/quash.h File Reference

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "execute.h"
```

Classes

- struct QuashState

  Holds information about the state and environment Quash is running in.

Typedefs

- typedef struct QuashState QuashState

  Holds information about the state and environment Quash is running in.

Functions

- bool is_tty ()

  Check if Quash is receiving input from the command line (TTY)
- char * get_command_string ()

  Get a deep copy of the current command string.
- bool is_running ()

  Query if quash should accept more input or not.
- void end_main_loop ()

  Causes the execution loop to end.

5.10.1 Detailed Description

Quash essential functions and structures.

5.10.2 Function Documentation

5.10.2.1 char* get_command_string ( )

Get a deep copy of the current command string.

Note

The free function must be called on the result eventually

Returns

A copy of the command string

5.10.2.2 bool is_running ( )

Query if quash should accept more input or not.

Returns

True if Quash should accept more input and false otherwise

5.10.2.3 bool is_tty ( )

Check if Quash is receiving input from the command line (TTY)

Returns

True if stdin is a TTY false if stdin is not a TTY