My Shell

Doxygen 1.9.4

Chapter 1

Chapter 2

Chapter 3

Artshell

MyShell, version 1.0.0-release (x86_64-pc-linux-gnu)

MyShell

1) bg <job> —— <job> <job>

2) cd <directory> —— <directory> <directory>

3) clr ——

4) dir <directory> —— <directory> <directory>

5) echo <comment> —— <comment>

6) exec <command> —— <command>

7) exit —— shell

8) fg <job> —— <job> <job>

9) help ——

10) jobs <job> ——

11) pwd ——

12) set ——

13) test <expression> —— <expression> true false

14) time ——

15) umask <mask> —— <mask>

E:/Artshell/doc/ .md ” ”

Chapter 4

4.1

, :

Chapter 5

5.1

:

Chapter 6

6.1

:

Chapter 7

7.1

:

Chapter 8

8.1 SHELL

- int shell_setup (int argc, char *argv[], char *env[])
  shell
- int shell_loop (Console *model, Display *view, Executor *controller, char *env[])
  shell

8.1.1

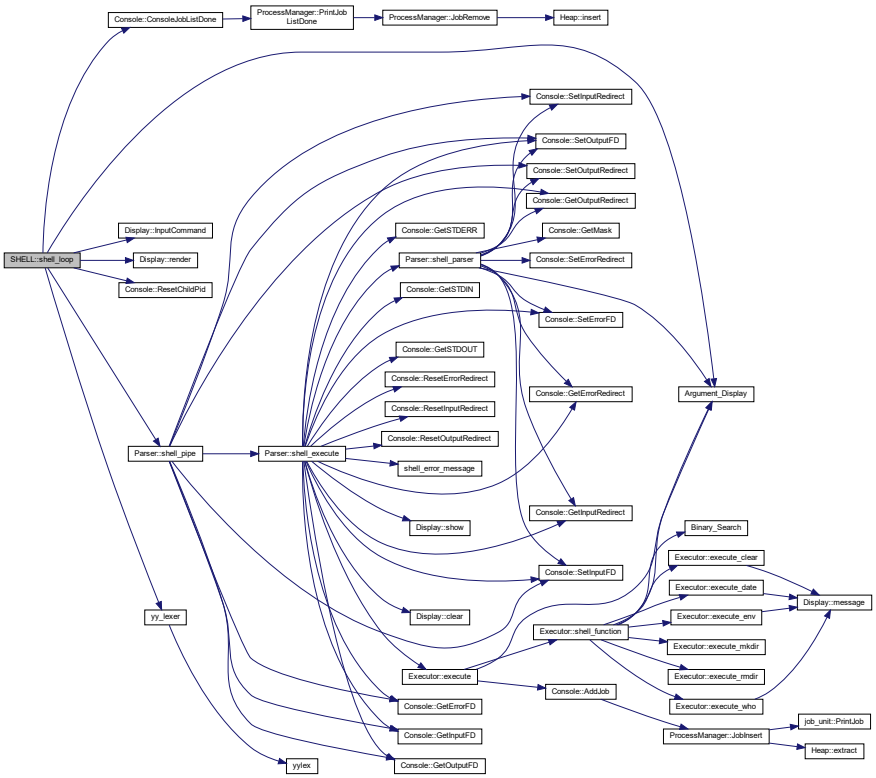
8.1.1.1 shell_loop()

```
int SHELL::shell_loop (
                Console * model,
                Display * view,
                Executor * controller,
                char * env[] )
```

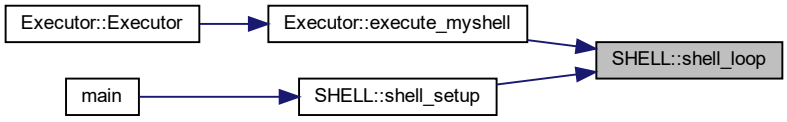
shell

myshell.cpp 71 .

:



:



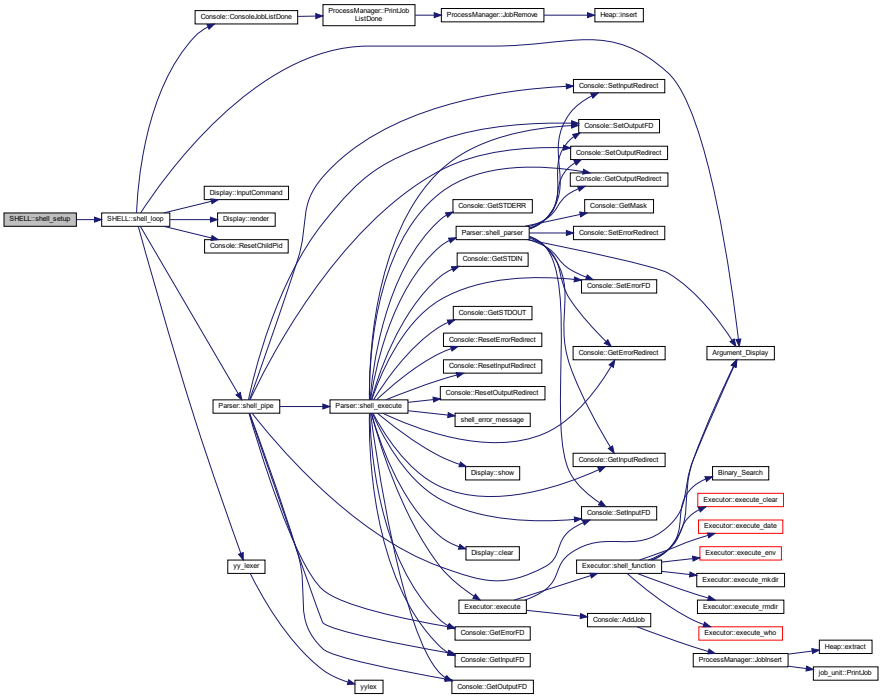
8.1.1.2 shell_setup()

```
int SHELL::shell_setup (
                int argc,
                char * argv[],
                char * env[] )
```

shell

myshell.cpp 35 .

:


Console::ConsoleJobListDone
ProcessManager::PrintJob ListDone
ProcessManager::JobRemove
Heap::insert
Console::SetInputRedirect
Console::SetOutputFD
Console::SetOutputRedirect
Console::GetOutputRedirect
Display::InputCommand
SHELL::shell_setup
SHELL::shell_loop
Display::render
Console::ResetChildPid
Console::GetSTDERR
Console::GetMask
Parser::shell_parser
Console::SetErrorRedirect
Console::GetSTDIN
Console::SetErrorFD
Console::GetSTDOUT
Console::ResetErrorRedirect
Console::GetErrorRedirect
Argument_Display
Console::ResetInputRedirect
Console::ResetOutputRedirect
Parser::shell_pipe
Parser::shell_execute
shell_error_message
Console::GetInputRedirect
Display::show
Binary_Search
Console::SetInputFD
Executor::execute_clear
Executor::execute_date
Display::clear
Executor::execute_env
yy_lexer
Executor::shell_function
Executor::execute_mkdir
Executor::execute_rmdir
Executor::execute
Console::AddJob
Executor::execute_who
Console::GetErrorFD
ProcessManager::JobInsert
Heap::extract
Console::GetInputFD
job_unit::PrintJob
yylex
Console::GetOutputFD

:

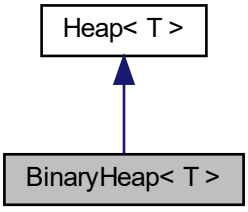

main
SHELL::shell_setup

Chapter 9

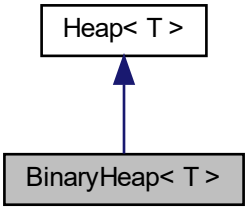
9.1 BinaryHeap< T >

#include <BinaryHeap.h>

BinaryHeap< T > :



BinaryHeap< T > :



- class ExtractEmptyHeap
- class OutOfMemory

Public

- BinaryHeap (size_t heap_capacity=HeapBlockSize)
- BinaryHeap (T data[], size_t size, size_t heap_capacity=HeapBlockSize)
- virtual ~BinaryHeap ()
- virtual void build (T data[], size_t size)
- virtual void insert (T value)
- virtual T top () const
- virtual T extract ()

Protected

- void AllocMoreSpace ()

Protected

- size_t capacity_
- T * node

Private

- void build_heap ()

9.1.1

template<class T>
class BinaryHeap< T >

T	

0.1

(3200105842@zju.edu.cn)

2022-07-20

BinaryHeap.h 34 .

9.1.2

9.1.2.1 BinaryHeap() [1/2]

```
template<class T >
BinaryHeap< T >::BinaryHeap (
                size_t heap_capacity = HeapBlockSize )   [inline]
```

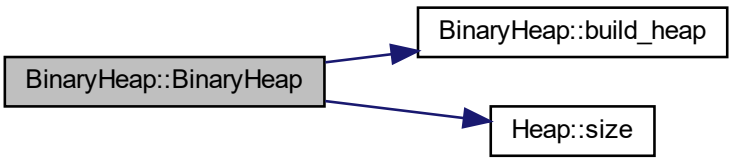
BinaryHeap.h 41 .

9.1.2.2 BinaryHeap() [2/2]

```
template<class T >
BinaryHeap< T >::BinaryHeap (
                T data[],
                size_t size,
                size_t heap_capacity = HeapBlockSize )   [inline]
```

BinaryHeap.h 51 .

:



9.1.2.3 ~BinaryHeap()

template<class T >
virtual BinaryHeap< T >::~BinaryHeap ( ) [inline], [virtual]

BinaryHeap.h 65 .

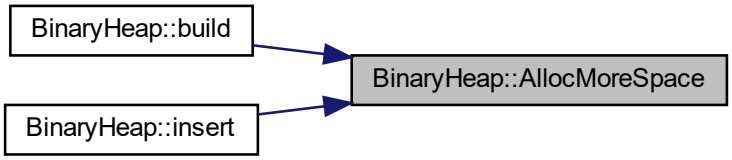
9.1.3

9.1.3.1 AllocMoreSpace()

template<class T >
void BinaryHeap< T >::AllocMoreSpace ( ) [inline], [protected]

BinaryHeap.h 150 .

:



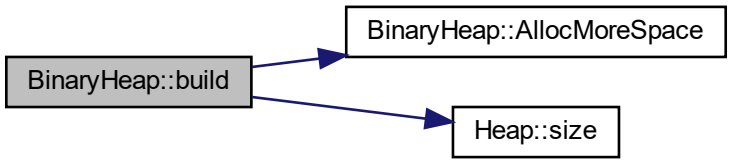
9.1.3.2 build()

template<class T >
virtual void BinaryHeap< T >::build (
T data[],
size_t size) [inline], [virtual]

Heap< T >.

BinaryHeap.h 70 .

:



9.1.3.3 build_heap()

template<class T >
void BinaryHeap< T >::build_heap ( ) [inline], [private]

BinaryHeap.h 166 .

:

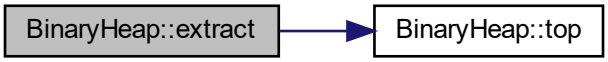
9.1.3.4 extract()

template<class T >
virtual T BinaryHeap< T >::extract ( ) [inline], [virtual]

Heap< T > .

BinaryHeap.h 117 .

:



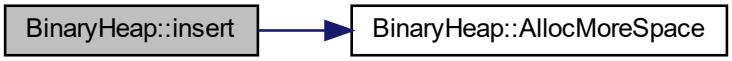
9.1.3.5 insert()

template<class T >
virtual void BinaryHeap< T >::insert (
T value) [inline], [virtual]

Heap< T > .

BinaryHeap.h 97 .

:



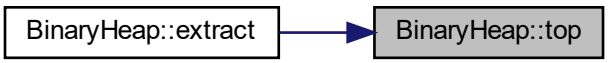
9.1.3.6 top()

template<class T >
virtual T BinaryHeap< T >::top ( ) const [inline], [virtual]

Heap< T > .

BinaryHeap.h 110 .

:



9.1.4

9.1.4.1 capacity_

template<class T >
size_t BinaryHeap< T >::capacity_ [protected]

BinaryHeap.h 144 .

9.1.4.2 node

template<class T >
T* BinaryHeap< T >::node [protected]

BinaryHeap.h 145 .

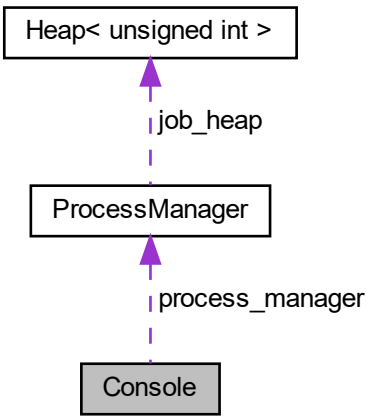
:

- E:/Artshell/inc/BinaryHeap.h

9.2 Console

#include <Console.h>

Console :



Public

- Console ()
- virtual ~Console ()
- int init ()
- void ConsoleJobList () const
- void ConsoleJobListDone ()
- unsigned int AddJob (int pid, job_state state, int argc, char *argv[])
- void ResetChildPid ()
- void SetInputFD (int _fd)
- void SetOutputFD (int _fd)
- void SetErrorFD (int _fd)
- int GetInputFD () const
- int GetOutputFD () const
- int GetErrorFD () const
- void SetInputRedirect ()
- void SetOutputRedirect ()
- void SetErrorRedirect ()
- void ResetInputRedirect ()
- void ResetOutputRedirect ()
- void ResetErrorRedirect ()
- bool GetInputRedirect () const
- bool GetOutputRedirect () const
- bool GetErrorRedirect () const
- int GetSTDIN () const
- int GetSTDOUT () const
- int GetSTDERR () const
- void SetMask (mode_t _mask)
- mode_t GetMask () const

Private

- char user_name [BUFFER_SIZE]
- char host_name [BUFFER_SIZE]
- char current_working_dictionary [BUFFER_SIZE]
- char home [BUFFER_SIZE]
- char shell_path_env [BUFFER_SIZE]
- pid_t process_id
- ProcessManager * process_manager
- int input_file_descriptor
- int output_file_descriptor
- int error_file_descriptor
- bool redirect_input
- bool redirect_output
- bool redirect_error
- mode_t umask_
- int argc
- char argv [MAX_ARGUMENT_NUMBER][BUFFER_SIZE]

Private

- static pid_t child_process_id = -1
- static int input_std_fd
- static int output_std_fd
- static int error_std_fd

- class Display
- class Executor
- class ProcessManager
- void SignalHandler (int)

9.2.1

Console.h 38 .

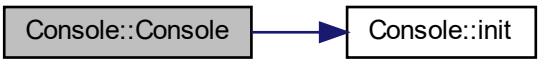
9.2.2

9.2.2.1 Console()

Console::Console ( )

Console.cpp 32 .

:



9.2.2.2 ~Console()

Console::~Console ( ) [virtual]

Console.cpp 44 .

9.2.3

9.2.3.1 AddJob()

unsigned int Console::AddJob (
int pid,
job_state state,
int argc,
char * argv[])

Console.cpp 213 .

:

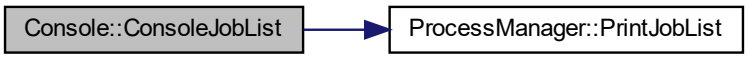
:

9.2.3.2 ConsoleJobList()

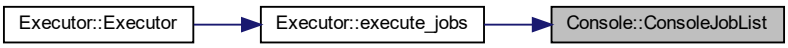
void Console::ConsoleJobList ( ) const

Console.cpp 201 .

:



:



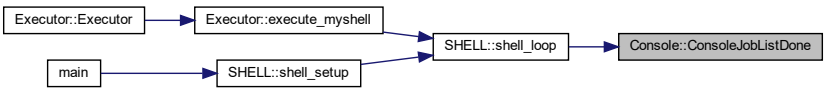
9.2.3.3 ConsoleJobListDone()

void Console::ConsoleJobListDone ( )

Console.cpp 207 .

:

:

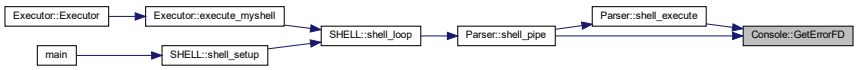


9.2.3.4 GetErrorFD()

int Console::GetErrorFD ( ) const [inline]

Console.h 110 .

:

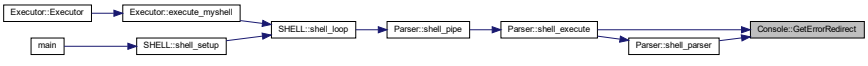


9.2.3.5 GetErrorRedirect()

bool Console::GetErrorRedirect ( ) const [inline]

Console.h 131 .

:

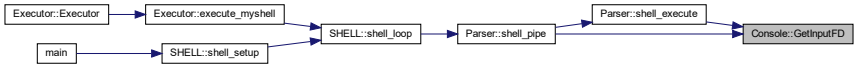


9.2.3.6 GetInputFD()

int Console::GetInputFD ( ) const [inline]

Console.h 106 .

:

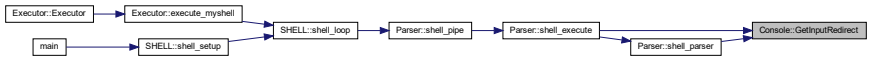


9.2.3.7 GetInputRedirect()

bool Console::GetInputRedirect ( ) const [inline]

Console.h 127 .

:



9.2.3.8 GetMask()

mode_t Console::GetMask ( ) const [inline]

Console.h 143 .

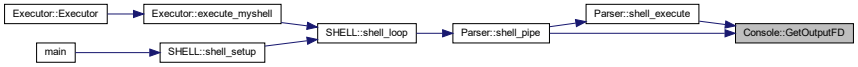
:

9.2.3.9 GetOutputFD()

int Console::GetOutputFD ( ) const [inline]

Console.h 108 .

:

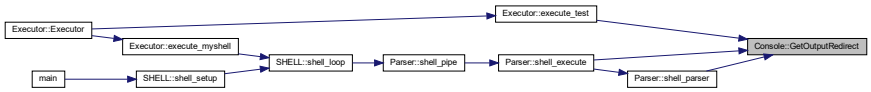


9.2.3.10 GetOutputRedirect()

bool Console::GetOutputRedirect ( ) const [inline]

Console.h 129 .

:

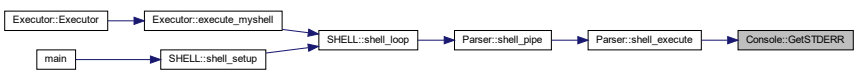


9.2.3.11 GetSTDERR()

int Console::GetSTDERR ( ) const [inline]

Console.h 138 .

:

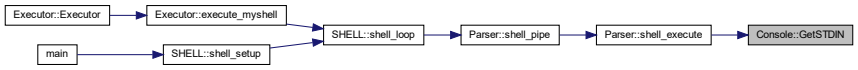


9.2.3.12 GetSTDIN()

int Console::GetSTDIN ( ) const [inline]

Console.h 134 .

:

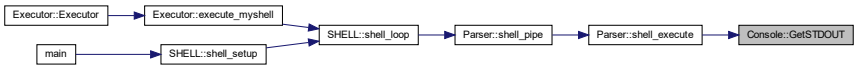


9.2.3.13 GetSTDOUT()

int Console::GetSTDOUT ( ) const [inline]

Console.h 136 .

:

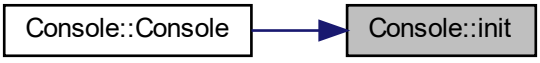


9.2.3.14 init()

int Console::init ( )

Console.cpp 123 .

:

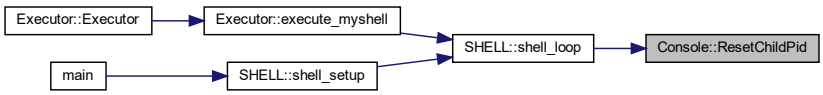


9.2.3.15 ResetChildPid()

void Console::ResetChildPid ( ) [inline]

Console.h 96 .

:

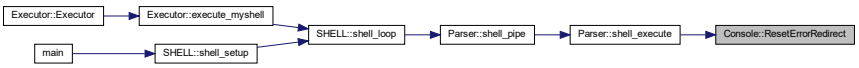


9.2.3.16 ResetErrorRedirect()

void Console::ResetErrorRedirect ( ) [inline]

Console.h 124 .

:

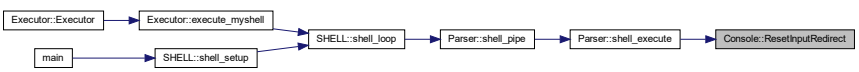


9.2.3.17 ResetInputRedirect()

void Console::ResetInputRedirect ( ) [inline]

Console.h 120 .

:

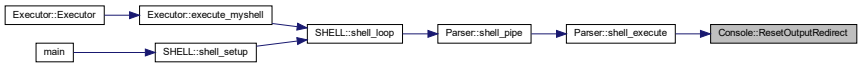


9.2.3.18 ResetOutputRedirect()

void Console::ResetOutputRedirect ( ) [inline]

Console.h 122 .

:



9.2.3.19 SetErrorFD()

void Console::SetErrorFD (
int _fd) [inline]

Console.h 103 .

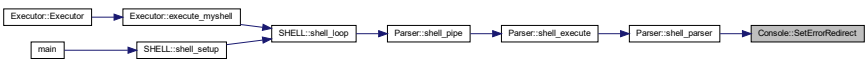
:

9.2.3.20 SetErrorRedirect()

void Console::SetErrorRedirect ( ) [inline]

Console.h 117 .

:

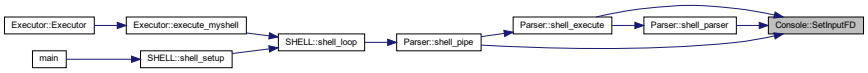


9.2.3.21 SetInputFD()

void Console::SetInputFD (
int _fd) [inline]

Console.h 99 .

:

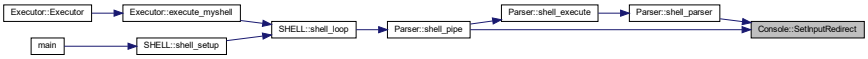


9.2.3.22 SetInputRedirect()

void Console::SetInputRedirect ( ) [inline]

Console.h 113 .

:



9.2.3.23 SetMask()

void Console::SetMask (
mode_t _mask) [inline]

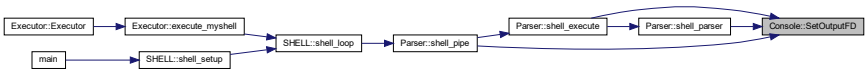
Console.h 141 .

9.2.3.24 SetOutputFD()

void Console::SetOutputFD (
int _fd) [inline]

Console.h 101 .

:

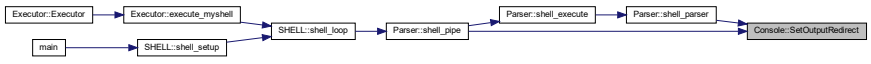


9.2.3.25 SetOutputRedirect()

void Console::SetOutputRedirect ( ) [inline]

Console.h 115 .

:



9.2.4

9.2.4.1 Display

friend class Display [friend]

Console.h 145 .

9.2.4.2 Executor

friend class Executor [friend]

Console.h 146 .

9.2.4.3 ProcessManager

friend class ProcessManager [friend]

Console.h 147 .

9.2.4.4 SignalHandler

void SignalHandler (
int signal_) [friend]

signal↩ _	

0.1

(3200105842@zju.edu.cn)

2022-07-21

Console.cpp 49 .

9.2.5

9.2.5.1 argc

int Console::argc [private]

Console.h 74 .

9.2.5.2 argv

char Console::argv[MAX_ARGUMENT_NUMBER][BUFFER_SIZE] [private]

Console.h 75 .

9.2.5.3 child_process_id

pid_t Console::child_process_id = -1 [static], [private]

Console.h 53 .

9.2.5.4 current_working_dictionary

char Console::current_working_dictionary[BUFFER_SIZE] [private]

Console.h 44 .

9.2.5.5 error_file_descriptor

int Console::error_file_descriptor [private]

Console.h 59 .

9.2.5.6 error_std_fd

int Console::error_std_fd [static], [private]

Console.h 64 .

9.2.5.7 home

char Console::home[BUFFER_SIZE] [private]

Console.h 46 .

9.2.5.8 host_name

char Console::host_name[BUFFER_SIZE] [private]

Console.h 43 .

9.2.5.9 input_file_descriptor

int Console::input_file_descriptor [private]

Console.h 57 .

9.2.5.10 input_std_fd

int Console::input_std_fd [static], [private]

Console.h 62 .

9.2.5.11 output_file_descriptor

int Console::output_file_descriptor [private]

Console.h 58 .

9.2.5.12 output_std_fd

int Console::output_std_fd [static], [private]

Console.h 63 .

9.2.5.13 process_id

pid_t Console::process_id [private]

Console.h 52 .

9.2.5.14 process_manager

ProcessManager* Console::process_manager [private]

Console.h 54 .

9.2.5.15 redirect_error

bool Console::redirect_error [private]

Console.h 69 .

9.2.5.16 redirect_input

bool Console::redirect_input [private]

Console.h 67 .

9.2.5.17 redirect_output

bool Console::redirect_output [private]

Console.h 68 .

9.2.5.18 shell_path_env

char Console::shell_path_env[BUFFER_SIZE] [private]

Console.h 49 .

9.2.5.19 umask_

mode_t Console::umask_ [private]

Console.h 72 .

9.2.5.20 user_name

char Console::user_name[BUFFER_SIZE] [private]

Console.h 42 .

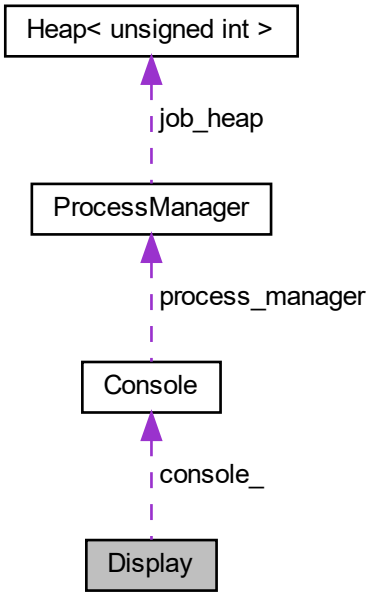
:

- E:/Artshell/inc/Console.h
- E:/Artshell/src/Console.cpp

9.3 Display

#include <Display.h>

Display :



Public

- Display (Console *console)
- virtual ~Display ()
- int InputCommand (char *input, const int len)
- void render ()
- void prompt () const
- void message (const char *msg)

  msg
- void show () const
- void clear ()

Protected

- std::string buffer_

Private

- Console * console_
- bool perform

9.3.1

Display.h 19 .

9.3.2

9.3.2.1 Display()

Display::Display (
Console * console)

Display.cpp 20 .

9.3.2.2 ~Display()

Display::~Display ( ) [virtual]

Display.cpp 25 .

9.3.3

9.3.3.1 clear()

void Display::clear ( ) [inline]

Display.h 55 .

:

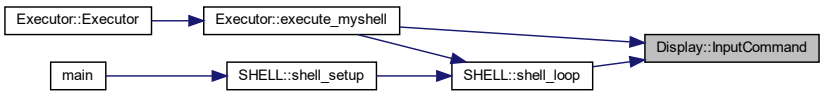
9.3.3.2 InputCommand()

int Display::InputCommand (
char * input,
const int len)

0 EOF

Display.cpp 29 .

:



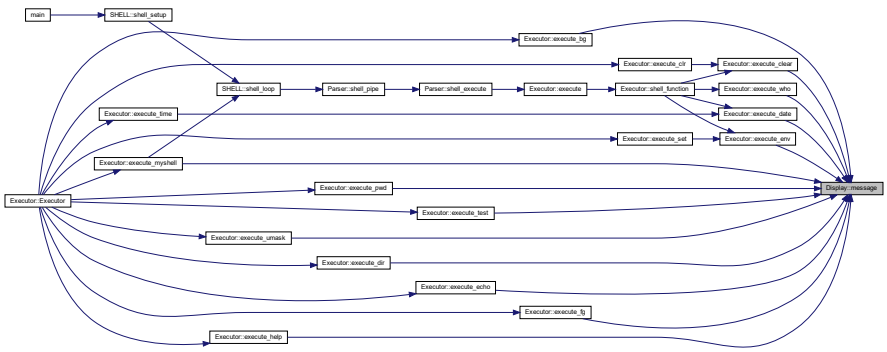
9.3.3.3 message()

void Display::message (
const char * msg)

msg

Display.cpp 147 .

:

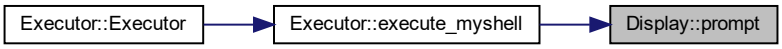


9.3.3.4 prompt()

void Display::prompt ( ) const

Display.cpp 139 .

:

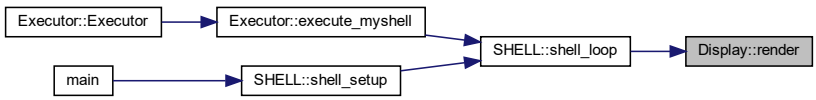


9.3.3.5 render()

void Display::render ( )

Display.cpp 88 .

:



9.3.3.6 show()

void Display::show ( ) const

Display.cpp 152 .

:

9.3.4

9.3.4.1 buffer_

std::string Display::buffer_ [protected]

Display.h 28 .

9.3.4.2 console_

Console* Display::console_ [private]

Display.h 23 .

9.3.4.3 perform

bool Display::perform [private]

Display.h 25 .

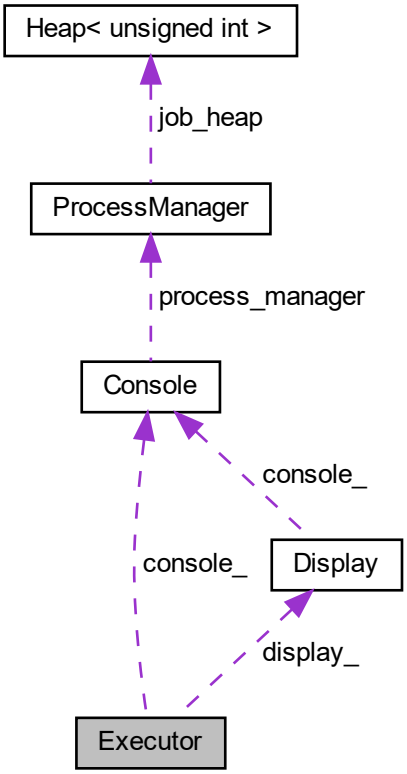
:

- E:/Artshell/inc/Display.h
- E:/Artshell/src/Display.cpp

9.4 Executor

#include <Executor.h>

Executor :



Public

- Executor (Console *model, Display *view)
- virtual ~Executor ()
- sh_err_t execute (const int argc, char *const argv[], char *const env[]) const

Protected

- typedef sh_err_t(Executor::* MemFuncPtr) (const int argc, char *const argv[], char *const env[]) const

STL

Protected

- sh_err_t shell_function (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_cd (const int argc, char *const argv[], char *const env[]) const

- sh_err_t execute_pwd (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_time (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_clr (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_dir (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_set (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_echo (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_help (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_exit (const int argc, char *const argv[], char *const env[]) const
  shell
- sh_err_t execute_date (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_clear (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_env (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_who (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_mkdir (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_rmdir (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_bg (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_fg (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_jobs (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_exec (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_test (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_umask (const int argc, char *const argv[], char *const env[]) const
- sh_err_t execute_myshell (const int argc, char *const argv[], char *const env[]) const
  myshell

Protected

- static bool test_file_state (const int argc, const char *const argv[])
- static bool test_number_compare (const int argc, const char *const argv[])
- static bool test_string_compare (const int argc, const char *const argv[])

Protected

- MemFuncPtr FunctionArray [FunctionNumber]

Private

- Console * console_
- Display * display_

9.4.1

Executor.h 22 .

9.4.2

9.4.2.1 MemFuncPtr

typedef sh_err_t(Executor::* Executor::MemFuncPtr) (const int argc, char *const argv[], char *const env[]) const [protected]

STL

Executor.h 104 .

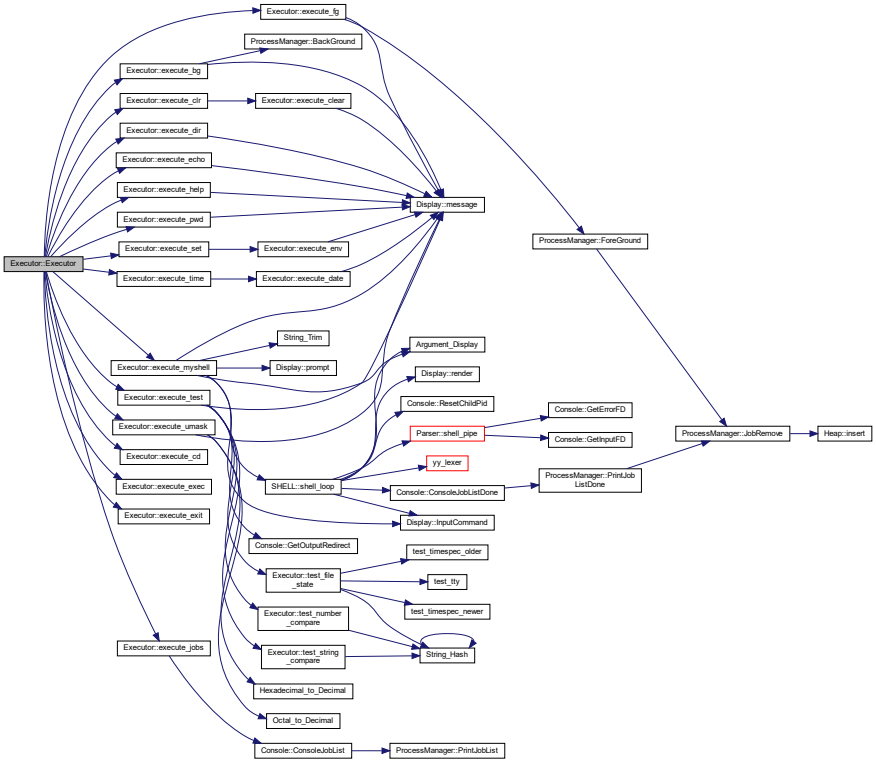
9.4.3

9.4.3.1 Executor()

Executor::Executor (
Console * model,
Display * view)

Executor.cpp 42 .

:



9.4.3.2 ~Executor()

Executor::~Executor ( ) [virtual]

Executor.cpp 72 .

9.4.4

9.4.4.1 execute()

sh_err_t Executor::execute (
const int argc,
char *const argv[],
char *const env[]) const

argc	
argv	
env	

sh_err_t

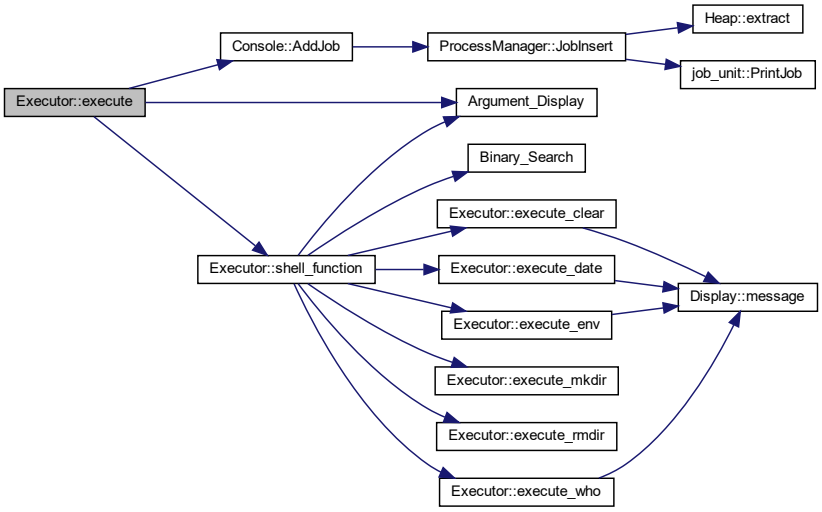
0.1

(3200105842@zju.edu.cn)

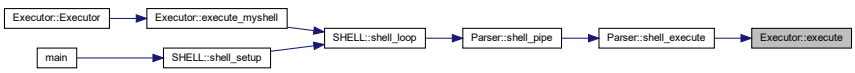
2022-07-04

Executor.cpp 76 .

:



:

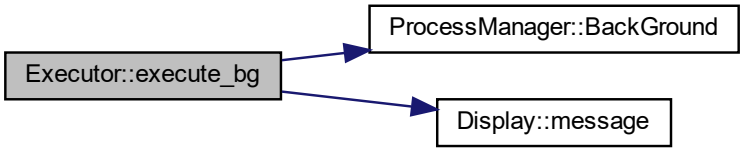


9.4.4.2 execute_bg()

```
sh_err_t Executor::execute_bg (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 585 .

:



:

9.4.4.3 execute_cd()

```
sh_err_t Executor::execute_cd (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 251 .

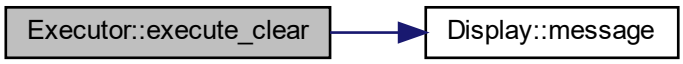
:

9.4.4.4 execute_clear()

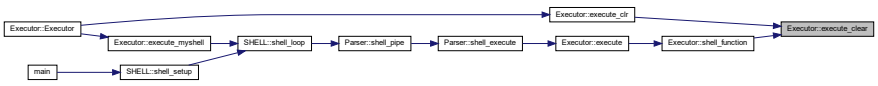
sh_err_t Executor::execute_clear (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 524 .

:



:

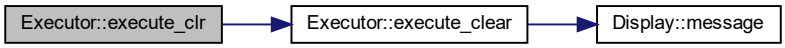


9.4.4.5 execute_clr()

```
sh_err_t Executor::execute_clr (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 316 .

:



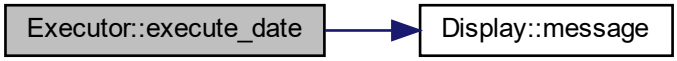
:

9.4.4.6 execute_date()

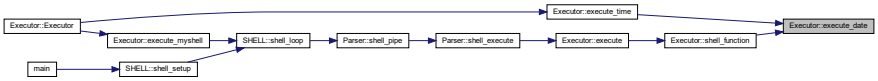
```
sh_err_t Executor::execute_date (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 499 .

:



:

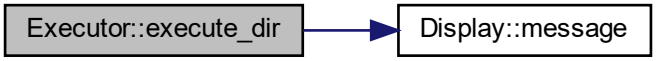


9.4.4.7 execute_dir()

sh_err_t Executor::execute_dir (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 324 .

:



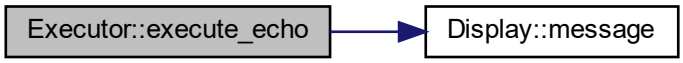
:

9.4.4.8 execute_echo()

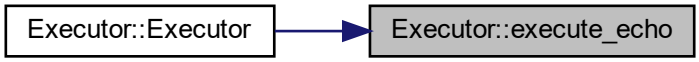
```
sh_err_t Executor::execute_echo (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 445 .

:



:

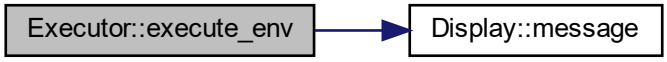


9.4.4.9 execute_env()

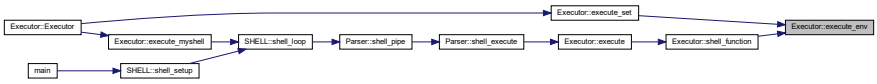
```
sh_err_t Executor::execute_env (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 531 .

:



:

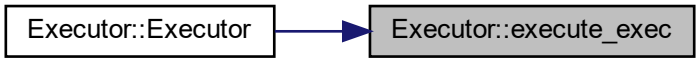


9.4.4.10 execute_exec()

sh_err_t Executor::execute_exec (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 639 .

:



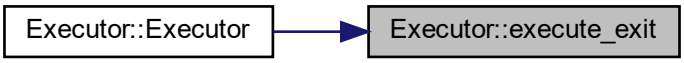
9.4.4.11 execute_exit()

sh_err_t Executor::execute_exit (
const int argc,
char *const argv[],
char *const env[]) const [protected]

shell

Executor.cpp 493 .

:

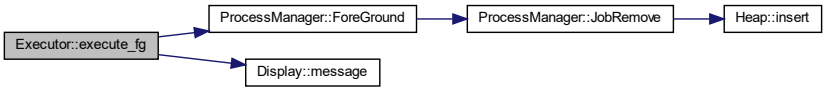


9.4.4.12 execute_fg()

```
sh_err_t Executor::execute_fg (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 611 .

:



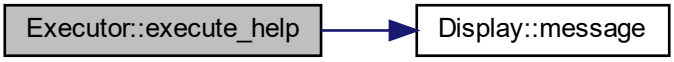
:

9.4.4.13 execute_help()

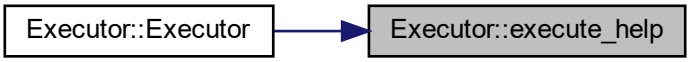
```
sh_err_t Executor::execute_help (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 462 .

:



:

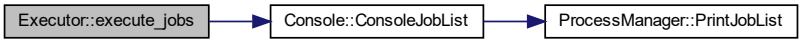


9.4.4.14 execute_jobs()

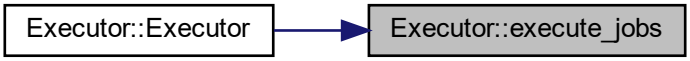
sh_err_t Executor::execute_jobs (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 630 .

:



:

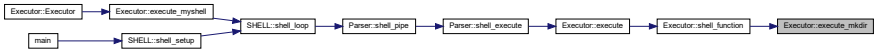


9.4.4.15 execute_mkdir()

sh_err_t Executor::execute_mkdir (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 555 .

:



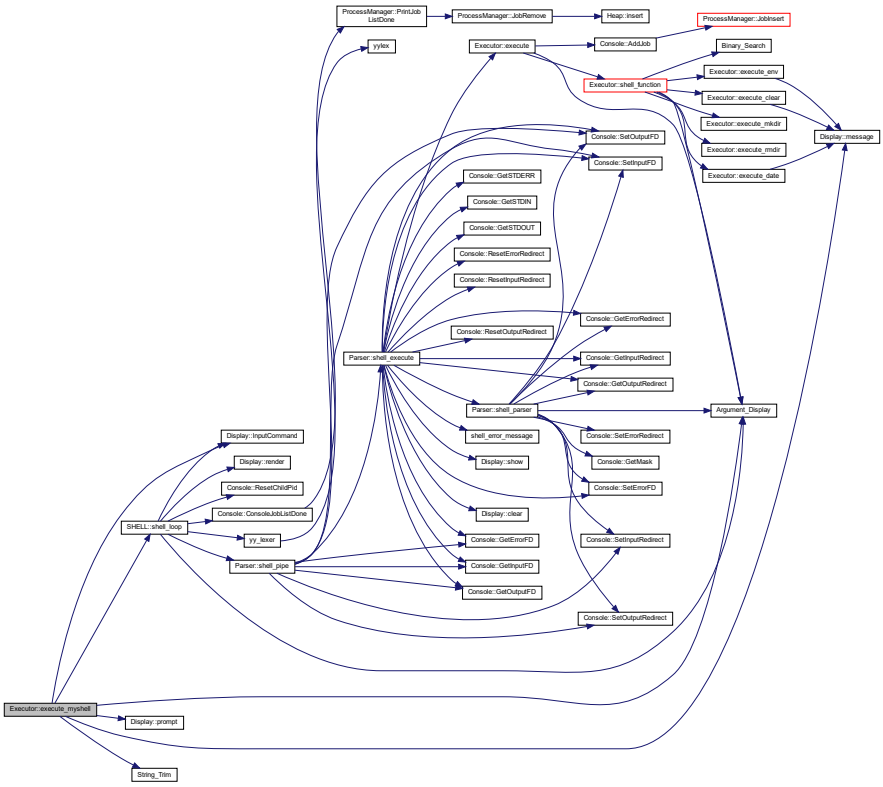
9.4.4.16 execute_myshell()

sh_err_t Executor::execute_myshell (
const int argc,
char *const argv[],
char *const env[]) const [protected]

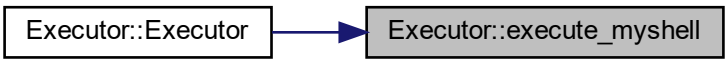
myshell

Executor.cpp 739 .

:



:

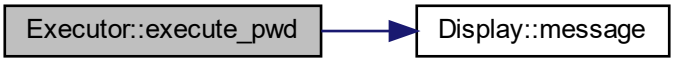


9.4.4.17 execute_pwd()

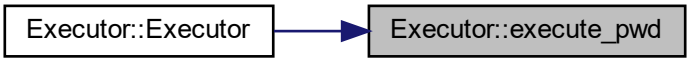
sh_err_t Executor::execute_pwd (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 300 .

:



:



9.4.4.18 execute_rmdir()

```
sh_err_t Executor::execute_rmdir (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 571 .

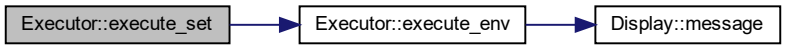
:

9.4.4.19 execute_set()

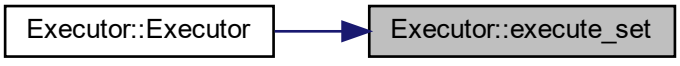
```
sh_err_t Executor::execute_set (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 437 .

:



:

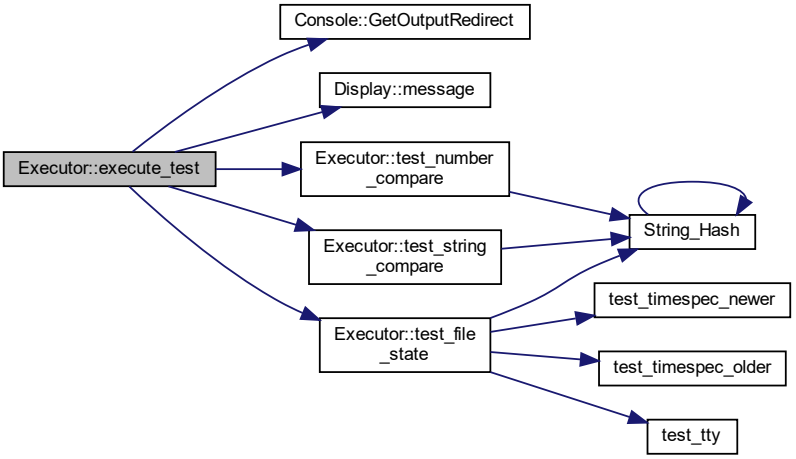


9.4.4.20 execute_test()

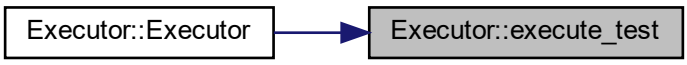
```
sh_err_t Executor::execute_test (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 657 .

:



:

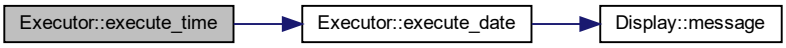


9.4.4.21 execute_time()

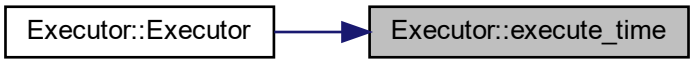
```
sh_err_t Executor::execute_time (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 308 .

:



:

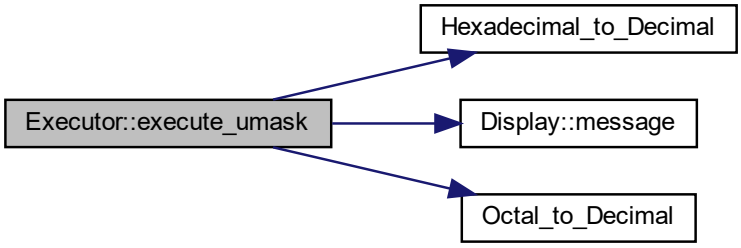


9.4.4.22 execute_umask()

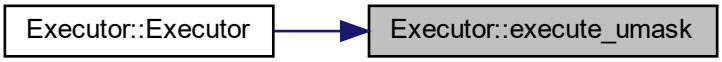
```
sh_err_t Executor::execute_umask (
                const int argc,
                char *const argv[],
                char *const env[] ) const    [protected]
```

Executor.cpp 702 .

:



:

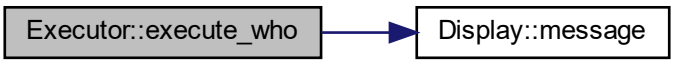


9.4.4.23 execute_who()

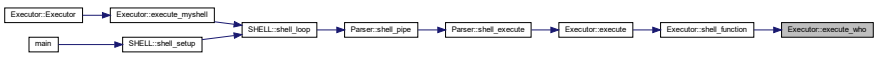
sh_err_t Executor::execute_who (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 547 .

:



:

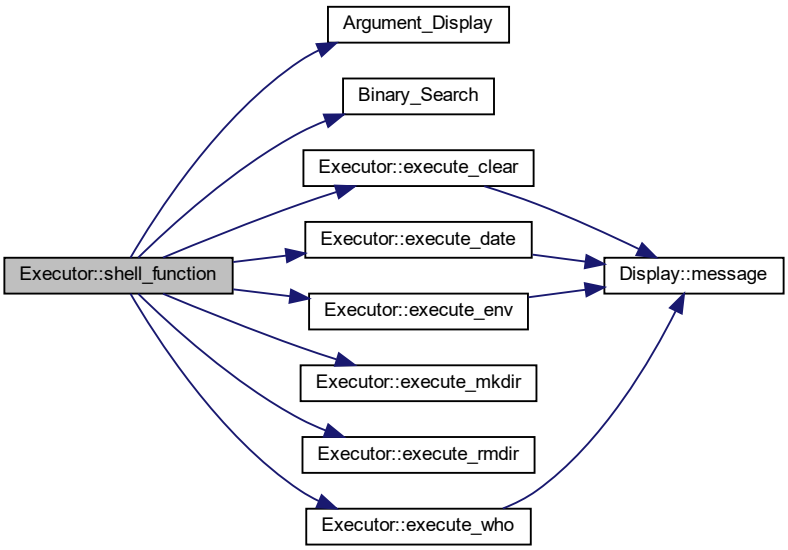


9.4.4.24 shell_function()

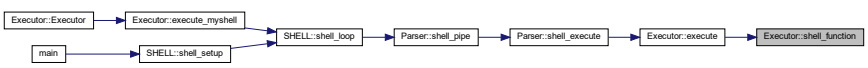
sh_err_t Executor::shell_function (
const int argc,
char *const argv[],
char *const env[]) const [protected]

Executor.cpp 168 .

:



:

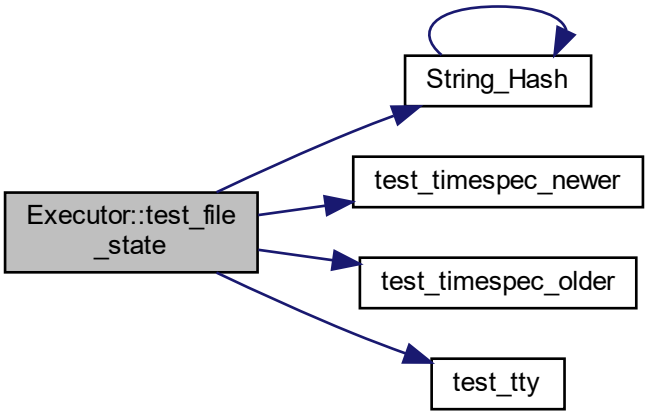


9.4.4.25 test_file_state()

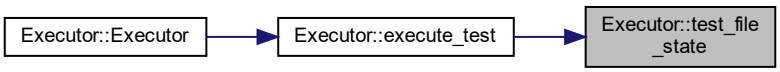
bool Executor::test_file_state (
const int argc,
const char *const argv[]) [static], [protected]

Executor.cpp 856 .

:



:

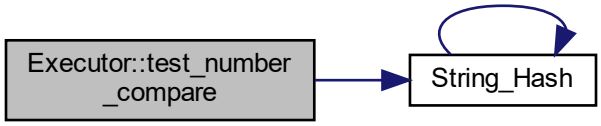


9.4.4.26 test_number_compare()

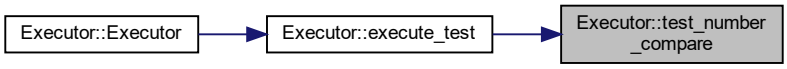
bool Executor::test_number_compare (
const int argc,
const char *const argv[]) [static], [protected]

Executor.cpp 966 .

:



:

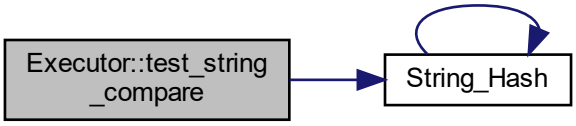


9.4.4.27 test_string_compare()

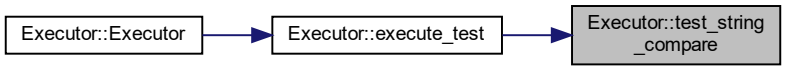
bool Executor::test_string_compare (
const int argc,
const char *const argv[]) [static], [protected]

Executor.cpp 1008 .

:



:



9.4.5

9.4.5.1 console_

Console* Executor::console_ [private]

Executor.h 26 .

9.4.5.2 display_

Display* Executor::display_ [private]

Executor.h 28 .

9.4.5.3 FunctionArray

MemFuncPtr Executor::FunctionArray[FunctionNumber] [protected]

Executor.h 106 .

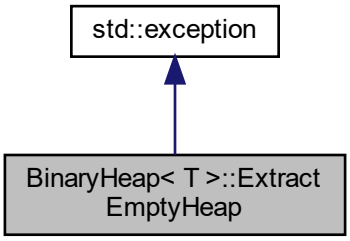
:

- E:/Artshell/inc/Executor.h
- E:/Artshell/src/Executor.cpp

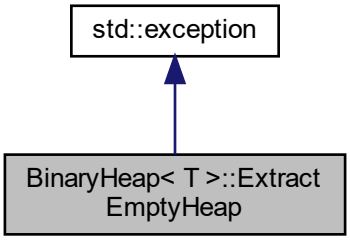
9.5 BinaryHeap< T >::ExtractEmptyHeap

#include <BinaryHeap.h>

BinaryHeap< T >::ExtractEmptyHeap :



BinaryHeap< T >::ExtractEmptyHeap :



9.5.1

template<class T>
class BinaryHeap< T >::ExtractEmptyHeap

BinaryHeap.h 147 .

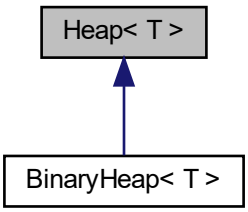
:

- E:/Artshell/inc/BinaryHeap.h

9.6 Heap< T >

#include <Heap.h>

Heap< T > :



Public

- Heap ()
- virtual ~Heap ()

  Destroy the Heap object Heap
- size_t size () const
- virtual void build (T data[], size_t size)=0
- virtual void insert (T value)
- virtual T top () const
- virtual T extract ()

Protected

- size_t size_

9.6.1

template<class T>
class Heap< T >

T	

0.1

(3200105842@zju.edu.cn)

2022-08-10

Heap.h 28 .

9.6.2

9.6.2.1 Heap()

template<class T >
Heap< T >::Heap ( ) [inline]

Heap.h 31 .

9.6.2.2 ∼Heap()

template<class T >
virtual Heap< T >::∼Heap ( ) [inline], [virtual]

Destroy the Heap object Heap

0.1

(3200105842@zju.edu.cn)

2022-08-10

Heap.h 43 .

9.6.3

9.6.3.1 build()

template<class T >
virtual void Heap< T >::build (
T data[],
size_t size) [pure virtual]

BinaryHeap< T > .

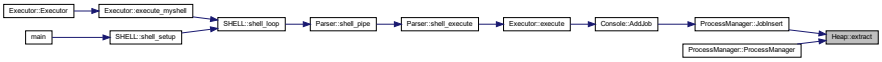
9.6.3.2 extract()

template<class T >
virtual T Heap< T >::extract ( ) [inline], [virtual]

BinaryHeap< T > .

Heap.h 60 .

:



9.6.3.3 insert()

template<class T >
virtual void Heap< T >::insert (
T value) [inline], [virtual]

BinaryHeap< T > .

Heap.h 49 .

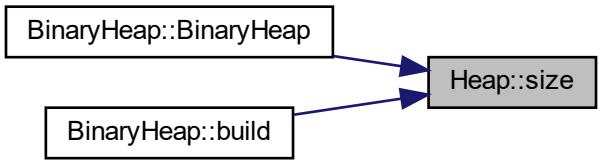
:

9.6.3.4 size()

template<class T >
size_t Heap< T >::size ( ) const [inline]

Heap.h 45 .

:



9.6.3.5 top()

template<class T >
virtual T Heap< T >::top ( ) const [inline], [virtual]

BinaryHeap< T > .

Heap.h 54 .

9.6.4

9.6.4.1 size_

template<class T >
size_t Heap< T >::size_ [protected]

Heap.h 67 .

:

- E:/Artshell/inc/Heap.h

9.7 job_unit

#include <ProcessManager.h>

Public

- job_unit (unsigned int _id, int _pid, job_state _state, int _argc, char *_argv[])
- void PrintJob (int output_fd=STDOUT_FILENO)
- bool operator== (const job_unit &rhs) const
- bool operator!= (const job_unit &rhs) const
- bool operator< (const job_unit &rhs) const
- bool operator> (const job_unit &rhs) const
- bool operator<= (const job_unit &rhs) const
- bool operator>= (const job_unit &rhs) const

Public

- unsigned int id
- pid_t pid
- job_state state
- int argc
- char argv [MAX_ARGUMENT_NUMBER][BUFFER_SIZE]

9.7.1

ProcessManager.h 30 .

9.7.2

9.7.2.1 job_unit()

```
job_unit::job_unit (
                unsigned int _id,
                int _pid,
                job_state _state,
                int _argc,
                char * _argv[] )
```

ProcessManager.cpp 12 .

9.7.3

9.7.3.1 operator"!=()

bool job_unit::operator!= (
const job_unit & rhs) const [inline]

ProcessManager.h 45 .

9.7.3.2 operator<()

bool job_unit::operator< (
const job_unit & rhs) const [inline]

ProcessManager.h 50 .

9.7.3.3 operator<=()

bool job_unit::operator<= (
const job_unit & rhs) const [inline]

ProcessManager.h 60 .

9.7.3.4 operator==()

bool job_unit::operator== (
const job_unit & rhs) const [inline]

ProcessManager.h 40 .

9.7.3.5 operator>()

bool job_unit::operator> (
const job_unit & rhs) const [inline]

ProcessManager.h 55 .

9.7.3.6 operator>=()

bool job_unit::operator>= (
const job_unit & rhs) const [inline]

ProcessManager.h 65 .

9.7.3.7 PrintJob()

void job_unit::PrintJob (
int output_fd = STDOUT_FILENO)

ProcessManager.cpp 21 .

:

9.7.4

9.7.4.1 argc

int job_unit::argc

ProcessManager.h 74 .

9.7.4.2 argv

char job_unit::argv[MAX_ARGUMENT_NUMBER][BUFFER_SIZE]

ProcessManager.h 75 .

9.7.4.3 id

unsigned int job_unit::id

ProcessManager.h 71 .

9.7.4.4 pid

pid_t job_unit::pid

ProcessManager.h 72 .

9.7.4.5 state

job_state job_unit::state

ProcessManager.h 73 .

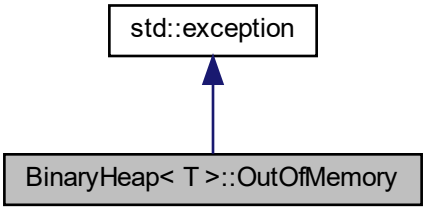
:

- E:/Artshell/inc/ProcessManager.h
- E:/Artshell/src/ProcessManager.cpp

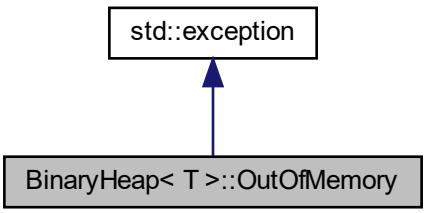
9.8 BinaryHeap< T >::OutOfMemory

#include <BinaryHeap.h>

BinaryHeap< T >::OutOfMemory :



BinaryHeap< T >::OutOfMemory :



9.8.1

template<class T>
class BinaryHeap< T >::OutOfMemory

BinaryHeap.h 148 .

:

- E:/Artshell/inc/BinaryHeap.h

9.9 Parser

#include <Parser.h>

Public

- Parser ()
- virtual ~Parser ()=0

  Destroy the Parser object

Public

- static bool shell_pipe (Console *model, Display *view, Executor *controller, int &argc, char *argv[], char *env[])
- static int shell_parser (Console *model, Display *view, Executor *controller, int &argc, char *argv[], char *env[])

Private

- enum { SUCCESS = 0 , EXIT = 1 }

Private

- static bool shell_execute (Console *model, Display *view, Executor *controller, int &argc, char *argv[], char *env[])

  shell

9.9.1

Parser.h 19 .

9.9.2

9.9.2.1 anonymous enum

anonymous enum [private]

SUCCESS	
EXIT	

Parser.h 22 .

9.9.3

9.9.3.1 Parser()

Parser::Parser ( ) [inline]

Parser.h 43 .

9.9.3.2 ∼Parser()

virtual Parser::∼Parser ( ) [pure virtual]

Destroy the Parser object

0.1

(3200105842@zju.edu.cn)

2022-07-19

9.9.4

9.9.4.1 shell_execute()

bool Parser::shell_execute (
Console * model,
Display * view,
Executor * controller,
int & argc,
char * argv[],
char * env[]) [static], [private]

shell

model	
view	
controller	
argc	
argv	
env	

true
false

0.1

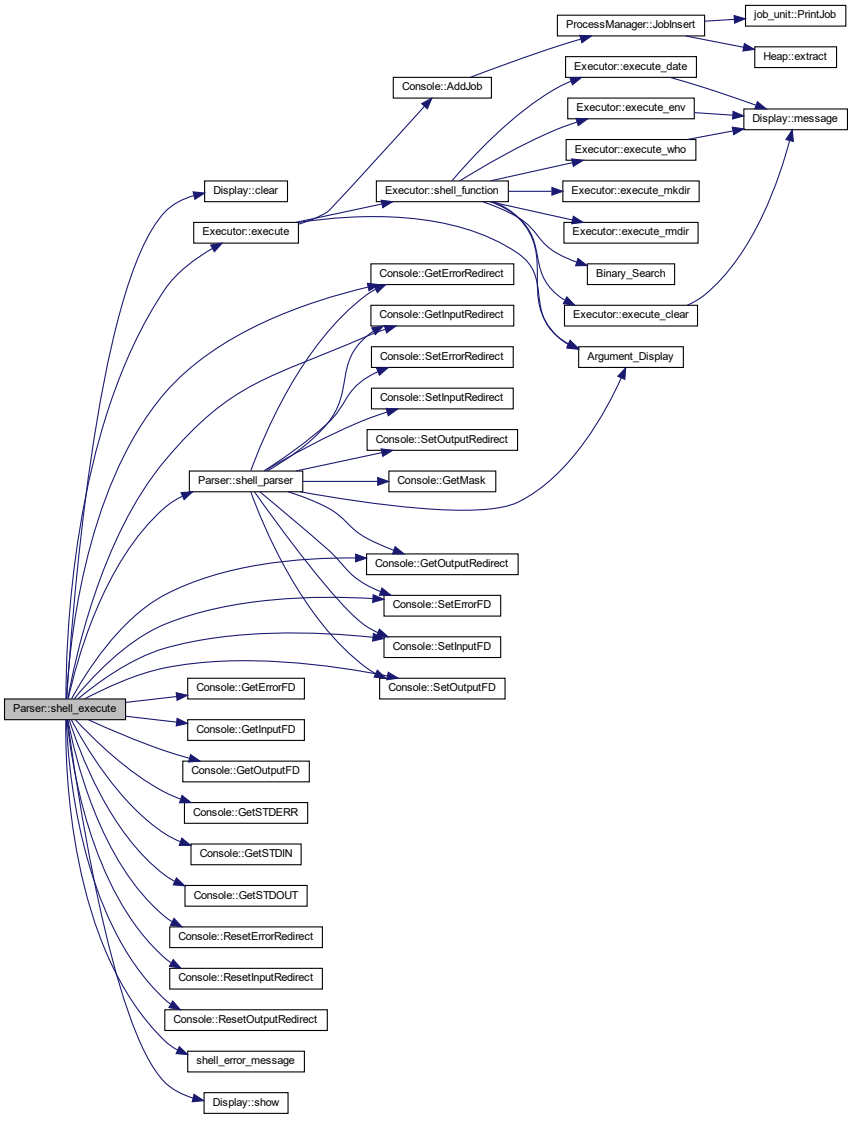
(3200105842@zju.edu.cn)

2022-07-19

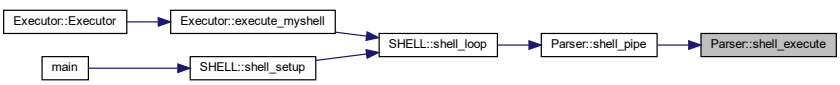
Copyright (c) 2022

Parser.cpp 268 .

:



:

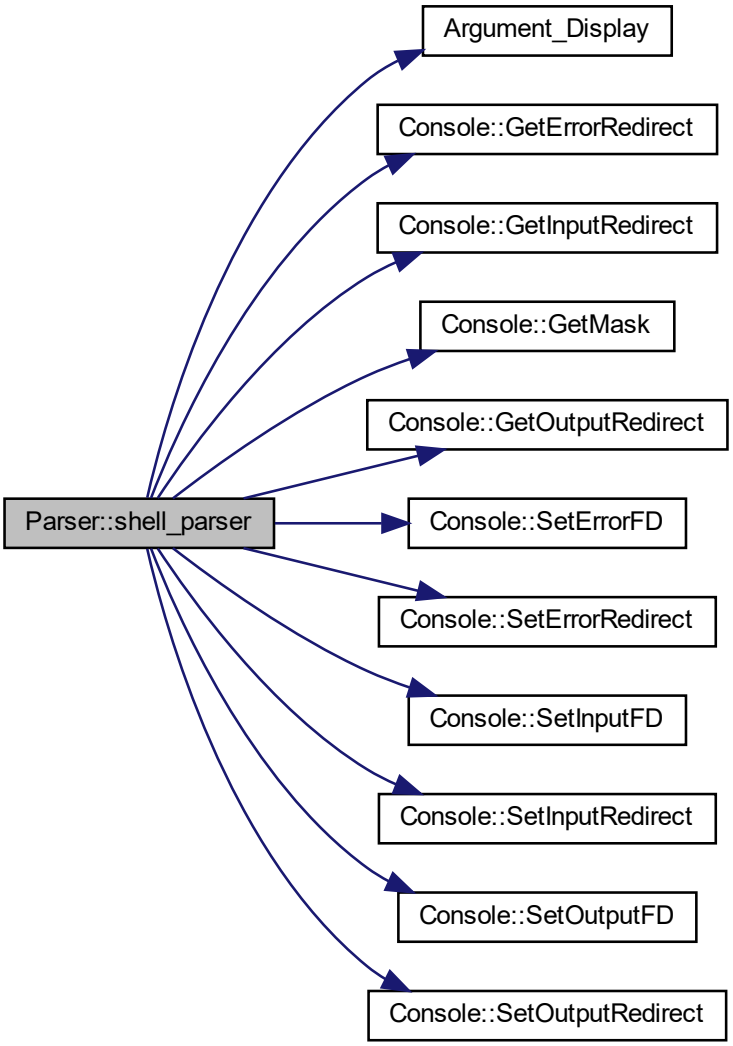


9.9.4.2 shell_parser()

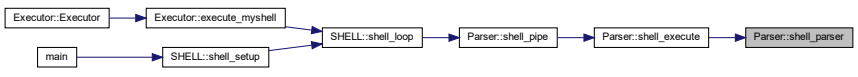
```
int Parser::shell_parser (
                Console * model,
                Display * view,
                Executor * controller,
                int & argc,
                char * argv[],
                char * env[] )    [static]
```

Parser.cpp 132 .

:



:



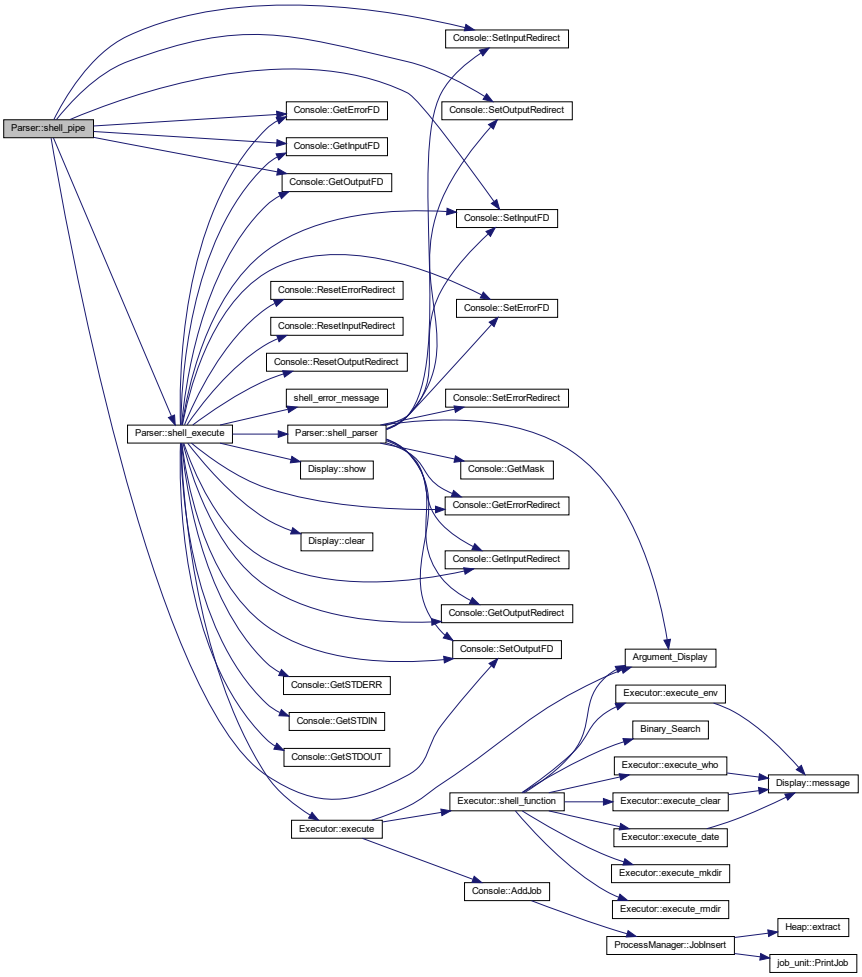
9.9.4.3 shell_pipe()

bool Parser::shell_pipe (
Console * model,

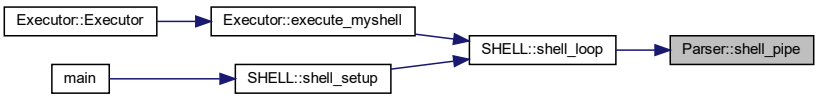
Display * view,
Executor * controller,
int & argc,
char * argv[],
char * env[]) [static]

Parser.cpp 27 .

:



:



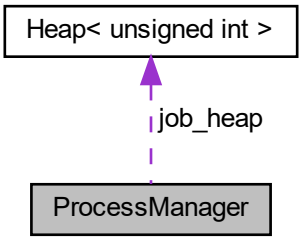
:

- E:/Artshell/inc/Parser.h
- E:/Artshell/src/Parser.cpp

9.10 ProcessManager

#include <ProcessManager.h>

ProcessManager :



Public

- ProcessManager ()
- virtual ~ProcessManager ()
- void PrintJobList (int output_fd=STDOUT_FILENO) const
- void PrintJobListDone (int output_fd=STDOUT_FILENO)
- unsigned int JobInsert (int pid, job_state state, int argc, char *argv[])

- void JobRemove (job_unit *job)

- void JobRemove (std::set< job_unit >::iterator &job)
- int ForeGround (unsigned int jobid)
- int BackGround (unsigned int jobid)

Private

- Heap< unsigned int > * job_heap
- std::set< class job_unit > jobs

9.10.1

ProcessManager.h 78 .

9.10.2

9.10.2.1 ProcessManager()

ProcessManager::ProcessManager ( )

ProcessManager.cpp 77 .

:

9.10.2.2 ~ProcessManager()

ProcessManager::~ProcessManager ( ) [virtual]

ProcessManager.cpp 90 .

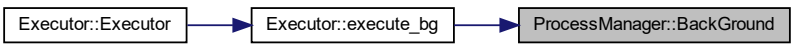
9.10.3

9.10.3.1 BackGround()

int ProcessManager::BackGround (
unsigned int jobid)

ProcessManager.cpp 205 .

:

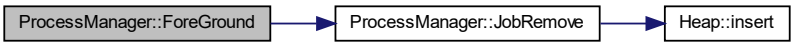


9.10.3.2 ForeGround()

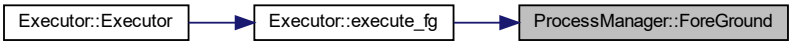
int ProcessManager::ForeGround (
unsigned int jobid)

ProcessManager.cpp 176 .

:



:



9.10.3.3 JobInsert()

unsigned int ProcessManager::JobInsert (
int pid,
job_state state,
int argc,
char * argv[])

pid	
state	
argc	
argv	

unsigned int 0

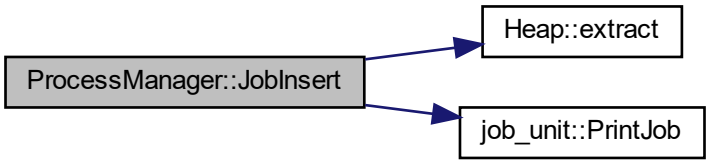
0.1

(3200105842@zju.edu.cn)

2022-07-20

ProcessManager.cpp 140 .

:



:

9.10.3.4 JobRemove() [1/2]

void ProcessManager::JobRemove (
job_unit * job)

job	

0.1

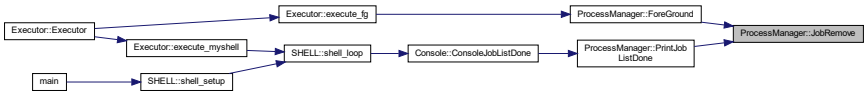
(3200105842@zju.edu.cn)

2022-07-21

ProcessManager.cpp 159 .

:

:

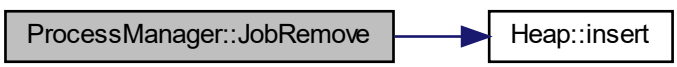


9.10.3.5 JobRemove() [2/2]

void ProcessManager::JobRemove (
std::set< job_unit >::iterator & job)

ProcessManager.cpp 168 .

:



9.10.3.6 PrintJobList()

void ProcessManager::PrintJobList (
int output_fd = STDOUT_FILENO) const

ProcessManager.cpp 95 .

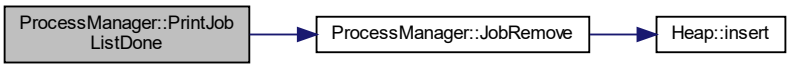
:

9.10.3.7 PrintJobListDone()

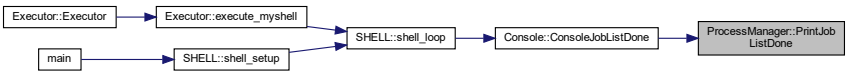
void ProcessManager::PrintJobListDone (
int output_fd = STDOUT_FILENO)

ProcessManager.cpp 103 .

:



:



9.10.4

9.10.4.1 job_heap

Heap<unsigned int>* ProcessManager::job_heap [private]

ProcessManager.h 82 .

9.10.4.2 jobs

std::set<class job_unit> ProcessManager::jobs [private]

ProcessManager.h 83 .

:

- E:/Artshell/inc/ProcessManager.h
- E:/Artshell/src/ProcessManager.cpp

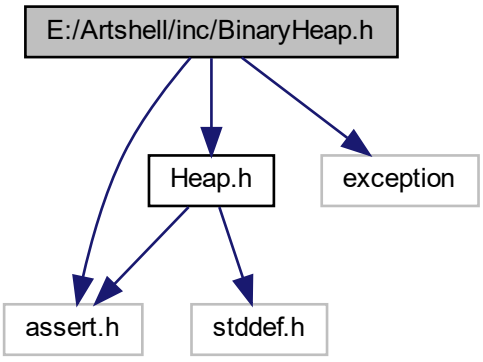
Chapter 10

10.1 E:/Artshell/doc/ .md

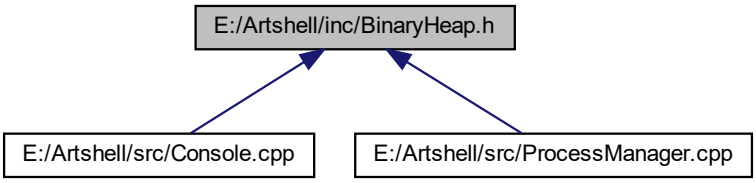
10.2 E:/Artshell/doc/ .md

10.3 E:/Artshell/inc/BinaryHeap.h

#include ”Heap.h”
#include <assert.h>
#include <exception>
BinaryHeap.h (Include) :



:



- class BinaryHeap< T >

- class BinaryHeap< T >::ExtractEmptyHeap
- class BinaryHeap< T >::OutOfMemory

- static constexpr size_t HeapBlockSize = 1024

10.3.1

(3200105842@zju.edu.cn)

0.1

2022-07-20

Copyright (c) 2022

BinaryHeap.h .

10.3.2

10.3.2.1 HeapBlockSize

constexpr size_t HeapBlockSize = 1024 [static], [constexpr]

BinaryHeap.h 19 .

10.4 BinaryHeap.h

.

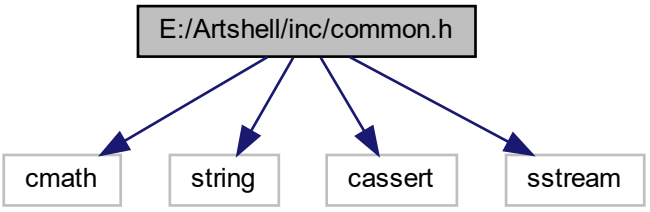
```
00001
00012 #ifndef __BINARY_HEAP_H_
00013 #define __BINARY_HEAP_H_
00014
00015 #include "Heap.h"
00016 #include <assert.h>
00017 #include <exception>
00018
00019 static constexpr size_t HeapBlockSize = 1024;   //
00020
00021 // template <class T>
00022 // static constexpr T INF = -0x7f7f7f7f;         //
00023
00033 template <class T>
00034 class BinaryHeap : public Heap<T>
00035 {
00036     //              using
00037     //   this
00038     using Heap<T>::size_;
00039
00040     public:
00041         BinaryHeap(size_t heap_capacity = HeapBlockSize)
00042         : Heap<T>(), capacity_(heap_capacity)
00043         {
00044             assert(heap_capacity > 0);
00045
00046             node = new T[heap_capacity+1];  //
00047             if (node == NULL)               //
00048                 throw OutOfMemory();
00049         }
00050
00051         BinaryHeap(T data[], size_t size, size_t heap_capacity = HeapBlockSize)
00052         : Heap<T>(), capacity_(heap_capacity)
00053         {
00054             node = new T[(size>heap_capacity?size:heap_capacity) + 1];  //
00055             if (node == NULL)               //
00056                 throw OutOfMemory();
00057
00058             size_ = size;
00059             for (size_t i = 1; i <= size; ++i) //
00060                 node[i] = data[i-1];
00061
00062             build_heap();                    //
00063         }
00064
00065         virtual ~BinaryHeap()
00066         {
00067             delete [] node;
00068         }
00069
00070         virtual void build(T data[], size_t size)
00071         {
00072             while (capacity_ < size)    //
00073                 AllocMoreSpace();
00074             size_ = size;
00075             for (size_t i = 0; i < size; ++i)   //
00076                 node[i+1] = data[i];
00077
00078             for (size_t i = (size_»1); i>0; --i)    // n/2
00079             {
00080                 size_t p, child;
00081                 T X = node[i];
00082                 for (p = i; (p«1) <= size_; p = child) //
00083                 {
00084                     child=(p«1);   //
00085                     if (child != size_ && node[child+1] < node[child])
00086                         ++child;
00087
00088                     if (X > node[child])
00089                         node[p] = node[child];
```

```
00090                 else
00091                     break;
00092             }
00093             node[p] = X;
00094         }
00095     }
00096
00097     virtual void insert(T value)
00098     {
00099         if (size_ + 2 >= capacity_)
00100         {
00101             AllocMoreSpace();    //
00102         }
00103
00104         int p;
00105         for (p = ++size_; node[p»1] > value && p > 1; p = p»1)    //
00106             node[p] = node[p»1];   //   swap
00107         node[p] = value; //
00108     }
00109
00110     virtual T top() const
00111     {
00112         if (size_ == 0)
00113             throw ExtractEmptyHeap();
00114         return node[1];
00115     }
00116
00117     virtual T extract()
00118     {
00119         if (size_ == 0)   //
00120             throw ExtractEmptyHeap();    //
00121
00122         T top, last;
00123         top = node[1];
00124         last = node[size_--];
00125
00126         size_t p, child;
00127         for (p = 1; (p«1) <= size_; p = child)  //
00128         {
00129             child = (p«1);   //
00130             if (child != size_ && node[child+1] < node[child])
00131                 ++child;
00132
00133             if (last >node[child])    //
00134                 node[p]=node[child];  //
00135             else
00136                 break;
00137         }
00138
00139         node[p] = last;
00140         return top;
00141     }
00142
00143 protected:
00144     size_t capacity_;  //
00145     T *node;           //
00146
00147     class ExtractEmptyHeap : public std::exception {};
00148     class OutOfMemory : public std::exception {};
00149
00150     void AllocMoreSpace()   //
00151     {
00152         capacity_«=1;      //
00153         T *newNode = new T[capacity_];
00154         if (newNode == NULL)
00155         {
00156             throw OutOfMemory();    //
00157         }
00158
00159         for (size_t i = 0; i < size_; ++i)
00160             std::swap(node[i], newNode[i]); //
00161         delete [] node;
00162         node = std::move(newNode);          //
00163     }
00164
00165 private:
00166     void build_heap()
00167     {
00168         for (size_t i = (size_»1); i>0; --i)    //  n/2
00169         {
00170             size_t p, child;
00171             T X = node[i];
00172             for (p = i; (p«1) <= size_; p = child) //percolate down
00173             {
00174                 child=(p«1);   //
00175                 if (child != size_ && node[child+1] < node[child])
00176                     ++child;
```

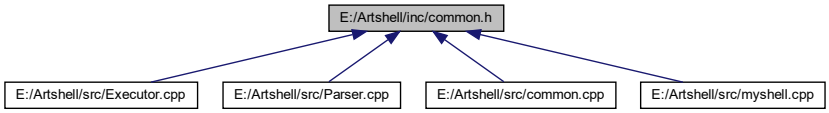
```
00177
00178                 if (X > node[child])    //
00179                     node[p] = node[child];  //
00180                 else
00181                     break;
00182             }
00183             node[p] = X;  //
00184         }
00185     }
00186 };
00187
00188 #endif
```

10.5 E:/Artshell/inc/common.h

#include <cmath>
#include <string>
#include <cassert>
#include <sstream>
common.h (Include) :



:



- #define ASSERT(expr, message) assert((expr) && (message))

- void Argument_Display (const int argc, char *const argv[])
- template<typename T >
  int Binary_Search (int left, int right, T val, T array[], int cmp(T a, T b))
  [l, r)
- std::string & String_Trim (std::string &s)
- template<class Type >
  Type String_to_Number (const std::string &str)
- template<typename T >
  T Min (const T &a, const T &b)
- template<typename T >
  T Max (const T &a, const T &b)
- template<typename T >
  T Octal_to_Decimal (T octalNumber)
- template<typename T >
  T Decimal_to_Octal (T decimalNumber)
- template<typename T >
  T Hexadecimal_to_Decimal (T hexadecimalNumber)
- template<typename T >
  T Decimal_to_Hexadecimal (T decimalNumber)
- bool test_timespec_newer (struct timespec &time1, struct timespec &time2)
  timespec
- bool test_timespec_older (struct timespec &time1, struct timespec &time2)
  timespec

10.5.1

(3200105842@zju.edu.cn)

0.1

2022-07-15

Copyright (c) 2022

common.h .

10.5.2

10.5.2.1 ASSERT

```
#define ASSERT(
                expr,
                message ) assert((expr) && (message))
```

common.h 21 .

10.5.3

10.5.3.1 Argument_Display()

```
void Argument_Display (
                const int argc,
                char *const argv[] )
```

argc	
argv	

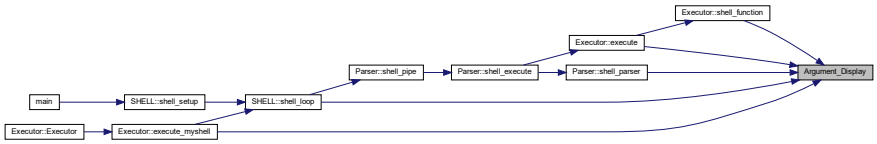
0.1

(3200105842@zju.edu.cn)

2022-07-15

common.cpp 16 .

:



10.5.3.2 Binary_Search()

```
template<typename T >
int Binary_Search (
                int left,
                int right,
                T val,
                T array[],
                int  cmpT a, T b )
```

[l, r)

T	
Tp	

lelt	
right	
val	
array	
cmp	

int -1

0.1

(3200105842@zju.edu.cn)

2022-07-17

Copyright (c) 2022

common.h 54 .

:

10.5.3.3 Decimal_to_Hexadecimal()

```
template<typename T >
T Decimal_to_Hexadecimal (
                T decimalNumber )
```

T	

decimalNumber	

T

0.1

(3200105842@zju.edu.cn)

2022-07-19

Copyright (c) 2022

common.h 204 .

10.5.3.4 Decimal_to_Octal()

```
template<typename T >
T Decimal_to_Octal (
                T decimalNumber )
```

T	

decimalNumber	

T

0.1

(3200105842@zju.edu.cn)

2022-07-19

common.h 154 .

10.5.3.5 Hexadecimal_to_Decimal()

```
template<typename T >
T Hexadecimal_to_Decimal (
                T hexadecimalNumber )
```

T	

hexadecimalNumber	

T

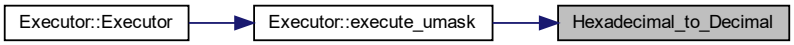
0.1

(3200105842@zju.edu.cn)

2022-07-19

common.h 179 .

:



10.5.3.6 Max()

```
template<typename T >
T Max (
                const T & a,
                const T & b )    [inline]
```

common.h 112 .

10.5.3.7 Min()

```
template<typename T >
T Min (
                const T & a,
                const T & b )   [inline]
```

common.h 105 .

10.5.3.8 Octal_to_Decimal()

```
template<typename T >
T Octal_to_Decimal (
                T octalNumber )
```

T	

octalNumber	

T

0.1

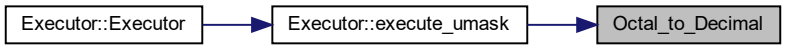
(3200105842@zju.edu.cn)

2022-07-19

Copyright (c) 2022

common.h 129 .

:



10.5.3.9 String_to_Number()

```
template<class Type >
Type String_to_Number (
                const std::string & str )
```

Type	

str	

Type

0.1

(3200105842@zju.edu.cn)

2022-07-18

Copyright (c) 2022

common.h 95 .

10.5.3.10 String_Trim()

std::string & String_Trim (
std::string & s)

s	

std::string&

0.1

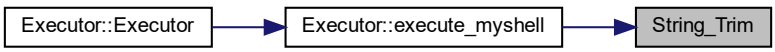
(3200105842@zju.edu.cn)

2022-07-17

Copyright (c) 2022

common.cpp 27 .

:



10.5.3.11 test_timespec_newer()

bool test_timespec_newer (
struct timespec & time1,
struct timespec & time2) [inline]

timespec

time1	1
time2	2

true time1 time2
false time1 time2

0.1

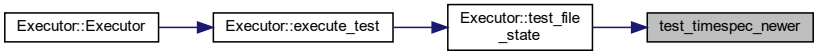
(3200105842@zju.edu.cn)

2022-07-20

Copyright (c) 2022

common.h 229 .

:



10.5.3.12 test_timespec_older()

bool test_timespec_older (
struct timespec & time1,
struct timespec & time2) [inline]

timespec

time1	1
time2	2

true time1 time2

false time1 time2

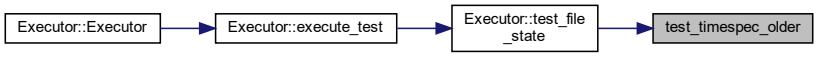
0.1

(3200105842@zju.edu.cn)

2022-07-20

common.h 251 .

:



10.6 common.h

.

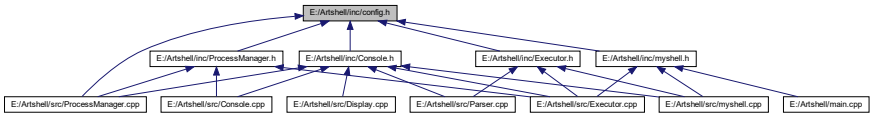
```
00001
00012 #ifndef __COMMON_H_
00013 #define __COMMON_H_
00014
00015 #include <cmath>
00016 #include <string>
00017 #include <cassert>
00018 #include <sstream>
00019
00020 //
00021 #define ASSERT(expr, message) assert((expr) && (message))
00022
00035 void Argument_Display(const int argc, char* const argv[]);
00036
00053 template<typename T>
00054 int Binary_Search(int left, int right, T val, T array[], int cmp(T a, T b))
00055 {
00056     while (left < right)
00057     {
00058         int mid = (left + right) » 1;
00059         int compare_result = cmp(val, array[mid]);
00060         if (compare_result == 0)
00061             return mid;
00062         else if (compare_result > 0)
00063             left = mid + 1;
00064         else
00065             right = mid;
00066     }
00067
00068     return -1;
00069 }
```

```
00070
00081 std::string& String_Trim(std::string &s);
00082
00094 template <class Type>
00095 Type String_to_Number(const std::string& str)
00096 {
00097     std::istringstream iss(str);
00098     Type num;
00099     iss » num;
00100     return num;
00101 }
00102
00104 template <typename T>
00105 inline T Min(const T& a, const T& b)
00106 {
00107     return a < b ? a : b;
00108 }
00109
00111 template <typename T>
00112 inline T Max(const T& a, const T& b)
00113 {
00114     return a > b ? a : b;
00115 }
00116
00128 template <typename T>
00129 T Octal_to_Decimal(T octalNumber)
00130 {
00131     T decimalNumber = 0, i = 0, remainderNumber;
00132     while (octalNumber != 0)
00133     {
00134         remainderNumber = octalNumber % 10; //
00135         octalNumber /= 10;                  //
00136         decimalNumber += remainderNumber * pow(8, i);   //
00137         ++i;
00138     }
00139     return decimalNumber;
00140 }
00141
00153 template <typename T>
00154 T Decimal_to_Octal(T decimalNumber)
00155 {
00156     T remainderNumber, i = 1, octalNumber = 0;
00157     while (decimalNumber != 0)
00158     {
00159         remainderNumber = decimalNumber % 8; //
00160         decimalNumber /= 8;                  //
00161         octalNumber += remainderNumber * i;  //
00162         i *= 10;
00163     }
00164     return octalNumber;
00165 }
00166
00178 template <typename T>
00179 T Hexadecimal_to_Decimal(T hexadecimalNumber)
00180 {
00181     T decimalNumber = 0, i = 0, remainderNumber;
00182     while (hexadecimalNumber != 0)
00183     {
00184         remainderNumber = hexadecimalNumber % 10; //
00185         hexadecimalNumber /= 10;                  //
00186         decimalNumber += remainderNumber * pow(16, i);  //
00187         ++i;
00188     }
00189     return decimalNumber;
00190 }
00191
00203 template <typename T>
00204 T Decimal_to_Hexadecimal(T decimalNumber)
00205 {
00206     T remainderNumber, i = 1, hexadecimalNumber = 0;
00207     while (decimalNumber != 0)
00208     {
00209         remainderNumber = decimalNumber % 16; //
00210         decimalNumber /= 16;                  //
00211         hexadecimalNumber += remainderNumber * i;  //
00212         i *= 10;
00213     }
00214     return hexadecimalNumber;
00215 }
00216
00229 inline bool test_timespec_newer(struct timespec& time1, struct timespec& time2)
00230 {
00231     if (time1.tv_sec > time2.tv_sec)    //
00232         return true;
00233     else if (time1.tv_sec < time2.tv_sec)
00234         return false;
00235     else
```

```
00236         return time1.tv_nsec > time2.tv_nsec;   //
00237 }
00238
00251 inline bool test_timespec_older(struct timespec& time1, struct timespec& time2)
00252 {
00253     if (time1.tv_sec < time2.tv_sec)    //
00254         return true;
00255     else if (time1.tv_sec > time2.tv_sec)
00256         return false;
00257     else
00258         return time1.tv_nsec < time2.tv_nsec;   //
00259 }
00260
00261 #endif
```

10.7 E:/Artshell/inc/config.h

:



- enum sh_err_t {
  SH_SUCCESS = 0 , SH_FAILED , SH_UNDEFINED , SH_ARGS ,
  SH_EXIT }
- enum job_state { Running , Stopped , Done , Terminated }

- unsigned int constexpr String_Hash (char const *input, unsigned int prime=hash_prime, unsigned int basis=hash_basis)

- static constexpr int BUFFER_SIZE = 1024
- static constexpr int MAX_PROCESS_NUMBER = 1024
- static constexpr int MAX_ARGUMENT_NUMBER = 128
- constexpr unsigned int hash_prime = 33u
- constexpr unsigned int hash_basis = 5381u

10.7.1

(3200105842@zju.edu.cn)

0.1

2022-07-03

Copyright (c) 2022

config.h .

10.7.2

10.7.2.1 job_state

enum job_state

Running	
Stopped	
Done	
Terminated	

config.h 28 .

10.7.2.2 sh_err_t

enum sh_err_t

SH_SUCCESS	
SH_FAILED	
SH_UNDEFINED	
SH_ARGS	
SH_EXIT	

config.h 19 .

10.7.3

10.7.3.1 String_Hash()

unsigned int constexpr String_Hash (
char const * input,
unsigned int prime = hash_prime,
unsigned int basis = hash_basis) [constexpr]

input	

unsigned constexpr

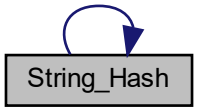
0.1

(3200105842@zju.edu.cn)

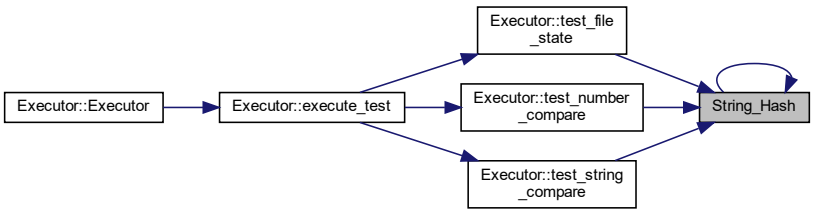
2022-07-19

config.h 49 .

:



:



10.7.4

10.7.4.1 BUFFER_SIZE

constexpr int BUFFER_SIZE = 1024 [static], [constexpr]

config.h 15 .

10.7.4.2 hash_basis

constexpr unsigned int hash_basis = 5381u [constexpr]

config.h 37 .

10.7.4.3 hash_prime

constexpr unsigned int hash_prime = 33u [constexpr]

config.h 36 .

10.7.4.4 MAX_ARGUMENT_NUMBER

constexpr int MAX_ARGUMENT_NUMBER = 128 [static], [constexpr]

config.h 17 .

10.7.4.5 MAX_PROCESS_NUMBER

constexpr int MAX_PROCESS_NUMBER = 1024 [static], [constexpr]

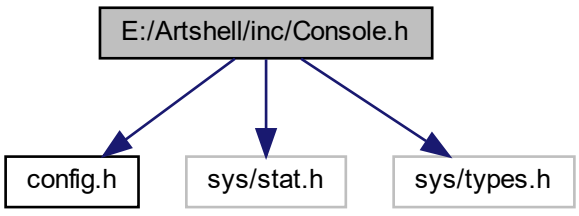
config.h 16 .

10.8 config.h

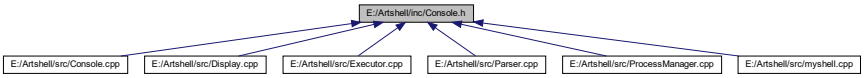
```
00001
00012 #ifndef __CONFIG__H__
00013 #define __CONFIG__H__
00014
00015 static constexpr int BUFFER__SIZE = 1024;        //
00016 static constexpr int MAX__PROCESS__NUMBER = 1024; //
00017 static constexpr int MAX__ARGUMENT__NUMBER = 128; //
00018
00019 enum sh__err__t   //  shell
00020 {
00021     SH__SUCCESS = 0, //
00022     SH__FAILED,      //
00023     SH__UNDEFINED,   //
00024     SH__ARGS,        //
00025     SH__EXIT,        //
00026 };
00027
00028 enum job__state                              //
00029 {
00030     Running,                                //
00031     Stopped,                                //
00032     Done,                                   //
00033     Terminated                               //
00034 };
00035
00036 constexpr unsigned int hash__prime = 33u;   //
00037 constexpr unsigned int hash__basis = 5381u; //
00038
00049 unsigned int constexpr String__Hash(char const *input, unsigned int prime = hash__prime, unsigned int basis =
       hash__basis)
00050 {
00051     return *input ?                                                  //
00052         static__cast<unsigned int>(*input) + prime * String__Hash(input + 1) : //
00053         basis;                                                      //
00054 }
00055
00056 #endif
```

10.9 E:/Artshell/inc/Console.h

#include ”config.h”
#include <sys/stat.h>
#include <sys/types.h>
Console.h (Include) :



:



- class Console

- void SignalHandler (int signal_)

10.9.1

(3200105842@zju.edu.cn)

0.1

2022-07-03

Copyright (c) 2022

Console.h .

10.9.2

10.9.2.1 SignalHandler()

void SignalHandler (
int signal_)

signal↩ __	

0.1

(3200105842@zju.edu.cn)

2022-07-21

Console.cpp 49 .

10.10 Console.h

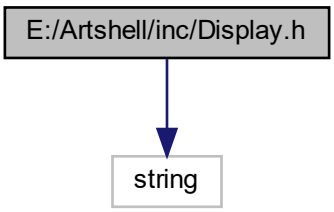
.

```
00001
00012 #ifndef __CONSOLE_H__
00013 #define __CONSOLE_H__
00014
00015 #include "config.h"
00016
00017 #include <sys/stat.h>
00018 #include <sys/types.h>
00019
00020 class ProcessManager;   //
00021
00031 void SignalHandler(int signal_);
00032
00038 class Console
00039 {
00040     private:
00041         //
00042         char user_name[BUFFER_SIZE];                    //
00043         char host_name[BUFFER_SIZE];                    //
00044         char current_working_dictionary[BUFFER_SIZE];   //
00045
00046         char home[BUFFER_SIZE];                         //
00047
00048         //
00049         char shell_path_env[BUFFER_SIZE];               // shell
00050
00051         //
00052         pid_t process_id;                           //    pid
00053         static pid_t child_process_id;              //    pid
00054         ProcessManager* process_manager;            //
00055
00056         //
00057         int input_file_descriptor;                  //
00058         int output_file_descriptor;                 //
00059         int error_file_descriptor;                  //
00060
00061         //
00062         static int input_std_fd;                    //
00063         static int output_std_fd;                   //
00064         static int error_std_fd;                    //
00065
00066         //
```

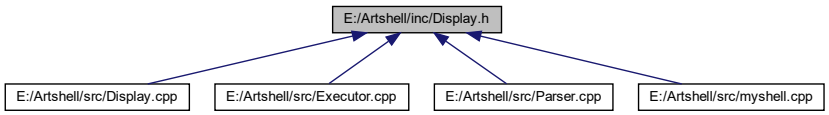
```
00067         bool redirect_input;                          //
00068         bool redirect_output;                         //
00069         bool redirect_error;                          //
00070
00071         //
00072         mode_t umask_;                                //
00073
00074         int argc;                                    //
00075         char argv[MAX_ARGUMENT_NUMBER][BUFFER_SIZE];     //
00076
00077     public:
00078         Console(/* args */);
00079
00080         virtual ~Console();
00081
00082         /*    */
00083         int init();
00084
00085         /*      */
00086         void ConsoleJobList() const;
00087
00088         /*         */
00089         void ConsoleJobListDone();
00090
00091         /*    */
00092         unsigned int AddJob(int pid, job_state state, int argc, char *argv[]);
00093
00094         // void RemoveJob();
00095
00096         void ResetChildPid() { child_process_id = -1; }
00097
00098         /*       */
00099         void SetInputFD(int _fd)  { input_file_descriptor = _fd; }
00100         /*       */
00101         void SetOutputFD(int _fd) { output_file_descriptor = _fd; }
00102         /*       */
00103         void SetErrorFD(int _fd)  { error_file_descriptor = _fd; }
00104
00105         /*       */
00106         int GetInputFD() const  { return input_file_descriptor; }
00107         /*       */
00108         int GetOutputFD() const { return output_file_descriptor; }
00109         /*       */
00110         int GetErrorFD() const  { return error_file_descriptor; }
00111
00112         /*       */
00113         void SetInputRedirect()  { redirect_input = true; }
00114         /*       */
00115         void SetOutputRedirect() { redirect_output = true; }
00116         /*       */
00117         void SetErrorRedirect()  { redirect_error = true; }
00118
00119         /*       */
00120         void ResetInputRedirect()  { redirect_input = false; }
00121         /*       */
00122         void ResetOutputRedirect() { redirect_output = false; }
00123         /*       */
00124         void ResetErrorRedirect()  { redirect_error = false; }
00125
00126         /*       */
00127         bool GetInputRedirect() const  { return redirect_input ; }
00128         /*       */
00129         bool GetOutputRedirect() const { return redirect_output; }
00130         /*       */
00131         bool GetErrorRedirect() const  { return redirect_error ; }
00132
00133         /*           */
00134         int GetSTDIN()  const { return input_std_fd;  }
00135         /*           */
00136         int GetSTDOUT() const { return output_std_fd; }
00137         /*           */
00138         int GetSTDERR() const { return error_std_fd;  }
00139
00140         /*    */
00141         void SetMask(mode_t _mask) { umask_ = _mask; }
00142         /*    */
00143         mode_t GetMask() const { return umask_; }
00144
00145         friend class Display;
00146         friend class Executor;
00147         friend class ProcessManager;
00148         friend void SignalHandler(int);
00149 };
00150
00151 #endif
```

10.11 E:/Artshell/inc/Display.h

#include <string>
Display.h (Include) :



:



- class Display

10.11.1

(3200105842@zju.edu.cn)

0.1

2022-07-03

Copyright (c) 2022

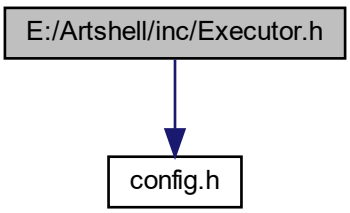
Display.h .

10.12 Display.h

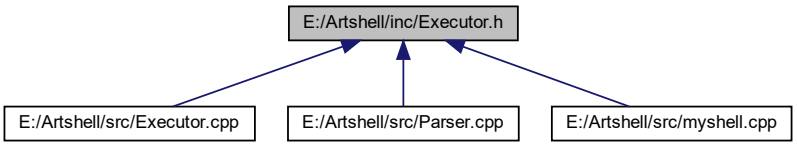
```
00001
00012 #ifndef __DISPLAY__H__
00013 #define __DISPLAY__H__
00014
00015 class Console;
00016
00017 #include <string>
00018
00019 class Display
00020 {
00021     private:
00022
00023         Console* console_;
00024
00025         bool perform;   //
00026
00027     protected:
00028         std::string buffer_;
00029
00030     public:
00031         Display(Console* console);
00032
00033         virtual ~Display();
00034
00040         int InputCommand(char *input, const int len);
00041
00043         void render();
00044
00046         void prompt() const;
00047
00049         void message(const char * msg);
00050
00052         void show() const;
00053
00055         void clear() { buffer_ = ""; }
00056 };
00057
00058 #endif
```

10.13 E:/Artshell/inc/Executor.h

#include ”config.h”
Executor.h (Include) :



:



- class Executor

- static constexpr int FunctionNumber = 16

10.13.1

(3200105842@zju.edu.cn)

0.1

2022-07-04

Copyright (c) 2022

Executor.h .

10.13.2

10.13.2.1 FunctionNumber

constexpr int FunctionNumber = 16 [static], [constexpr]

Executor.h 20 .

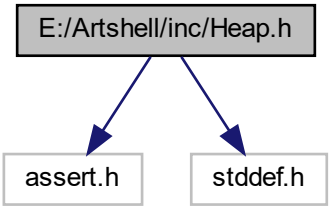
10.14 Executor.h

```
00001
00012 #ifndef __EXECUTOR_H_
00013 #define __EXECUTOR_H_
00014
00015 #include "config.h"
00016
00017 class Console;
00018 class Display;
00019
00020 static constexpr int FunctionNumber = 16;
00021
00022 class Executor
00023 {
00024     private:
00025
00026         Console *console_;
00028         Display *display_;
00030     protected:
00031
00033         sh_err_t shell_function(const int argc, char * const argv[], char * const env[]) const;
00034
00036         sh_err_t execute_cd(const int argc, char * const argv[], char * const env[]) const;
00037
00039         sh_err_t execute_pwd(const int argc, char * const argv[], char * const env[]) const;
00040
00042         sh_err_t execute_time(const int argc, char * const argv[], char * const env[]) const;
00043
00045         sh_err_t execute_clr(const int argc, char * const argv[], char * const env[]) const;
00046
00047         /*      */
00048         sh_err_t execute_dir(const int argc, char * const argv[], char * const env[]) const;
00049
00050         /*       */
00051         sh_err_t execute_set(const int argc, char * const argv[], char * const env[]) const;
00052
00053         /*   */
00054         sh_err_t execute_echo(const int argc, char * const argv[], char * const env[]) const;
00055
00056         /*      */
00057         sh_err_t execute_help(const int argc, char * const argv[], char * const env[]) const;
00058
00060         sh_err_t execute_exit(const int argc, char * const argv[], char * const env[]) const;
00061
00063         sh_err_t execute_date(const int argc, char * const argv[], char * const env[]) const;
00064
00066         sh_err_t execute_clear(const int argc, char * const argv[], char * const env[]) const;
00067
00069         sh_err_t execute_env(const int argc, char * const argv[], char * const env[]) const;
00070
00072         sh_err_t execute_who(const int argc, char * const argv[], char * const env[]) const;
00073
00075         sh_err_t execute_mkdir(const int argc, char * const argv[], char * const env[]) const;
00076
00078         sh_err_t execute_rmdir(const int argc, char * const argv[], char * const env[]) const;
00079
00081         sh_err_t execute_bg(const int argc, char * const argv[], char * const env[]) const;
00082
00084         sh_err_t execute_fg(const int argc, char * const argv[], char * const env[]) const;
00085
00087         sh_err_t execute_jobs(const int argc, char * const argv[], char * const env[]) const;
00088
00090         sh_err_t execute_exec(const int argc, char * const argv[], char * const env[]) const;
00091
00093         sh_err_t execute_test(const int argc, char * const argv[], char * const env[]) const;
00094
00096         sh_err_t execute_umask(const int argc, char * const argv[], char * const env[]) const;
00097
00099         sh_err_t execute_myshell(const int argc, char * const argv[], char * const env[]) const;
00100
00104         typedef sh_err_t (Executor::*MemFuncPtr)(const int argc, char * const argv[], char * const env[]) const;
00106         MemFuncPtr FunctionArray[FunctionNumber];
00107
00109         static bool test_file_state(const int argc, const char * const argv[]);
00111         static bool test_number_compare(const int argc, const char * const argv[]);
00113         static bool test_string_compare(const int argc, const char * const argv[]);
00114
00115     public:
00116         Executor(Console *model, Display *view);
00117
00118         virtual ~Executor();
00119
00132         sh_err_t execute(const int argc, char * const argv[], char * const env[]) const;
```

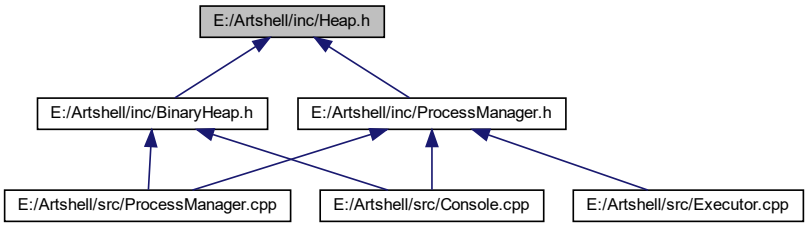
```
00133 };
00134
00135
00136 #endif
```

10.15 E:/Artshell/inc/Heap.h

#include <assert.h>
#include <stddef.h>
Heap.h (Include) :



:



- class Heap< T >

10.15.1

(3200105842@zju.edu.cn)

0.1

2022-07-20

Heap.h .

10.16 Heap.h

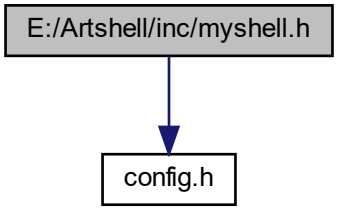
.

```
00001
00012 #ifndef __HEAP__H_
00013 #define __HEAP__H_
00014
00015 #include <assert.h>
00016 #include <stddef.h>
00017
00027 template <class T>
00028 class Heap
00029 {
00030     public:
00031         Heap() : size_(0) {};
00032
00043         virtual ~Heap() {};
00044
00045         size_t size() const { return size_; }
00046
00047         virtual void build(T data[], size_t size) = 0;
00048
00049         virtual void insert(T value)
00050         {
00051             assert(false && "insert not implemented.");
00052         }
00053
00054         virtual T top() const
00055         {
00056             assert(false && "top not implemented.");
00057             return 0;
00058         }
00059
00060         virtual T extract()
00061         {
00062             assert(false && "extract not implemented.");
00063             return 0;
00064         }
00065
00066     protected:
00067         size_t size_;      //
00068 };
00069
00070
00071 #endif
```

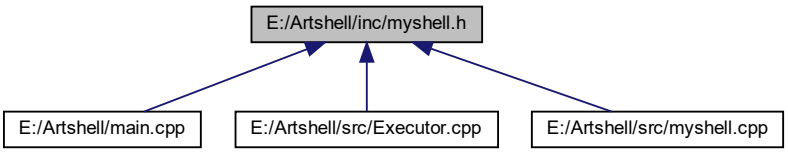
10.17 E:/Artshell/inc/myshell.h

myshell　　myshell.cpp

#include ”config.h”
myshell.h　(Include)　:



:



- namespace SHELL

- int SHELL::shell_setup (int argc, char ∗argv[], char ∗env[])
  shell
- int SHELL::shell_loop (Console ∗model, Display ∗view, Executor ∗controller, char ∗env[])
  shell

10.17.1

myshell myshell.cpp

(3200105842@zju.edu.cn)

0.1

2022-07-02

myshell.h .

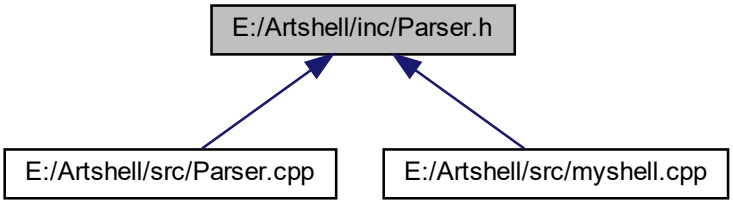
10.18 myshell.h

.

```
00001 //
00002 //      3200105842
00003
00017 #ifndef _MYSHELL_H_
00018 #define _MYSHELL_H_
00019
00020 /*     */
00021 #include "config.h"
00022
00023 /*     */
00024 class Console;
00025 class Display;
00026 class Executor;
00027
00028 namespace SHELL
00029 {
00031     int shell_setup(int argc, char *argv[], char *env[]);
00032
00034     int shell_loop(Console* model, Display* view, Executor* controller, char *env[]);
00035
00036 } // namespace SHELL
00037
00038
00039 #endif
```

10.19 E:/Artshell/inc/Parser.h

:



- class Parser

10.19.1

(3200105842@zju.edu.cn)

0.1

2022-07-19

Parser.h .

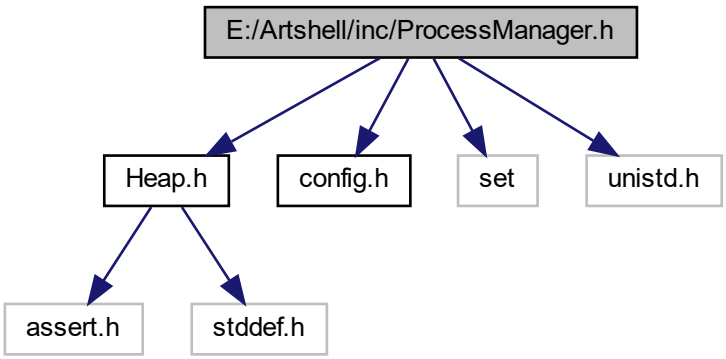
10.20 Parser.h

.

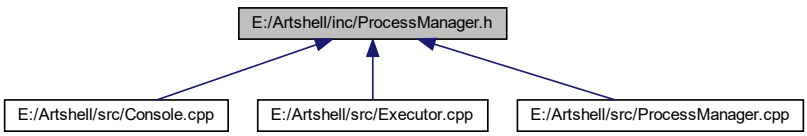
```
00001
00012 #ifndef __PARSER__H_
00013 #define __PARSER__H_
00014
00015 class Console;
00016 class Display;
00017 class Executor;
00018
00019 class Parser
00020 {
00021     private:
00022         enum {SUCCESS = 0, EXIT = 1};
00023
00040         static bool shell_execute(Console *model, Display* view, Executor* controller, int& argc, char *argv[], char *env[]);
00041
00042     public:
00043         Parser(/* args */) {};
00044
00053         virtual ~Parser() = 0;  //
00054
00055         static bool shell_pipe(Console *model, Display* view, Executor* controller, int& argc, char *argv[], char *env[]);
00056
00057         static int shell_parser(Console *model, Display* view, Executor* controller, int& argc, char *argv[], char *env[]);
00058 };
00059
00060 #endif
```

10.21 E:/Artshell/inc/ProcessManager.h

#include ”Heap.h”
#include ”config.h”
#include <set>
#include <unistd.h>
ProcessManager.h (Include) :



:



- class job_unit
- class ProcessManager

10.21.1

(3200105842@zju.edu.cn)

0.1

2022-08-10

ProcessManager.h .

10.22 ProcessManager.h

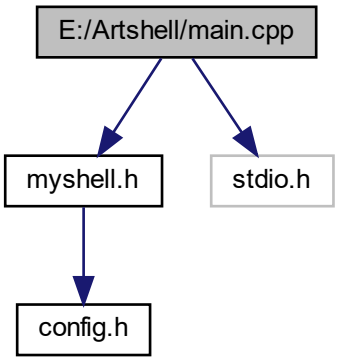
.

```
00001
00012 #ifndef __PROCESS__MANAGER__H__
00013 #define __PROCESS__MANAGER__H__
00014
00015 #include "Heap.h"
00016 #include "config.h"
00017
00018 #include <set>
00019 #include <unistd.h>
00020
00021 //  config.h
00022 // enum job_state                          //
00023 // {
00024 //     Running,                           //
00025 //     Stopped,                           //
00026 //     Done,                              //
00027 //     Terminated                          //
00028 // };
00029
00030 class job_unit
00031 {
00032     public:
00033         job_unit(unsigned int _id, int _pid, job_state _state, int _argc, char * _argv[]);
00034
00035         // ~job_unit();
00036
00037         void PrintJob(int output_fd = STDOUT_FILENO);
00038
00039         /*          job unit       */
00040         bool operator== ( const job_unit& rhs ) const
00041         {
00042             return id == rhs.id;
00043         }
00044
00045         bool operator!= ( const job_unit& rhs ) const
00046         {
00047             return !(*this == rhs);
00048         }
00049
00050         bool operator< ( const job_unit& rhs ) const
00051         {
00052             return id < rhs.id;
00053         }
00054
00055         bool operator> ( const job_unit& rhs ) const
00056         {
00057             return rhs < *this;
00058         }
00059
00060         bool operator<= ( const job_unit& rhs ) const
00061         {
00062             return !(rhs < *this);
00063         }
00064
00065         bool operator>= ( const job_unit& rhs ) const
00066         {
00067             return !(*this < rhs);
00068         }
00069
```

```
00070     // private:
00071         unsigned int id;                        //    id
00072         pid_t pid;                              //    pid
00073         job_state state;                        //
00074         int argc;                               //
00075         char argv[MAX_ARGUMENT_NUMBER][BUFFER_SIZE];//
00076 };
00077
00078 class ProcessManager
00079 {
00080     private:
00081         //
00082         Heap<unsigned int> *job_heap;                   //  id
00083         std::set<class job_unit> jobs;                  //     STL
00084
00085     public:
00086         ProcessManager(/* args */);
00087         virtual ~ProcessManager();
00088
00089         void PrintJobList(int output_fd = STDOUT_FILENO) const; //
00090
00091         void PrintJobListDone(int output_fd = STDOUT_FILENO); //
00092
00106         unsigned int JobInsert(int pid, job_state state, int argc, char *argv[]);
00107
00117         void JobRemove(job_unit * job);
00118         void JobRemove(std::set<job_unit>::iterator& job);
00119
00120         int ForeGround(unsigned int jobid);
00121         int BackGround(unsigned int jobid);
00122 };
00123
00124 #endif
```

10.23 E:/Artshell/main.cpp

#include ”myshell.h”
#include <stdio.h>
main.cpp (Include) :



- int main (int argc, char ∗argv[ ], char ∗∗env)

10.23.1

(3200105842@zju.edu.cn)

0.1

2022-07-17

main.cpp .

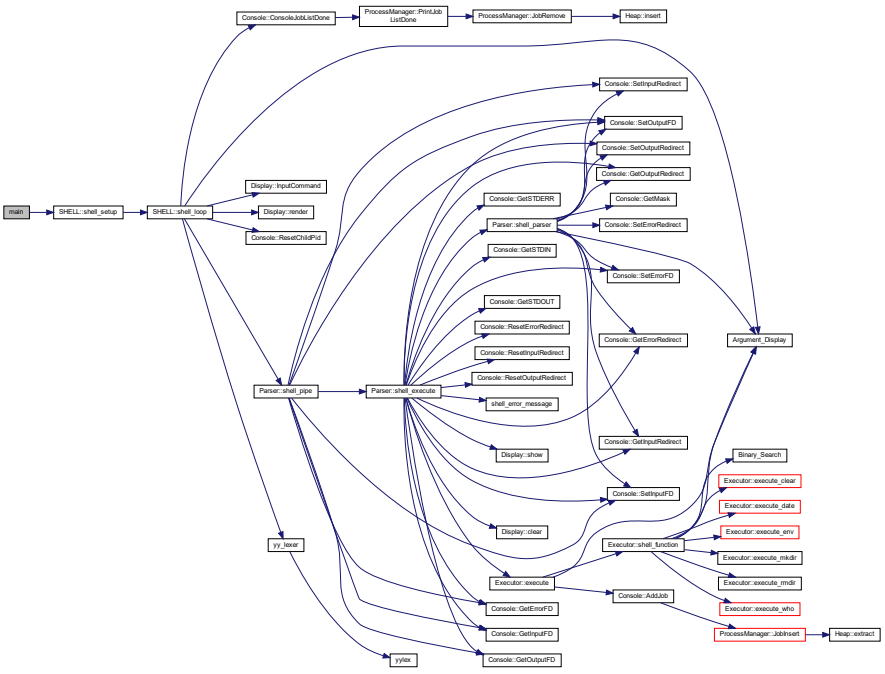
10.23.2

10.23.2.1 main()

int main (
int argc,
char ∗ argv[],
char ∗∗ env)

main.cpp 19 .

:



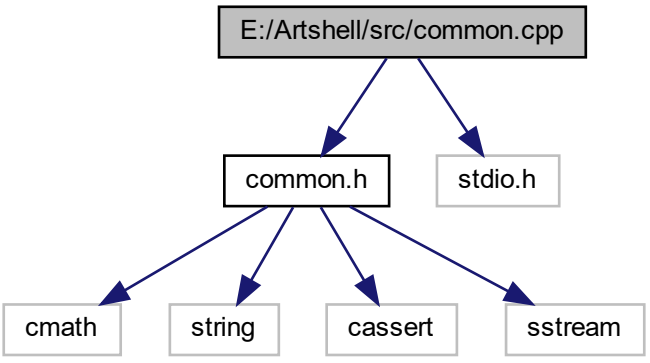
10.24 E:/Artshell/main.cpp

```
00001 //
00002 //      3200105842
00003
00015 #include "myshell.h"
00016
00017 #include <stdio.h>
00018
00019 int main(int argc, char *argv[], char **env)
00020 {
00021     //
00022     // puts("Welcome to MyShell ! \n");
00023
00024     if (SHELL::shell_setup(argc, argv, env) != 0)
00025         puts("shell setup failed.");
00026
00027     //
00028     // puts("Bye~");
00029
00030     return 0;
00031 }
```

10.25 E:/Artshell/README.md

10.26 E:/Artshell/src/common.cpp

#include "common.h"
#include <stdio.h>
common.cpp (Include) :



- void Argument_Display (const int argc, char *const argv[])
- std::string & String_Trim (std::string &s)

10.26.1

(3200105842@zju.edu.cn)

0.1

2022-07-15

common.cpp .

10.26.2

10.26.2.1 Argument_Display()

```
void Argument_Display (
                const int argc,
                char *const argv[] )
```

argc	
argv	

0.1

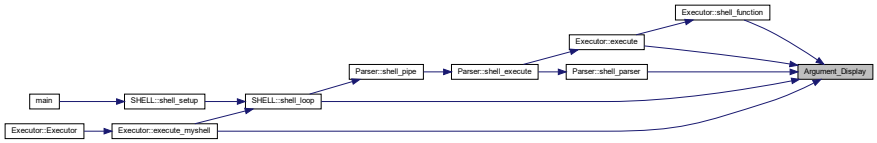
(3200105842@zju.edu.cn)

2022-07-15

Copyright (c) 2022

common.cpp 16 .

:



10.26.2.2 String_Trim()

std::string & String_Trim (
std::string & s)

s	

std::string&

0.1

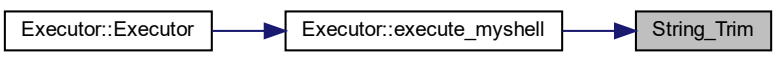
(3200105842@zju.edu.cn)

2022-07-17

Copyright (c) 2022

common.cpp 27 .

:

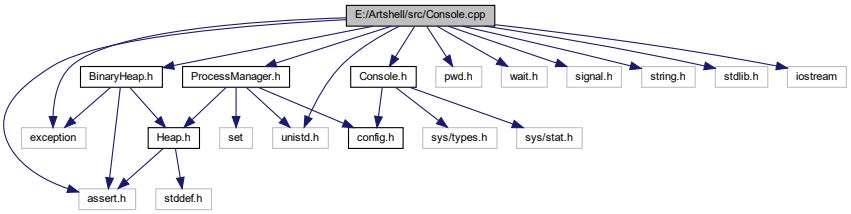


10.27 common.cpp

```
00001
00012 #include "common.h"
00013
00014 #include <stdio.h>
00015
00016 void Argument_Display(const int argc, char* const argv[])
00017 {
00018     printf("argc: %d\n", argc);
00019     for (int i = 0; i < argc; ++i)
00020     {
00021         printf("%s ", argv[i]);
00022     }
00023     putchar('\n');
00024     return;
00025 }
00026
00027 std::string& String_Trim(std::string &s)
00028 {
00029     if (s.empty()) //  s
00030     {
00031         return s;   //
00032     }
00033
00034     s.erase(0,s.find_first_not_of(" "));    //
00035     s.erase(s.find_last_not_of(" ") + 1);   //
00036     return s;
00037 }
```

10.28 E:/Artshell/src/Console.cpp

#include "Console.h"
#include "BinaryHeap.h"
#include "ProcessManager.h"
#include <pwd.h>
#include <wait.h>
#include <assert.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <iostream>
#include <exception>
Console.cpp (Include) :



- void SignalHandler (int signal_)

- static Console * cp = nullptr

10.28.1

(3200105842@zju.edu.cn)

0.1

2022-07-03

Console.cpp .

10.28.2

10.28.2.1 SignalHandler()

void SignalHandler (
int signal_)

signal↩ _	

0.1

(3200105842@zju.edu.cn)

2022-07-21

Copyright (c) 2022

Console.cpp 49 .

10.28.3

10.28.3.1 cp

Console* cp = nullptr [static]

Console.cpp 30 .

10.29 Console.cpp

.

```
00001
00012 #include "Console.h"
00013 #include "BinaryHeap.h"
00014 #include "ProcessManager.h"
00015
00016 #include <pwd.h>
00017 #include <wait.h>
00018 #include <assert.h>
00019 #include <signal.h>
00020 #include <string.h>
00021 #include <stdlib.h>
00022 #include <unistd.h>
00023 #include <iostream>
00024 #include <exception>
00025
00026 int Console::input_std_fd;
00027 int Console::output_std_fd;
00028 int Console::error_std_fd;
00029 pid_t Console::child_process_id = -1;
00030 static Console* cp = nullptr;   //
00031
00032 Console::Console(/* args */)
00033 {
00034     [[maybe_unused]] int ret;
00035     ret = init();       //
00036     assert(ret == 0);   //
00037
00038     process_manager = new ProcessManager();
00039     cp = this;
00040
00041     return;
00042 }
00043
00044 Console::~Console()
00045 {
00046     delete process_manager;
00047 }
00048
00049 void SignalHandler(int signal_)
00050 {
00051     switch (signal_)
00052     {
00053         case SIGINT:    // Ctrl C
00054             #ifdef _DEBUG_
00055             printf("Ctrl + C\n");
00056             #endif
00057             if (write(STDOUT_FILENO, "\n", 1) < 0)
00058                 throw std::exception();
00059             //  kill pid < 0    |pid|
00060             // kill(-getpid(), SIGINT);
00061
```

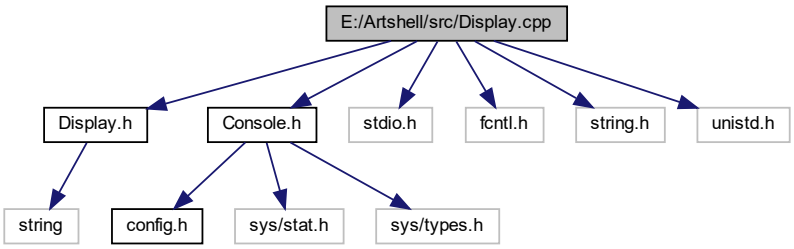
```
00062             //    CTRL C
00063             break;
00064
00065         case SIGTSTP:    // Ctrl Z
00066             #ifdef _DEBUG_
00067             printf("Ctrl + Z\n");
00068             #endif
00069             if (write(STDOUT_FILENO, "\n", 1) < 0)
00070                 throw std::exception();
00071
00072             if (Console::child_process_id >= 0)
00073             {
00074                 setpgid(Console::child_process_id, 0);
00075                 kill(-Console::child_process_id, SIGTSTP);
00076
00077                 unsigned int jobid = cp->AddJob(Console::child_process_id, Stopped, cp->argc, (char **)cp->argv);
00078
00079                 //
00080                 char buffer[BUFFER_SIZE];
00081                 snprintf(buffer, BUFFER_SIZE-1, "[%u] %d\n", jobid, Console::child_process_id);
00082                 if (write(cp->output_std_fd, buffer, strlen(buffer)) == -1)
00083                     throw std::exception();
00084
00085                 snprintf(buffer, BUFFER_SIZE-1, "[%u]%c\tStopped\t\t\t\t\t", jobid, ' ');
00086                 if (write(cp->output_std_fd, buffer, strlen(buffer)) == -1)
00087                     throw std::exception();
00088
00089                 //
00090                 if (cp->argc > 0)
00091                 {
00092                     //
00093                     if (write(cp->output_std_fd, cp->argv[0], strlen(cp->argv[0])) == -1)
00094                         throw std::exception();
00095                     for (int i = 1; i < cp->argc; ++i)
00096                     {
00097                         if (write(cp->output_std_fd, " ", 1) == -1)    //
00098                             throw std::exception();
00099
00100                         //
00101                         if (write(cp->output_std_fd, cp->argv[i], strlen(cp->argv[i])) == -1)
00102                             throw std::exception();
00103                     }
00104                 }
00105                 if (write(cp->output_std_fd, "\n", 1) == -1)
00106                     throw std::exception();
00107
00108                 Console::child_process_id = -1;
00109             }
00110             break;
00111
00112         case SIGCHLD:   //
00113             //
00114             // waitpid(-1, NULL, WNOHANG);
00115             break;
00116
00117         default:
00118             break;
00119     }
00120
00121 }
00122
00123 int Console::init()
00124 {
00125     try
00126     {
00127         //
00128         struct passwd *pw = getpwuid(getuid());
00129         if (pw == nullptr)
00130         {
00131             throw "get user database entry error";
00132         }
00133         memset(user_name, 0, BUFFER_SIZE);
00134         strncpy(user_name, pw->pw_name, BUFFER_SIZE-1);
00135
00136         //
00137         int ret;
00138         ret = gethostname(host_name, BUFFER_SIZE-1);
00139         if (ret != 0)
00140         {
00141             throw "Error when getting host name";
00142         }
00143
00144         //
00145         char *result;
00146         result = getcwd(current_working_dictionary, BUFFER_SIZE);
00147         if (result == NULL)
00148         {
```

```
00149             throw "Error when getting current working dictionary";
00150         }
00151 
00152         //
00153         memset(home, 0, BUFFER_SIZE);
00154         strncpy(home, getenv("HOME"), BUFFER_SIZE-1);
00155 
00156         //  shell
00157         strncpy(shell_path_env, current_working_dictionary, BUFFER_SIZE);
00158         strncat(shell_path_env, "/myshell", BUFFER_SIZE);
00159         setenv("shell", shell_path_env, 1);
00160 
00161         //
00162         umask_ = umask(022);  //
00163         umask(umask_); //
00164 
00165         //
00166         input_file_descriptor = STDIN_FILENO;
00167         output_file_descriptor = STDOUT_FILENO;
00168         error_file_descriptor = STDERR_FILENO;
00169 
00170         //
00171         redirect_input = false;
00172         redirect_output = false;
00173         redirect_error = false;
00174 
00175         //  STD IO
00176         input_std_fd = dup(STDIN_FILENO);
00177         output_std_fd = dup(STDOUT_FILENO);
00178         error_std_fd = dup(STDERR_FILENO);
00179 
00180         //
00181         process_id = getpid();
00182         child_process_id = -1;  //
00183 
00184         // signal(SIGINT, SignalHandler);  // Ctrl + C
00185         signal(SIGTSTP, SignalHandler); // Ctrl + Z
00186         signal(SIGCHLD, SignalHandler); //
00187 
00188         //  shell    tcsetpcgrp
00189         signal(SIGTTIN, SIG_IGN);   //  SIGTTIN
00190         signal(SIGTTOU, SIG_IGN);   //  SIGTTOU
00191     }
00192     catch(const std::exception& e)
00193     {
00194         std::cerr « e.what() « '\n';
00195         return 1;
00196     }
00197 
00198     return 0;
00199 }
00200 
00201 void Console::ConsoleJobList() const
00202 {
00203     /*           */
00204     process_manager->PrintJobList(STDOUT_FILENO);
00205 }
00206 
00207 void Console::ConsoleJobListDone()
00208 {
00209     /*            */
00210     process_manager->PrintJobListDone(output_std_fd);
00211 }
00212 
00213 unsigned int Console::AddJob(int pid, job_state state, int argc, char *argv[])
00214 {
00215     return process_manager->JobInsert(pid, state, argc, argv);
00216 }
```

10.30 E:/Artshell/src/Display.cpp

```
#include "Display.h"
#include "Console.h"
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
```

Display.cpp (Include) :



10.30.1

(3200105842@zju.edu.cn)

0.1

2022-07-03

Copyright (c) 2022

Display.cpp .

10.31 Display.cpp

.

```
00001
00012 #include "Display.h"
00013 #include "Console.h"
00014
00015 #include <stdio.h>
00016 #include <fcntl.h>
00017 #include <string.h>
00018 #include <unistd.h>
00019
00020 Display::Display(Console* console)
00021 : console_(console), perform(true), buffer_("")
00022 {
00023 }
00024
00025 Display::~Display()
00026 {
00027 }
00028
00029 int Display::InputCommand(char *input, const int len)
00030 {
00031     tcsetpgrp(STDIN_FILENO, getpid());
00032
00033     //
```

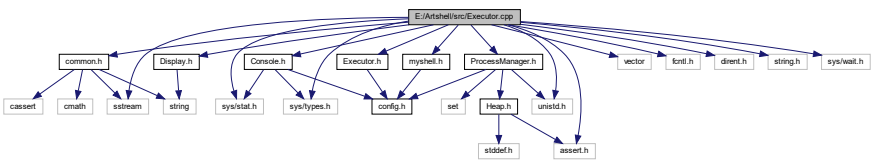
```
00034     char ch;
00035     int i = 0;
00036     memset(input, 0, len);
00037
00038     //
00039     do
00040     {
00041         ssize_t state = read(console_->input_file_descriptor, &ch, 1);
00042         if (state == 0)
00043         {
00044             //   EOF
00045             if (i == 0) //
00046                 return 0;   //
00047             else    //
00048             {
00049                 input[i++] = '\n';  //
00050                 return i;           //
00051             }
00052         }
00053         else if (state == -1)
00054         {
00055             throw "Read Input Error";
00056         }
00057
00058
00059         if (ch == '\\') //       \
00060         {
00061             ch = getchar(); //
00062             continue;
00063         }
00064
00065         if (ch == ';')  //        lexer parser
00066         {
00067             ch = '\n';
00068             perform = false;
00069         }
00070
00071         input[i++] = ch;
00072
00073         if (i == len)   //
00074         {
00075             buffer_ = "\e[1;31mERROR\e[0m input compand exceeds maximum length.              ";
00076             memset(input, 0, len);  //
00077             return -1;
00078         }
00079     } while (ch != '\n');
00080
00081     #ifdef _DEBUG_
00082     printf("input: %s", input);
00083     #endif
00084
00085     return i;
00086 }
00087
00088 void Display::render()
00089 {
00090     buffer_ = "";   //
00091
00092     //
00093     if (console_->input_file_descriptor != STDIN_FILENO ||
00094         console_->output_file_descriptor != STDOUT_FILENO)
00095         return; //
00096
00097     if (!perform)
00098     {
00099         perform = true;
00100         return;
00101     }
00102
00103     int sret = 0;
00104     const size_t len = strlen(console_->home);
00105     if (strlen(console_->current_working_dictionary) >= len)
00106     {
00107         size_t i = 0;
00108         while (i < len)
00109         {
00110             if (console_->current_working_dictionary[i] != console_->home[i])
00111                 break;
00112             ++i;
00113         }
00114         if (i == len)
00115             sret = i;
00116     }
00117
00118     char buffer[BUFFER_SIZE];   //
00119     sret = sret
00120         ? snprintf(buffer, BUFFER_SIZE, "\e[1;32m%s@%s\e[0m:\e[1;34m~%s\e[0m> ", \
```

```
00121         console_->user_name, console_->host_name, console_->current_working_dictionary+sret)
00122         : snprintf(buffer, BUFFER_SIZE, "\e[1;32m%s@%s\e[0m:\e[1;34m%s\e[0m> ", \
00123         console_->user_name, console_->host_name, console_->current_working_dictionary);
00124     if (sret == -1)
00125     {
00126         throw "Error when writing into output buffer";
00127     }
00128 
00129     ssize_t ret;
00130     ret = write(console_->output_file_descriptor, buffer, strlen(buffer));
00131     if (ret == -1)
00132     {
00133         throw "Error when writing from buffer";
00134     }
00135 
00136     return;
00137 }
00138 
00139 void Display::prompt() const
00140 {
00141     if (write(console_->output_file_descriptor, "> ", 2) == -1)
00142     {
00143         throw std::exception();
00144     }
00145 }
00146 
00147 void Display::message(const char * msg)
00148 {
00149     buffer_ += std::string(msg);
00150 }
00151 
00152 void Display::show() const
00153 {
00154     ssize_t ret;
00155     ret = write(console_->output_file_descriptor, buffer_.c_str(), buffer_.length());
00156     if (ret == -1)
00157     {
00158         throw "Error when showing buffer in Display";
00159     }
00160 }
```

10.32 E:/Artshell/src/Executor.cpp

```
#include "common.h"
#include "myshell.h"
#include "Console.h"
#include "Display.h"
#include "Executor.h"
#include "ProcessManager.h"
#include <vector>
#include <sstream>
#include <fcntl.h>
#include <assert.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/types.h>
```

Executor.cpp (Include) :



- static bool test_tty (const char *file_name)

- static const char * OperandArray []

10.32.1

(3200105842@zju.edu.cn)

0.1

2022-07-04

Copyright (c) 2022

Executor.cpp .

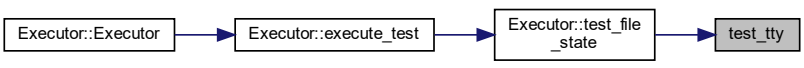
10.32.2

10.32.2.1 test_tty()

static bool test_tty (
const char * file_name) [inline], [static]

Executor.cpp 841 .

:



10.32.3

10.32.3.1 OperandArray

const char* OperandArray[] [static]

:
=
{
"bg", "cd", "clr", "dir", "echo", "exec", "exit", "fg",
"help", "jobs", "myshell", "pwd", "set", "test", "time", "umask"
}

Executor.cpp 36 .

10.33 Executor.cpp

.

```
00001
00012 #include "common.h"
00013 #include "myshell.h"
00014 #include "Console.h"
00015 #include "Display.h"
00016 #include "Executor.h"
00017 #include "ProcessManager.h"
00018
00019 #include <vector>
00020 #include <sstream>
00021
00022 #include <fcntl.h>
00023 #include <assert.h>
00024 #include <dirent.h>
00025 #include <string.h>
00026 #include <unistd.h>
00027
00028 #include <sys/stat.h>
00029 #include <sys/wait.h>
00030 #include <sys/types.h>
00031
00033 static inline bool test_tty(const char * file_name);
00034
00036 static const char* OperandArray[] =
00037 {
00038     "bg", "cd", "clr", "dir", "echo", "exec", "exit", "fg",
00039     "help", "jobs", "myshell", "pwd", "set", "test", "time", "umask"
00040 };
00041
00042 Executor::Executor(Console *model, Display *view)
00043 : console_(model), display_(view)
00044 {
00045     assert(console_ != nullptr);
00046     assert(display_ != nullptr);
00047
00049     int i = 0;
00050
00051     FunctionArray[i] = &Executor::execute_bg;       ++i;
00052     FunctionArray[i] = &Executor::execute_cd;       ++i;
00053     FunctionArray[i] = &Executor::execute_clr;      ++i;
00054     FunctionArray[i] = &Executor::execute_dir;      ++i;
00055     FunctionArray[i] = &Executor::execute_echo;     ++i;
00056     FunctionArray[i] = &Executor::execute_exec;     ++i;
00057     FunctionArray[i] = &Executor::execute_exit;     ++i;
00058     FunctionArray[i] = &Executor::execute_fg;       ++i;
00059
00060     FunctionArray[i] = &Executor::execute_help;     ++i;
00061     FunctionArray[i] = &Executor::execute_jobs;     ++i;
00062     FunctionArray[i] = &Executor::execute_myshell;  ++i;
00063     FunctionArray[i] = &Executor::execute_pwd;      ++i;
```

```
00064     FunctionArray[i] = &Executor::execute_set;      ++i;
00065     FunctionArray[i] = &Executor::execute_test;     ++i;
00066     FunctionArray[i] = &Executor::execute_time;     ++i;
00067     FunctionArray[i] = &Executor::execute_umask;    ++i;
00068
00069     return;
00070 }
00071
00072 Executor::~Executor()
00073 {
00074 }
00075
00076 sh_err_t Executor::execute(const int argc, char * const argv[], char * const env[]) const
00077 {
00078     if (argc == 0)
00079         return SH_SUCCESS;  //
00080     else if (argv == nullptr || argv[0] == nullptr)
00081     {
00082         assert(false);
00083         return SH_FAILED;   //
00084     }
00085
00086     /*      */
00087     int& argc_ = const_cast<int&>(argc);
00088     if (strcmp(argv[argc - 1], "&") == 0)    //
00089     {
00090         --argc_;
00091         if (argc == 0)  //
00092             return SH_ARGS;
00093
00094         pid_t pid;
00095         if ((pid = fork()) < 0)
00096         {
00097             /*    */
00098             throw "Fork Error,    ";
00099         }
00100         else if (pid == 0)
00101         {
00102             /*    */
00103             setenv("parent", console_->shell_path_env, 1);  //
00104             Console::child_process_id = getpid();
00105
00106             #ifdef _DEBUG_
00107             printf("child pid: %d\n", console_->process_id);
00108             #endif
00109
00110             setpgid(0, 0);
00111             signal(SIGINT, SIG_DFL);    //  Ctrl C
00112             signal(SIGTSTP, SIG_DFL);   //  Ctrl Z
00113
00114             char **&argv_ = const_cast<char **&>(argv);
00115             argv_[argc] = NULL;
00116             #ifdef _DEBUG_
00117             Argument_Display(argc, argv);
00118             #endif
00119
00120             //
00121             shell_function(argc, argv, env);
00122
00123             //
00124             return SH_EXIT;
00125         }
00126         else
00127         {
00128             /*    */
00129             #ifdef _DEBUG_
00130             printf("parent pid: %d\n", pid);
00131             #endif
00132
00133             //
00134             char **&argv_ = const_cast<char **&>(argv);
00135             argv_[argc] = NULL;
00136             unsigned int jobid = console_->AddJob(pid, Running, argc_, argv_);
00137             // console_->process_id = getpid();    //       pid
00138             console_->child_process_id = pid;
00139
00140             //
00141             char buffer[32];
00142             snprintf(buffer, 32, "[%u] %d\n", jobid, pid);
00143             if (write(console_->output_std_fd, buffer, strlen(buffer)) == -1)
00144                 throw std::exception();
00145
00146             // setpgid(pid, pid);
00147
00148             // //
00149             // tcsetpgrp(STDIN_FILENO, pid);
00150             // tcsetpgrp(STDOUT_FILENO, pid);
```

```
00151             // tcsetpgrp(STDERR_FILENO, pid);
00152
00153             // int status;
00154             // waitpid(pid, &status, WNOHANG);
00155
00156             // //
00157             // tcsetpgrp(STDIN_FILENO, getpid());
00158             // tcsetpgrp(STDOUT_FILENO, getpid());
00159             // tcsetpgrp(STDERR_FILENO, getgid());
00160
00161             return SH_SUCCESS;
00162         }
00163     }
00164
00165     return shell_function(argc, argv, env);
00166 }
00167
00168 sh_err_t Executor::shell_function(const int argc, char * const argv[], char * const env[]) const
00169 {
00170     const char *op = argv[0];
00171     console_->argc = argc;
00172     for (int i = 0; i < argc; ++i)
00173         strncpy(console_->argv[i], argv[i], BUFFER_SIZE);
00174
00175 #ifdef _DEBUG_
00176     Argument_Display(argc, argv);
00177
00178     //
00179     if (strcmp(op, "date") == 0)
00180     {
00181         return execute_date(argc, argv, env);
00182     }
00183     else if (strcmp(op, "clear") == 0)
00184     {
00185         return execute_clear(argc, argv, env);
00186     }
00187     else if (strcmp(op, "env") == 0)
00188     {
00189         return execute_env(argc, argv, env);
00190     }
00191     else if (strcmp(op, "who") == 0)
00192     {
00193         return execute_who(argc, argv, env);
00194     }
00195     else if (strcmp(op, "mkdir") == 0)
00196     {
00197         return execute_mkdir(argc, argv, env);
00198     }
00199     else if (strcmp(op, "rmdir") == 0)
00200     {
00201         return execute_rmdir(argc, argv, env);
00202     }
00203 #endif
00204
00206     int index = Binary_Search(0, sizeof(OperandArray)/sizeof(OperandArray[0]), op, OperandArray, strcmp);
00207 #ifdef _DEBUG_
00208     printf("index: %d op: %s\n", index, OperandArray[index>=0?index:0]);
00209 #endif
00210
00211     if (index >= 0 && index < FunctionNumber)   //
00212     {
00213         MemFuncPtr FunctionPointer = FunctionArray[index];  //
00214         return (*this.*FunctionPointer)(argc, argv, env);   //
00215     }
00216
00217
00218     //
00219     // shell
00220     pid_t pid = getpid(); //      id
00221     if ((pid = vfork()) < 0)
00222     {
00223         /*     */
00224         throw "Fork Error,    ";
00225     }
00226     else if (pid == 0)
00227     {
00228         /*    */
00229         setenv("parent", console_->shell_path_env, 1);  //
00230         int status_code = execvp(argv[0], argv);        //
00231
00232         if (status_code == -1)
00233         {
00234             throw "Execvp Error, terminated incorrectly";
00235         }
00236
00237         return SH_UNDEFINED; //
00238     }
```

```
00239     else
00240     {
00241         /*    */
00242         console_->child_process_id = pid;   //     pid   Ctrl+Z
00243         wait(NULL); //
00244         console_->child_process_id = -1;
00245         return SH_SUCCESS;
00246     }
00247
00248     return SH_FAILED;
00249 }
00250
00251 sh_err_t Executor::execute_cd(const int argc, char * const argv[], char * const env[]) const
00252 {
00253     assert(strcmp(argv[0], "cd")==0 && "unexpected node type");
00254
00255     std::string path;
00256     if (argc == 1)
00257     {
00258         //
00259         path = console_->home;
00260     }
00261     else if (argc == 2)
00262     {
00263         path = argv[1];
00264
00265         #ifdef _DEBUG_
00266         printf("char: %c %d\n", path[0], (path[0] == '~'));
00267         #endif
00268
00269         if (path[0] == '~') //   ~
00270         {
00271             // ~
00272             path.replace(0, 1, console_->home);
00273         }
00274
00275         #ifdef _DEBUG_
00276         printf("Argv: %s\nHome: %s\nPath: %s\n", argv[1], console_->home, path.c_str());
00277         #endif
00278     }
00279     else
00280     {
00281         return SH_ARGS; //
00282     }
00283
00284     //
00285     int ret = chdir(path.c_str());
00286     if (ret != 0)   //
00287     {
00288         throw ((std::string)"cd:       " + path);
00289     }
00290
00291     //
00292     if (getcwd(console_->current_working_dictionary, BUFFER_SIZE) != nullptr )
00293         setenv("PWD", console_->current_working_dictionary, 1);
00294     else
00295         throw "get cwd error";
00296
00297     return SH_SUCCESS;
00298 }
00299
00300 sh_err_t Executor::execute_pwd(const int argc, char * const argv[], char * const env[]) const
00301 {
00302     assert(strcmp(argv[0], "pwd")==0 && "unexpected node type");
00303     display_->message(console_->current_working_dictionary);
00304     display_->message("\n");
00305     return SH_SUCCESS;
00306 }
00307
00308 sh_err_t Executor::execute_time(const int argc, char * const argv[], char * const env[]) const
00309 {
00310     assert(strcmp(argv[0], "time")==0 && "unexpected node type");
00311
00312     //   env             env    Linux
00313     return execute_date(argc, argv, env);
00314 }
00315
00316 sh_err_t Executor::execute_clr(const int argc, char * const argv[], char * const env[]) const
00317 {
00318     assert(strcmp(argv[0], "clr")==0 && "unexpected node type");
00319
00320     //   clear             env    Linux
00321     return execute_clear(argc, argv, env);
00322 }
00323
00324 sh_err_t Executor::execute_dir(const int argc, char * const argv[], char * const env[]) const
00325 {
```

```
00326     assert(strcmp(argv[0], "dir")==0 && "unexpected node type");
00327
00328     std::string real_path;
00329     if (argc == 1)
00330     {
00331         //
00332         real_path = console_->current_working_dictionary;
00333     }
00334     else if (argc == 2)
00335     {
00336         real_path = argv[1];
00337         if (real_path[0] == '~') //  ~
00338         {
00339             // ~
00340             real_path.replace(0, 1, console_->home);
00341         }
00342     }
00343     else
00344     {
00345         return SH_ARGS; //
00346     }
00347
00348     int ret;                //
00349     DIR *direction_pointer; //
00350     if ((direction_pointer = opendir(real_path.c_str())) == NULL)
00351     {
00352         throw ((std::string)"dir:       " + real_path);
00353     }
00354
00355     //
00356     ret = chdir(real_path.c_str());
00357     if (ret != 0)   //
00358     {
00359         throw ((std::string)"dir:       " + real_path);
00360     }
00361
00362     struct dirent *entry;   //
00363     while ((entry = readdir(direction_pointer)) != NULL)
00364     {
00365         struct stat stat_buffer;    //  stat
00366         lstat(entry->d_name, &stat_buffer); //       stat
00367
00368         char buffer[BUFFER_SIZE];
00369         if (S_ISDIR(stat_buffer.st_mode))   //
00370         {
00371             //
00372
00373             if (strcmp(".", entry->d_name) == 0 ||
00374                 strcmp("..", entry->d_name) == 0)
00375             {
00376                 //   . ..
00377                 continue;
00378             }
00379
00380             //
00381             snprintf(buffer, BUFFER_SIZE, "\033[34m%s\033[0m  ", entry->d_name);
00382             if (console_->redirect_output == false)
00383                 display_->message(buffer);
00384             else
00385             {
00386                 display_->message(entry->d_name);
00387                 display_->message("  ");
00388             }
00389         }
00390         else
00391         {
00392             //
00393             switch (entry->d_type)
00394             {
00395                 case DT_UNKNOWN:    //
00396                     snprintf(buffer, BUFFER_SIZE, "\033[31m%s\033[0m  ", entry->d_name);
00397                     break;
00398
00399                 case DT_REG:        //
00400                     if (access(entry->d_name, X_OK) == 0)    //
00401                         snprintf(buffer, BUFFER_SIZE, "\033[32m%s\033[0m  ", entry->d_name);
00402                     else
00403                         snprintf(buffer, BUFFER_SIZE, "\033[37m%s\033[0m  ", entry->d_name);
00404                     break;
00405
00406                 default:            //
00407                     snprintf(buffer, BUFFER_SIZE, "\033[36m%s\033[0m  ", entry->d_name);
00408                     break;
00409             }
00410             if (console_->redirect_output == false)
00411                 display_->message(buffer);
00412             else
```

```
00413             {
00414                 display_->message(entry->d_name);
00415                 display_->message("  ");
00416             }
00417         }
00418     }
00419     display_->message("\n");
00420 
00421     //
00422     ret = chdir(console_->current_working_dictionary);
00423     if (ret != 0)   //
00424     {
00425         throw ((std::string)"dir:       " + real_path);
00426     }
00427 
00428     ret = closedir(direction_pointer);
00429     if (ret == -1)  //
00430     {
00431         throw "dir:       ";
00432     }
00433 
00434     return SH_SUCCESS;
00435 }
00436 
00437 sh_err_t Executor::execute_set(const int argc, char * const argv[], char * const env[]) const
00438 {
00439     assert(strcmp(argv[0], "set")==0 && "unexpected node type");
00440 
00441     //   env              env    Linux
00442     return execute_env(argc, argv, env);
00443 }
00444 
00445 sh_err_t Executor::execute_echo(const int argc, char * const argv[], char * const env[]) const
00446 {
00447     assert(strcmp(argv[0], "echo")==0 && "unexpected node type");
00448 
00449     for (int i = 1; i < argc; ++i)
00450     {
00451         //
00452         if (i > 1)
00453             display_->message(" ");
00454 
00455         display_->message(argv[i]);
00456     }
00457     display_->message("\n");
00458 
00459     return SH_SUCCESS;
00460 }
00461 
00462 sh_err_t Executor::execute_help(const int argc, char * const argv[], char * const env[]) const
00463 {
00464     assert(strcmp(argv[0], "help")==0 && "unexpected node type");
00465 
00466     FILE* fp = fopen("README.md", "r");
00467     if (fp == nullptr)
00468     {
00469         return SH_FAILED;
00470     }
00471 
00472     char buffer[BUFFER_SIZE*2];
00473     if (fgets(buffer, BUFFER_SIZE, fp) == nullptr)
00474         return SH_FAILED;    //
00475 
00476     size_t size = fread(buffer, 1, BUFFER_SIZE*2, fp);
00477     if (size < 0)
00478     {
00479         return SH_FAILED;
00480     }
00481 
00482     if (fclose(fp) == -1)
00483     {
00484         throw std::exception();
00485     }
00486 
00487     display_->message(buffer);
00488     display_->message("\n");
00489 
00490     return SH_SUCCESS;
00491 }
00492 
00493 sh_err_t Executor::execute_exit(const int argc, char * const argv[], char * const env[]) const
00494 {
00495     assert(strcmp(argv[0], "exit")==0 && "unexpected node type");
00496     return SH_EXIT;
00497 }
00498 
00499 sh_err_t Executor::execute_date(const int argc, char * const argv[], char * const env[]) const
```

```
00500 {
00501     // assert(strcmp(argv[0], "date")==0 && "unexpected node type");
00502 
00503     //
00504     time_t t = time(NULL);
00505     struct tm *ptr = localtime(&t);
00506 
00507     //
00508     // char weekday[16], month[16];
00509     char date[256];
00510     // strftime(weekday, 16, "%A", ptr);
00511     // strftime(month, 16, "%B", ptr);
00512     strftime(date, 256, "%c", ptr);
00513 
00514     // char buffer[BUFFER_SIZE];
00515     // snprintf(buffer, BUFFER_SIZE, "%s %s %s\n", weekday, month, date);
00516 
00517     // display_->message(buffer);
00518     display_->message(date);
00519     display_->message("\n");
00520 
00521     return SH_SUCCESS;
00522 }
00523 
00524 sh_err_t Executor::execute_clear(const int argc, char * const argv[], char * const env[]) const
00525 {
00526     display_->message("\x1b[H\x1b[2J");    //       \x1b[H\x1b[2J
00527 
00528     return SH_SUCCESS;
00529 }
00530 
00531 sh_err_t Executor::execute_env(const int argc, char * const argv[], char * const env[]) const
00532 {
00533     extern char **environ; //env variables
00534     char ***update_env = const_cast<char ***>(&env);
00535     *update_env = environ;
00536 
00537     while(*env)
00538     {
00539         char buffer[BUFFER_SIZE];
00540         snprintf(buffer, BUFFER_SIZE, "%s\n", *env++);
00541         display_->message(buffer);
00542     }
00543 
00544     return SH_SUCCESS;
00545 }
00546 
00547 sh_err_t Executor::execute_who(const int argc, char * const argv[], char * const env[]) const
00548 {
00549     assert(strcmp(argv[0], "who")==0 && "unexpected node type");
00550     display_->message(console_->user_name);
00551     display_->message("\n");
00552     return SH_SUCCESS;
00553 }
00554 
00555 sh_err_t Executor::execute_mkdir(const int argc, char * const argv[], char * const env[]) const
00556 {
00557     assert(strcmp(argv[0], "mkdir")==0 && "unexpected node type");
00558 
00559     const char * path = argv[1];
00560     if (mkdir(path, S_IRWXU) == 0)
00561     {
00562         return SH_SUCCESS;
00563     }
00564     else
00565     {
00566         return SH_FAILED;
00567     }
00568 
00569 }
00570 
00571 sh_err_t Executor::execute_rmdir(const int argc, char * const argv[], char * const env[]) const
00572 {
00573     assert(strcmp(argv[0], "rmdir")==0 && "unexpected node type");
00574 
00575     if (rmdir(argv[1]) == 0)
00576     {
00577         return SH_SUCCESS;
00578     }
00579     else
00580     {
00581         return SH_FAILED;
00582     }
00583 }
00584 
00585 sh_err_t Executor::execute_bg(const int argc, char * const argv[], char * const env[]) const
00586 {
```

```
00587     assert(strcmp(argv[0], "bg")==0 && "unexpected node type");
00588
00589     if (argc == 1)
00590         return SH_SUCCESS;
00591
00592     unsigned int job_id = String_to_Number<unsigned int>(argv[1]);
00593     int id = console_->process_manager->BackGround(job_id);
00594     if (id == 0)
00595     {
00596         char buffer[BUFFER_SIZE];
00597         snprintf(buffer, BUFFER_SIZE, "bg: job %u already in background\n", job_id);
00598         display_->message(buffer);
00599     }
00600
00601     if (id == -1)
00602     {
00603         char buffer[BUFFER_SIZE];
00604         snprintf(buffer, BUFFER_SIZE, "bg: %u : no such job\n", job_id);
00605         display_->message(buffer);
00606     }
00607
00608     return SH_SUCCESS;
00609 }
00610
00611 sh_err_t Executor::execute_fg(const int argc, char * const argv[], char * const env[]) const
00612 {
00613     assert(strcmp(argv[0], "fg")==0 && "unexpected node type");
00614
00615     if (argc == 1)
00616         return SH_SUCCESS;
00617
00618     unsigned int job_id = String_to_Number<unsigned int>(argv[1]);
00619     int id = console_->process_manager->ForeGround(job_id);
00620     if (id == -1)
00621     {
00622         char buffer[BUFFER_SIZE];
00623         snprintf(buffer, BUFFER_SIZE, "bg: %u : no such job\n", job_id);
00624         display_->message(buffer);
00625     }
00626
00627     return SH_SUCCESS;
00628 }
00629
00630 sh_err_t Executor::execute_jobs(const int argc, char * const argv[], char * const env[]) const
00631 {
00632     assert(strcmp(argv[0], "jobs")==0 && "unexpected node type");
00633
00634     console_->ConsoleJobList();
00635
00636     return SH_SUCCESS;
00637 }
00638
00639 sh_err_t Executor::execute_exec(const int argc, char * const argv[], char * const env[]) const
00640 {
00641     assert(strcmp(argv[0], "exec")==0 && "unexpected node type");
00642
00643     //
00644     if (argc == 1)
00645         return SH_SUCCESS;
00646
00647     int status_code = execvp(argv[1], argv+1);        //
00648
00649     if (status_code == -1)
00650     {
00651         throw "Execvp Error, terminated incorrectly";
00652     }
00653
00654     return SH_UNDEFINED; //
00655 }
00656
00657 sh_err_t Executor::execute_test(const int argc, char * const argv[], char * const env[]) const
00658 {
00659     assert(strcmp(argv[0], "test")==0 && "unexpected node type");
00660
00661     bool ret = false;
00662     if (argc == 1)  //     false
00663     {
00664         ret = false;
00665     }
00666     else if (argc == 2) //
00667     {
00668         if (strcmp(argv[1], "!") == 0 || strcmp(argv[1], "-z") == 0)
00669             ret = true;
00670         else
00671             ret = false;
00672     }
00673     else
```

```
00674     {
00675         if (strcmp(argv[1], "!"))   //
00676         {
00677             if (argc == 3 || argc == 4) //
00678             {
00679                 //
00680                 ret = Executor::test_file_state(argc, argv)
00681                     | Executor::test_number_compare(argc, argv)
00682                     | Executor::test_string_compare(argc, argv);
00683             }
00684             else
00685             {
00686                 return SH_ARGS;
00687             }
00688         }
00689     }
00690
00691     if (console_->GetOutputRedirect() == false) //
00692     {
00693         if (ret)
00694             display_->message("true\n");
00695         else
00696             display_->message("false\n");
00697     }
00698
00699     return SH_SUCCESS;
00700 }
00701
00702 sh_err_t Executor::execute_umask(const int argc, char * const argv[], char * const env[]) const
00703 {
00704     assert(strcmp(argv[0], "umask")==0 && "unexpected node type");
00705
00706     if (argc == 1)
00707     {
00708         //
00709         char buffer[16];
00710         snprintf(buffer, 16, "%04o\n", console_->umask_);   //      0
00711         display_->message(buffer);
00712     }
00713     else if (argc == 2) //
00714     {
00715         //
00716         console_->umask_ = String_to_Number<mode_t>(argv[1]);
00717
00718         if (argv[1][0] == '0')
00719         {
00720             if (strlen(argv[1]) >= 2 && argv[1][1] == 'x')  //
00721                 console_->umask_ = Hexadecimal_to_Decimal(console_->umask_);
00722             else                                    //
00723                 console_->umask_ = Octal_to_Decimal(console_->umask_);
00724         }
00725
00726         #ifdef _DEBUG_
00727         printf("mask: %04u %04o\n", console_->umask_, console_->umask_);
00728         #endif
00729         umask(console_->umask_);
00730     }
00731     else
00732     {
00733         return SH_ARGS; //
00734     }
00735
00736     return SH_SUCCESS;
00737 }
00738
00739 sh_err_t Executor::execute_myshell(const int argc, char * const argv[], char * const env[]) const
00740 {
00741     assert(strcmp(argv[0], "myshell")==0 && "unexpected node type");
00742
00743     std::vector<std::string> FileList;
00744     if (argc == 1)
00745     {
00746         /* shell
00747                 */
00748         while (1)   //
00749         {
00750             display_->prompt();
00751
00752             int len;
00753             char input[BUFFER_SIZE];
00754             len = display_->InputCommand(input, BUFFER_SIZE);
00755
00756             if (len == 1 || len < 0)
00757                 continue;   //
00758             if (len == 0)
00759                 return SH_EXIT; // EOF
00760
```

```
00761           #ifdef __DEBUG__
00762           printf("len: %d\n", len);
00763           #endif
00764           input[len-1] = '\0'; //     \n
00765
00766           int& argc_ = const_cast<int&>(argc);        //
00767           // char **argv_ = const_cast<char **>(argv);   //
00768
00769           std::istringstream line(input); //
00770           std::string word;               //
00771
00772           while (std::getline(line, word, ' '))
00773           {
00774               word = String_Trim(word);   //
00775               if (word == "")
00776                   continue;
00777
00778               ++argc_;
00779               FileList.emplace_back(word);  //
00780           }
00781
00782           if (argc == 1)
00783               continue;   //
00784
00785           #ifdef __DEBUG__
00786           Argument_Display(argc, argv);   //
00787           #endif
00788
00789           break;
00790       }
00791   }
00792   else
00793   {
00794       for (int i = 1; i < argc; ++i) //
00795           FileList.push_back(argv[i]);    //
00796   }
00797
00798   assert(argc > 1);   //
00799
00800   int input_fd = console_->input_file_descriptor; //        fd
00801   for (std::string File : FileList)
00802   {
00803       try
00804       {
00805           int fd = open(File.c_str(), O_RDONLY);   //
00806           if (fd < 0) //
00807           {
00808               throw std::exception();
00809           }
00810
00811           #ifdef __DEBUG__
00812           fprintf(stdout, "FD: %d Input: %d Output: %d\n", fd, console_->input_file_descriptor, console_-
    >output_file_descriptor);
00813           #endif
00814
00815           console_->input_file_descriptor = fd;   //
00816
00817           //
00818           SHELL::shell_loop(console_, display_, const_cast<Executor *>(this), const_cast<char **>(env));
00819
00820           int state_code = close(fd); //
00821           if (state_code != 0)    //
00822           {
00823               throw std::exception();
00824           }
00825       }
00826       catch(...)
00827       {
00828           puts("every thing");
00829           std::string msg = "\e[1;31m[ERROR]\e[0m";
00830           msg = msg + "myshell" + ": (" + File + ") " + strerror(errno) + "\n";
00831           display_->message(msg.c_str());
00832       }
00833
00834   }
00835
00836   console_->input_file_descriptor = input_fd; //     input fd
00837
00838   return SH_SUCCESS;
00839 }
00840
00841 static inline bool test_tty(const char * file_name)
00842 {
00843   try
00844   {
00845       int fd = open(file_name, S_IREAD);  //
00846       bool tty = isatty(fd);              //
```

```
00847         close(fd);                          //
00848         return tty;
00849     }
00850     catch(...)
00851     {
00852         return false;                       //        false
00853     }
00854 }
00855
00856 bool Executor::test_file_state(const int argc, const char * const argv[])
00857 {
00858     assert(argc == 3 || argc == 4);
00859
00860     if (argc == 3)
00861     {
00862         struct stat file_stat;
00863
00864         if (lstat(argv[2], &file_stat) < 0) //
00865         {
00866             return false;   //      false
00867         }
00868
00869         //
00870         switch (String_Hash(argv[1]))   //         switch
00871         {
00872             /*     */
00873             case String_Hash("-e"):  //
00874                 return true;
00875
00876             /*      */
00877             case String_Hash("-f"):  //
00878                 return S_ISREG(file_stat.st_mode);
00879
00880             case String_Hash("-d"):  //
00881                 return S_ISDIR(file_stat.st_mode);
00882
00883             case String_Hash("-c"):  //
00884                 return S_ISCHR(file_stat.st_mode);
00885
00886             case String_Hash("-b"):  //
00887                 return S_ISBLK(file_stat.st_mode);
00888
00889             case String_Hash("-p"):  //
00890                 return S_ISFIFO(file_stat.st_mode);
00891
00892             case String_Hash("-L"):  //
00893                 return S_ISLNK(file_stat.st_mode);
00894
00895             case String_Hash("-S"):  //
00896                 return S_ISSOCK(file_stat.st_mode);
00897
00898             /*      */
00899             case String_Hash("-r"):  //
00900                 return access(argv[1], R_OK);
00901
00902             case String_Hash("-w"):  //
00903                 return access(argv[1], W_OK);
00904
00905             case String_Hash("-x"):  //
00906                 return access(argv[1], X_OK);
00907
00908             case String_Hash("-O"):  //
00909                 return file_stat.st_uid == getuid();
00910
00911             case String_Hash("-G"):  //
00912                 return file_stat.st_gid == getgid();
00913
00914             /*      */
00915             case String_Hash("-u"):  //     SUID
00916                 return S_ISUID & file_stat.st_mode;
00917
00918             case String_Hash("-g"):  //    GUID
00919                 return S_ISGID & file_stat.st_mode;
00920
00921             case String_Hash("-k"):  // Sticky bit
00922                 return S_ISVTX & file_stat.st_mode;
00923
00924             case String_Hash("-s"):  //     0
00925                 return file_stat.st_size > 0;
00926
00927             case String_Hash("-t"):  //
00928                 return test_tty(argv[2]);
00929
00930             /*       */
00931             default:
00932                 return false;
00933         }
```

```
00934     }
00935     else
00936     {
00937         struct stat file_stat1, file_stat2;
00938 
00939         if (lstat(argv[1], &file_stat1) < 0) //
00940         {
00941             return false;   //      false
00942         }
00943         if (lstat(argv[3], &file_stat2) < 0) //
00944         {
00945             return false;   //      false
00946         }
00947 
00948         //
00949         switch (String_Hash(argv[2]))   //          switch
00950         {
00951             case String_Hash("-nt"):    //  file1   file2
00952                 return test_timespec_newer(file_stat1.st_mtim, file_stat2.st_mtim);
00953 
00954             case String_Hash("-ot"):    //  file1   file2
00955                 return test_timespec_older(file_stat1.st_mtim, file_stat2.st_mtim);
00956 
00957             case String_Hash("-ef"):    //  file1 file2
00958                 return file_stat1.st_ino == file_stat2.st_ino;
00959 
00960             default:                    /*      */
00961                 return false;
00962         }
00963     }
00964 }
00965 
00966 bool Executor::test_number_compare(const int argc, const char * const argv[])
00967 {
00968     if (argc != 4)
00969         return false;
00970 
00971 
00972     int number1 = String_to_Number<int>(argv[1]);
00973     int number2 = String_to_Number<int>(argv[2]);
00974 
00975     //
00976     switch (String_Hash(argv[2]))   //          switch
00977     {
00978         case String_Hash("-eq"):
00979         case String_Hash("=="):     //  number1   number2
00980             return number1 == number2;
00981 
00982         case String_Hash("-ne"):
00983         case String_Hash("!="):     //  number1   number2
00984             return number1 != number2;
00985 
00986         case String_Hash("-ge"):
00987         case String_Hash(">="):     //  number1    number2
00988             return number1 >= number2;
00989 
00990         case String_Hash("-gt"):
00991         case String_Hash(">"):      //  number1   number2
00992             return number1 > number2;
00993 
00994         case String_Hash("-le"):
00995         case String_Hash("<="):     //  number1    number2
00996             return number1 <= number2;
00997 
00998         case String_Hash("-lt"):
00999         case String_Hash("<"):      //  number1   number2
01000             return number1 < number2;
01001 
01002         default:                    /*      */
01003             return false;
01004     }
01005 
01006 }
01007 
01008 bool Executor::test_string_compare(const int argc, const char * const argv[])
01009 {
01010     assert(argc == 3 || argc == 4);
01011 
01012     if (argc == 3)
01013     {
01014         //
01015         switch (String_Hash(argv[1]))   //          switch
01016         {
01017             /*     */
01018             case String_Hash("-n"):  //
01019                 return true;
01020 
```

```
01021             /*       */
01022             default:
01023                 return false;
01024         }
01025     }
01026     else
01027     {
01028         //
01029         switch (String_Hash(argv[2]))   //          switch
01030         {
01031             case String_Hash("="):     //  string1  string2
01032                 return !strcmp(argv[1], argv[3]);
01033 
01034             case String_Hash("!="):    //  string1  string2
01035                 return strcmp(argv[1], argv[3]);
01036 
01037             case String_Hash("\\>"):     //  string1  string2
01038                 return strcmp(argv[1], argv[3]) > 0;
01039 
01040             case String_Hash("\\<"):     //  string1  string2
01041                 return strcmp(argv[1], argv[3]) < 0;
01042 
01043             default:
01044                 return false;
01045         }
01046     }
01047 }
```

10.34 E:/Artshell/src/lexer.l

- #define MAX_ARGUMENT_NUMBER 128

- int yylex (void)
- int yywrap ()
- int yy_lexer (int *argc, char ***argv)

- char * _argvector [MAX_ARGUMENT_NUMBER]
- int _argcounter = 0

10.34.1

10.34.1.1 MAX_ARGUMENT_NUMBER

#define MAX_ARGUMENT_NUMBER 128

lexer.l 2 .

10.34.2

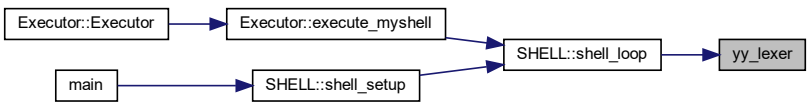
10.34.2.1 yy_lexer()

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

lexer.l 43 .

:

:

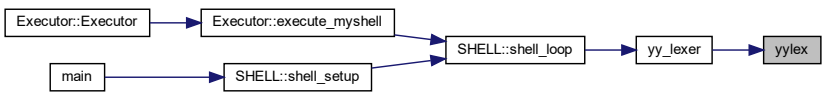


10.34.2.2 yylex()

```
int yylex (
                void  )
```

lexer.l 12 .

:



10.34.2.3 yywrap()

int yywrap ( )

lexer.l 38 .

10.34.3

10.34.3.1 _argcounter

int _argcounter = 0

lexer.l 4 .

10.34.3.2 _argvector

char* _argvector[MAX_ARGUMENT_NUMBER]

lexer.l 3 .

10.35 lexer.l

.

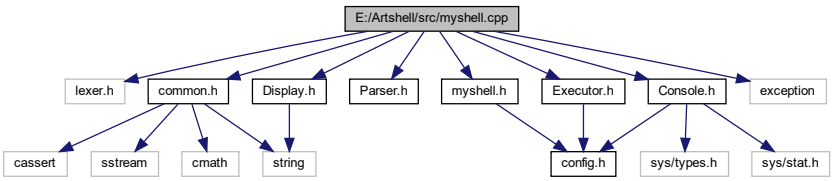
```
00001 %{
00002     #define MAX_ARGUMENT_NUMBER 128
00003     char *_argvector[MAX_ARGUMENT_NUMBER]; //
00004     int _argcounter = 0;                   //
00005 %}
00006
00007 WORD     [a-zA-Z0-9\/\.\-~]+
00008 STRINGLITERAL \"(\\.|[^\\"])*\"
00009 REDIRCT [0-9><]+
00010 SPECIAL [()|&*!]
00011
00012 %%
00013     _argcounter = 0;
00014     _argvector[0] = NULL;
00015
00016 {WORD}|{SPECIAL}|{REDIRCT}|{STRINGLITERAL} {
00017     if(_argcounter < MAX_ARGUMENT_NUMBER-1)
00018     {
00019         _argvector[_argcounter++] = (char *)strdup(yytext);
00020         _argvector[_argcounter] = NULL;
00021     }
00022 }
00023
00024 \n  return (int)_argvector;   //
00025
00026 [ \t]+
00027
00028 \#[^\n]*    ;   //   #
00029
00030 . {
00031   char str[128] = {0};
00032   sprintf(str, "Unrecognized token [%s] in input sql.", yytext);
00033 //   ParserSetError(str);
00034 }
00035
```

```
00036 %%
00037
00038 int yywrap()
00039 {
00040     return 1;
00041 }
00042
00043 int yy_lexer(int *argc, char ***argv)
00044 {
00045     yylex();
00046
00047     *argc = _argcounter;
00048     *argv = _argvector;
00049
00050     return 0;
00051 }
```

10.36 E:/Artshell/src/myshell.cpp

myshell main myshell

#include ”lexer.h”
#include ”myshell.h”
#include ”common.h”
#include ”Parser.h”
#include ”Console.h”
#include ”Display.h”
#include ”Executor.h”
#include <exception>
myshell.cpp (Include) :



- namespace SHELL

- int yy_lexer (int *argc, char ***argv)
- int SHELL::shell_setup (int argc, char *argv[], char *env[])
  shell
- int SHELL::shell_loop (Console *model, Display *view, Executor *controller, char *env[])
  shell

10.36.1

myshell main myshell

(3200105842@zju.edu.cn)

0.1

2022-07-02

myshell.cpp .

10.36.2

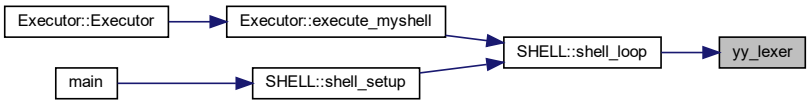
10.36.2.1 yy_lexer()

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

lexer.l 43 .

:

:



10.37 myshell.cpp

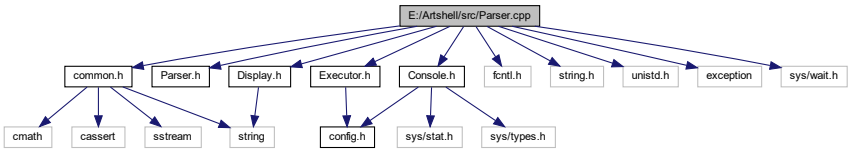
```
.
00001 //
00002 //      3200105842
00003
00015 // #define _DEBUG_
00016
00017 extern "C"
00018 {
00019     #include "lexer.h"
00020     int yy_lexer(int *argc, char ***argv);
00021 }
00022
00023 #include "myshell.h"
00024 #include "common.h"
00025 #include "Parser.h"
00026 #include "Console.h"
00027 #include "Display.h"
00028 #include "Executor.h"
00029
00030 #include <exception>
00031
00032
00033 namespace SHELL
00034 {
00035     int shell_setup(int argc, char *argv[], char *env[])
00036     {
00037         //
00038         Console *model = new Console;
00039         if (model == nullptr)
00040         {
00041             fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), "Out of Space for Console model");
00042             return 1;
00043         }
00044
00045         //
00046         Display *view = new Display(model);
00047         if (view == nullptr)
00048         {
00049             fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), "Out of Space for Display view");
00050             return 1;
00051         }
00052
00053         //
00054         Executor *controller = new Executor(model, view);
00055         if (controller == nullptr)
00056         {
00057             fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), "Out of Space for Executor controller");
00058             return 1;
00059         }
00060
00061         SHELL::shell_loop(model, view, controller, env);
00062
00063         //     MVC
00064         delete model;
00065         delete view;
00066         delete controller;
00067
00068         return 0;
00069     }
00070
00071     int shell_loop(Console* model, Display* view, Executor* controller, char *env[])
00072     {
00073         try
00074         {
00075             while (1)
00076             {
00077                 //
00078                 view->render();
00079
00080                 //
00081                 char input[BUFFER_SIZE];
00082                 int input_len = view->InputCommand(input, BUFFER_SIZE);
00083
00084                 if (input_len == 0)   //
00085                     return 0;
00086                 if (input_len < 0)   //
00087                     continue;
00088
00089                 //      buffer
00090                 YY_BUFFER_STATE bp = yy_scan_string(input);
00091                 if (bp == nullptr)
00092                 {
00093                     throw "Failed to create yy buffer state.";
```

```
00094               }
00095
00096               yy_switch_to_buffer(bp);
00097
00098               //
00099               int argument_counter = 0;
00100               char **argument_vector = nullptr;
00101               yy_lexer(&argument_counter, &argument_vector);
00102
00103               #ifdef _DEBUG_
00104               Argument_Display(argument_counter, argument_vector);
00105               #endif
00106
00107               model->ResetChildPid();
00108
00109               /*              */
00110               model->ConsoleJobListDone();
00111
00112               if (argument_counter == 0)
00113                   continue;
00114
00115               bool exit_state = Parser::shell_pipe(model, view, controller, argument_counter, argument_vector, env);
00116
00117               // view->show();   //
00118
00119               yylex_destroy();    //
00120
00121               if (exit_state == true)
00122                   break;
00123           }
00124       }
00125       catch(const char * message)
00126       {
00127           fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), message);
00128       }
00129       catch(const std::exception& e)
00130       {
00131           fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), e.what());
00132       }
00133       catch(...)
00134       {
00135           fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s\n", strerror(errno));
00136       }
00137
00138       return 0;
00139   }
00140
00141 }
```

10.38 E:/Artshell/src/Parser.cpp

```
#include "common.h"
#include "Parser.h"
#include "Console.h"
#include "Display.h"
#include "Executor.h"
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <exception>
#include <sys/wait.h>
```

Parser.cpp (Include) :



- static const char * shell_error_message (sh_err_t err)

10.38.1

(3200105842@zju.edu.cn)

0.1

2022-07-19

Parser.cpp .

10.38.2

10.38.2.1 shell_error_message()

static const char * shell_error_message (
sh_err_t err) [static]

Parser.cpp 353 .

:

10.39 Parser.cpp

```
00001
00012 #include "common.h"
00013 #include "Parser.h"
00014 #include "Console.h"
00015 #include "Display.h"
00016 #include "Executor.h"
00017
00018 #include <fcntl.h>
00019 #include <string.h>
00020 #include <unistd.h>
00021 #include <exception>
00022 #include <sys/wait.h>
00023
00025 static const char * shell_error_message(sh_err_t err);
00026
00027 bool Parser::shell_pipe(Console *model, Display* view, Executor* controller, int& argc, char *argv[], char *env[])
00028 {
00029     int count = 0;
00030     char *args[MAX_ARGUMENT_NUMBER];
00031
00032     int input_fd = model->GetInputFD();     //
00033     int output_fd = model->GetOutputFD();   //
00034
00035     int i = 0;
00036     do
00037     {
00038         if (strcmp(argv[i], "|") != 0)   //
00039         {
00040             args[count] = argv[i];
00041             count++;
00042         }
00043         else
00044         {
00045             args[count] = NULL; //
00046
00047             int channel[2];
00048             // channel[0] : read
00049             // channel[1] : write
00050             if (pipe(channel) == -1)
00051                 throw "Pipe Error,    ";
00052
00053             #ifdef _DEBUG_
00054             printf("channel: read %d write %d\n", channel[0], channel[1]);
00055             #endif
00056
00057             pid_t pid = fork(); //     fork      pid
00058             if (pid < 0)
00059             {
00060                 /*     */
00061                 throw "Fork Error,    ";
00062             }
00063             else if (pid == 0)
00064             {
00065                 /*    */
00066                 setenv("parent", getenv("shell"), 1);  //
00067                 close(channel[0]);  //
00068                 int fd = channel[1];
00069
00070                 model->SetOutputFD(fd);
00071                 model->SetOutputRedirect();
00072
00073                 dup2(fd, STDOUT_FILENO); //       channel[1]
00074
00075                 shell_execute(model, view, controller, count, args, env);
00076
00077                 /*  shell execute                  */
00078
00079                 return EXIT;
00080             }
00081             else
00082             {
00083                 /*    */
00084                 wait(NULL); //
00085
00086                 close(channel[1]);
00087                 int fd = channel[0];
00088
00089                 model->SetInputFD(fd);
00090                 model->SetInputRedirect();
00091
00092                 dup2(fd, STDIN_FILENO); //       fd
00093
```

```
00094                 count = 0;
00095             }
00096
00097         }
00098
00099         ++i;
00100     } while (i < argc);
00101
00102     #ifdef _DEBUG_
00103     printf("Parent Process\n");
00104     #endif
00105     /*          */
00106     args[count] = NULL; //
00107     bool exit_state = shell_execute(model, view, controller, count, args, env);
00108
00109     #ifdef _DEBUG_
00110     printf("pipe: Input %d Output %d Error %d\n", model->GetInputFD(), model->GetOutputFD(), model-
      >GetErrorFD());
00111     #endif
00112
00113     /*
00114                       */
00115     if (model->GetInputFD() != input_fd)    //
00116     {
00117         // dup2(model->GetSTDIN(), STDIN_FILENO);
00118         model->SetInputFD(input_fd);  //
00119         // model->ResetInputRedirect();    //
00120     }
00121
00122     if (model->GetOutputFD() != output_fd)    //
00123     {
00124         // dup2(model->GetSTDOUT(), STDOUT_FILENO);
00125         model->SetOutputFD(output_fd);  //
00126         // model->ResetOutputRedirect();    //
00127     }
00128
00129     return exit_state;
00130 }
00131
00132 int Parser::shell_parser(Console *model, Display* view, Executor* controller, int& argc, char *argv[], char *env[])
00133 {
00134     if (argc == 0)
00135         return 0;   //
00136
00137     for (int index = argc-1; index > 0; --index)    //
00138     {
00139         std::string arg(argv[index]);    //  string
00140
00144         if (arg == "<" || arg == "0<")
00145         {
00146             if (index + 1 == argc)  //
00147             {
00148                 throw "     ";
00149             }
00150
00151             if (model->GetInputRedirect())      //
00152             {
00153                 throw "     ";
00154             }
00155
00156             const char * input_file = argv[index + 1];
00157             int fd = open(input_file, O_RDONLY);
00158             if (fd < 0)
00159                 throw std::exception();
00160
00161             model->SetInputFD(fd);
00162             model->SetInputRedirect();
00163
00164             dup2(fd, STDIN_FILENO); //       fd
00165
00166             for (int jump = index + 2; jump < argc; ++jump)
00167                 argv[jump-2] = argv[jump];
00168             argc = argc - 2;
00169             argv[argc] = NULL;
00170         }
00171
00173         if (arg == ">" || arg == "1>")
00174         {
00175             if (index + 1 == argc)  //
00176             {
00177                 throw "     ";
00178             }
00179
00180             if (model->GetOutputRedirect())      //
00181             {
00182                 throw "     ";
00183             }
```

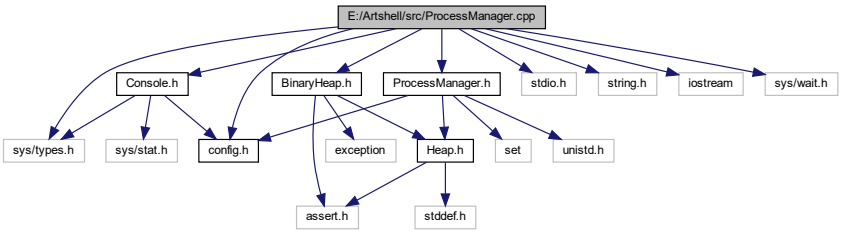
```
00184
00185             const char * output_file = argv[index + 1];
00186             int fd = open(output_file, O_WRONLY | O_TRUNC | O_CREAT, 0777&(~model->GetMask()));
00187             if (fd < 0)
00188                 throw std::exception();
00189
00190             model->SetOutputFD(fd);
00191             model->SetOutputRedirect();
00192
00193             dup2(fd, STDOUT_FILENO); //       fd
00194
00195             for (int jump = index + 2; jump < argc; ++jump)
00196                 argv[jump-2] = argv[jump];
00197             argc = argc - 2;
00198             argv[argc] = NULL;
00199         }
00200
00202         if (arg == "2>")
00203         {
00204             if (index + 1 == argc)  //
00205             {
00206                 throw "     ";
00207             }
00208
00209             if (model->GetErrorRedirect())      //
00210             {
00211                 throw "     ";
00212             }
00213
00214             const char * output_file = argv[index + 1];
00215             int fd = open(output_file, O_WRONLY | O_TRUNC | O_CREAT, 0777&(~model->GetMask()));
00216             if (fd < 0)
00217                 throw std::exception();
00218
00219             model->SetErrorFD(fd);
00220             model->SetErrorRedirect();
00221
00222             dup2(fd, STDERR_FILENO); //       fd
00223
00224             for (int jump = index + 2; jump < argc; ++jump)
00225                 argv[jump-2] = argv[jump];
00226             argc = argc - 2;
00227             argv[argc] = NULL;
00228         }
00229
00231         if (arg == "»" || arg == "1»")
00232         {
00233             /*       < >    */
00234             #ifdef _DEBUG_
00235             Argument_Display(argc, argv);
00236             #endif
00237
00238             if (index + 1 == argc)  //
00239             {
00240                 throw "     ";
00241             }
00242
00243             if (model->GetOutputRedirect())      //
00244             {
00245                 throw "     ";
00246             }
00247
00248             const char * output_file = argv[index + 1];
00249             int fd = open(output_file, O_WRONLY | O_APPEND | O_CREAT, 0777&(~model->GetMask()));
00250             if (fd < 0)
00251                 throw std::exception();
00252
00253             model->SetOutputFD(fd);
00254             model->SetOutputRedirect();
00255
00256             dup2(fd, STDOUT_FILENO); //       fd
00257
00258             for (int jump = index + 2; jump < argc; ++jump)
00259                 argv[jump-2] = argv[jump];
00260             argc = argc - 2;
00261             argv[argc] = NULL;
00262         }
00263     }
00264
00265     return 0;
00266 }
00267
00268 bool Parser::shell_execute(Console *model, Display* view, Executor* controller, int& argc, char *argv[], char *env[])
00269 {
00270     // Argument_Display(argc, argv);
00271
00272     int input_fd = model->GetInputFD();     //
```

```
00273     int output_fd = model->GetOutputFD();   //
00274     int error_fd = model->GetErrorFD();     //
00275
00276     //
00277     try
00278     {
00279         // Parser
00280         Parser::shell_parser(model, view, controller, argc, argv, env);
00281
00282         //
00283         sh_err_t err = controller->execute(argc, argv, env);
00284
00285         //
00286         if (err == SH_EXIT)
00287         {
00288             view->show(); //
00289             return true;
00290         }
00291         else if (err != SH_SUCCESS)
00292         {
00293             throw err;
00294         }
00295
00296         view->show();       //
00297         view->clear();      //
00298     }
00299     catch(const std::exception& e)
00300     {
00301         fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), e.what());
00302     }
00303     catch(const sh_err_t e)
00304     {
00305         fprintf(stderr, "\e[1;31m[ERROR]\e[0m MyShell: %s\n", shell_error_message(e));
00306     }
00307     catch(const char * message)
00308     {
00309         fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s: %s\n", strerror(errno), message);
00310     }
00311     catch(...)
00312     {
00313         fprintf(stderr, "\e[1;31m[ERROR]\e[0m %s\n", strerror(errno));
00314     }
00315
00316     if (model->GetInputRedirect())    //
00317     {
00318         int state_code = close(model->GetInputFD()); //
00319         if (state_code != 0)                         //
00320             throw std::exception();
00321
00322         dup2(model->GetSTDIN(), STDIN_FILENO);
00323         model->SetInputFD(input_fd);  //
00324         model->ResetInputRedirect();    //
00325     }
00326
00327     if (model->GetOutputRedirect())    //
00328     {
00329         int state_code = close(model->GetOutputFD()); //
00330         if (state_code != 0)                          //
00331             throw std::exception();
00332
00333         dup2(model->GetSTDOUT(), STDOUT_FILENO);
00334         model->SetOutputFD(output_fd);  //
00335         model->ResetOutputRedirect();    //
00336     }
00337
00338     if (model->GetErrorRedirect())    //
00339     {
00340         int state_code = close(model->GetErrorFD()); //
00341         if (state_code != 0)                         //
00342             throw std::exception();
00343
00344         dup2(model->GetSTDERR(), STDERR_FILENO);
00345         model->SetErrorFD(error_fd);  //
00346         model->ResetErrorRedirect();    //
00347     }
00348
00349     return false;
00350 }
00351
00353 static const char * shell_error_message(sh_err_t err)
00354 {
00355     switch (err)
00356     {
00357         case SH_FAILED:
00358             return "Shell Failed.   ";
00359         case SH_UNDEFINED:
00360             return "Undifined command.      ";
```

```
00361         case SH_ARGS:
00362             return "Argument error.    ";
00363 
00364         default:
00365             return "Unknown error.    ";
00366     }
00367 }
```

10.40 E:/Artshell/src/ProcessManager.cpp

#include "config.h"
#include "Console.h"
#include "BinaryHeap.h"
#include "ProcessManager.h"
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <sys/wait.h>
#include <sys/types.h>
ProcessManager.cpp (Include) :



10.41 ProcessManager.cpp

```
00001 #include "config.h"
00002 #include "Console.h"
00003 #include "BinaryHeap.h"
00004 #include "ProcessManager.h"
00005 
00006 #include <stdio.h>
00007 #include <string.h>
00008 #include <iostream>
00009 #include <sys/wait.h>
00010 #include <sys/types.h>
00011 
00012 job_unit::job_unit(unsigned int _id, int _pid, job_state _state, int _argc, char * _argv[])
00013             : id(_id), pid(_pid), state(_state), argc(_argc)
00014 {
00015     // argv          argv
00016     assert(argc < MAX_ARGUMENT_NUMBER);
00017     for (int i = 0; i < argc; ++i)
00018         strncpy(argv[i], _argv[i], BUFFER_SIZE);
00019 }
00020 
00021 void job_unit::PrintJob(int output_fd)
00022 {
00023     // if (argc <= 0)  //
00024     // {
00025     //     assert(false && "argument error");
00026     //     return;
00027     // }
00028 
00029     const char *State_;
```

```
00030     switch (state)  //
00031     {
00032         case Running:                                   //
00033             State_ = "Running";
00034             break;
00035         case Stopped:                                   //
00036             State_ = "Stopped";
00037             break;
00038         case Done:                                     //
00039             State_ = "Done";
00040             break;
00041         case Terminated:                                //
00042             State_ = "Terminated";
00043             break;
00044     }
00045
00046     //
00047     char buffer[BUFFER_SIZE];
00048     ssize_t write_state;
00049     snprintf(buffer, BUFFER_SIZE-1, "[%u]%c\t%s\t\t\t\t\t", id, ' ', State_);
00050     write_state = write(output_fd, buffer, strlen(buffer));
00051     if (write_state == -1)
00052         throw std::exception();
00053
00054     //
00055     if (argc > 0)
00056     {
00057         write_state = write(output_fd, argv[0], strlen(argv[0])); //
00058         if (write_state == -1)
00059             throw std::exception();
00060         for (int i = 1; i < argc; ++i)
00061         {
00062             write_state = write(output_fd, " ", 1);   //
00063             if (write_state == -1)
00064                 throw std::exception();
00065
00066             write_state = write(output_fd, argv[i], strlen(argv[i])); //
00067             if (write_state == -1)
00068                 throw std::exception();
00069         }
00070     }
00071
00072     write_state = write(output_fd, "\n", 1);   //
00073     if (write_state == -1)
00074         throw std::exception();
00075 }
00076
00077 ProcessManager::ProcessManager(/* args */)
00078 {
00079     unsigned int job_id[MAX_PROCESS_NUMBER];
00080     for (unsigned int i = 1; i <= MAX_PROCESS_NUMBER; ++i)
00081         job_id[i-1] = i;        //      id
00082     job_heap = new BinaryHeap<unsigned int>(job_id, MAX_PROCESS_NUMBER);
00083
00084     #ifdef _DEBUG_
00085     for (unsigned int i = 1; i <= MAX_PROCESS_NUMBER; ++i)
00086         printf("heap: %u\n", job_heap->extract());
00087     #endif
00088 }
00089
00090 ProcessManager::~ProcessManager()
00091 {
00092     delete job_heap;
00093 }
00094
00095 void ProcessManager::PrintJobList(int output_fd) const
00096 {
00097     for (auto job : jobs)
00098     {
00099         job.PrintJob(output_fd);
00100     }
00101 }
00102
00103 void ProcessManager::PrintJobListDone(int output_fd)
00104 {
00105     job_unit *pre_job = nullptr;
00106
00107     for (auto job : jobs)
00108     {
00109         #ifdef _DEBUG_
00110         printf("Id: %u pid: %d\n", job.id, job.pid);
00111         #endif
00112         if (pre_job != nullptr)     //
00113         {
00114             this->JobRemove(pre_job);
00115             pre_job = nullptr;
00116         }
```

```
00117
00118         /* waitpid  WNOHANG                pid
00119                 0         -1 */
00120         int stat_loc, wait_pid = waitpid(job.pid, &stat_loc, WNOHANG);
00121         #ifdef _DEBUG_
00122         printf("id: %u pid: %d wait: %d stat: %d\n", job.id, job.pid, wait_pid, stat_loc);
00123         #endif
00124         if (wait_pid == job.pid) //
00125         {
00126             job.state = Done;
00127             job.PrintJob();
00128             pre_job = &job;
00129         }
00130         else if (wait_pid < 0)  //
00131         {
00132             throw std::exception();
00133         }
00134     }
00135
00136     if (pre_job != nullptr)     //
00137         this->JobRemove(pre_job);
00138 }
00139
00140 unsigned int ProcessManager::JobInsert(int pid, job_state state, int argc, char *argv[])
00141 {
00142     try
00143     {
00144         unsigned int id = job_heap->extract();  //  id     id
00145         job_unit* newJob = new job_unit(id, pid, state, argc, argv);
00146         #ifdef _DEBUG_
00147         newJob->PrintJob();
00148         #endif
00149         jobs.emplace(*newJob);   //
00150         return id;
00151     }
00152     catch (std::exception& e)
00153     {
00154         std::cerr « e.what() « '\n';
00155         return 0;
00156     }
00157 }
00158
00159 void ProcessManager::JobRemove(job_unit * job)
00160 {
00161     assert(job->id > 0);
00162     job_heap->insert(job->id);  //  id  id
00163     jobs.erase(*job);           //
00164     // delete job;  //   set        erase set
00165     return;
00166 }
00167
00168 void ProcessManager::JobRemove(std::set<job_unit>::iterator& job)
00169 {
00170     job_heap->insert(job->id);  //  id  id
00171     jobs.erase(*job);           //
00172     // delete job;  //   set        erase set
00173     return;
00174 }
00175
00176 int ProcessManager::ForeGround(unsigned int jobid)
00177 {
00178     for (auto job : jobs)
00179     {
00180         if (job.id == jobid)
00181         {
00182             Console::child_process_id = job.pid;
00183             setpgid(job.pid, getgid());
00184
00185             //
00186             tcsetpgrp(STDIN_FILENO, job.pid);
00187             tcsetpgrp(STDOUT_FILENO, job.pid);
00188             tcsetpgrp(STDERR_FILENO, job.pid);
00189             job.state = Running;
00190
00191             kill(job.pid, SIGCONT);
00192             while(waitpid(Console::child_process_id, NULL, WNOHANG) == 0 && Console::child_process_id >= 0);
00193             Console::child_process_id = -1;
00194
00195             JobRemove(&job);
00196
00197             return jobid;
00198         }
00199     }
00200
00201     return -1;
00202 }
00203
```

```
00204
00205 int ProcessManager::BackGround(unsigned int jobid)
00206 {
00207     for (auto job : jobs)
00208     {
00209         if (job.id == jobid)
00210         {
00211             if (job.state == Running)
00212                 return 0;
00213
00214             job.state = Running;
00215
00216             kill(job.pid, SIGCONT);
00217
00218             return jobid;
00219         }
00220     }
00221
00222     return -1;
00223 }
```