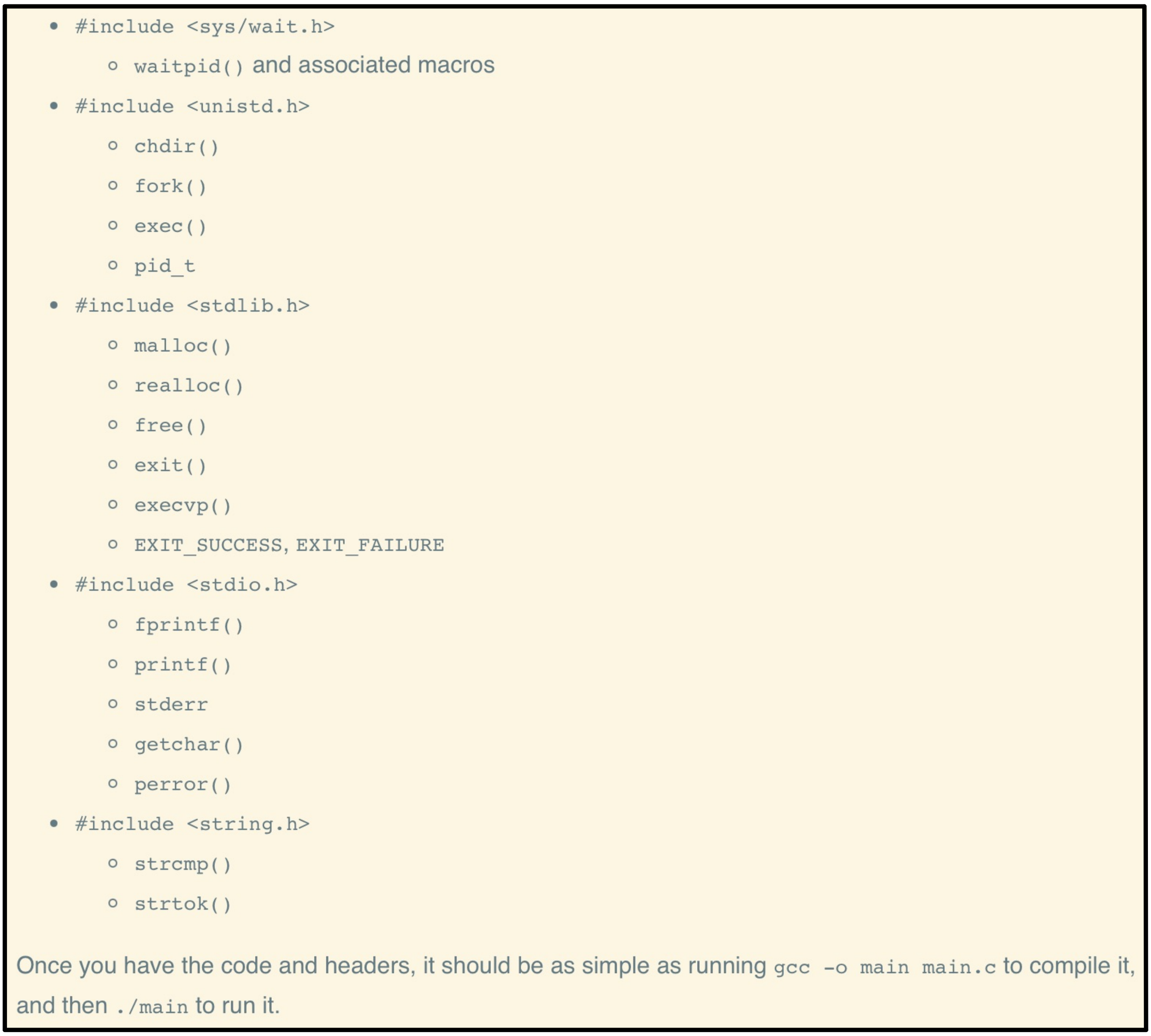
Project 1

→ **Basic Shell in C**

→ **Date : 19 June, 2024**

→ **Reference :** https://brennan.io/2015/01/16/write-a-shell-in-c/



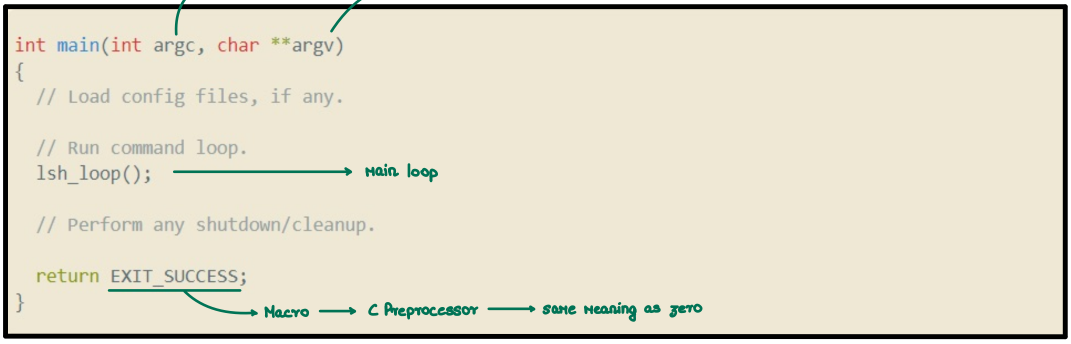
- `#include <sys/wait.h>`
  - `waitpid()` and associated macros
- `#include <unistd.h>`
  - `chdir()`
  - `fork()`
  - `exec()`
  - `pid_t`
- `#include <stdlib.h>`
  - `malloc()`
  - `realloc()`
  - `free()`
  - `exit()`
  - `execvp()`
  - `EXIT_SUCCESS`, `EXIT_FAILURE`
- `#include <stdio.h>`
  - `fprintf()`
  - `printf()`
  - `stderr`
  - `getchar()`
  - `perror()`
- `#include <string.h>`
  - `strcmp()`
  - `strtok()`

Once you have the code and headers, it should be as simple as running `gcc -o main main.c` to compile it, and then `./main` to run it.

No. of Command Line Arguements

Array of Pointers → to string

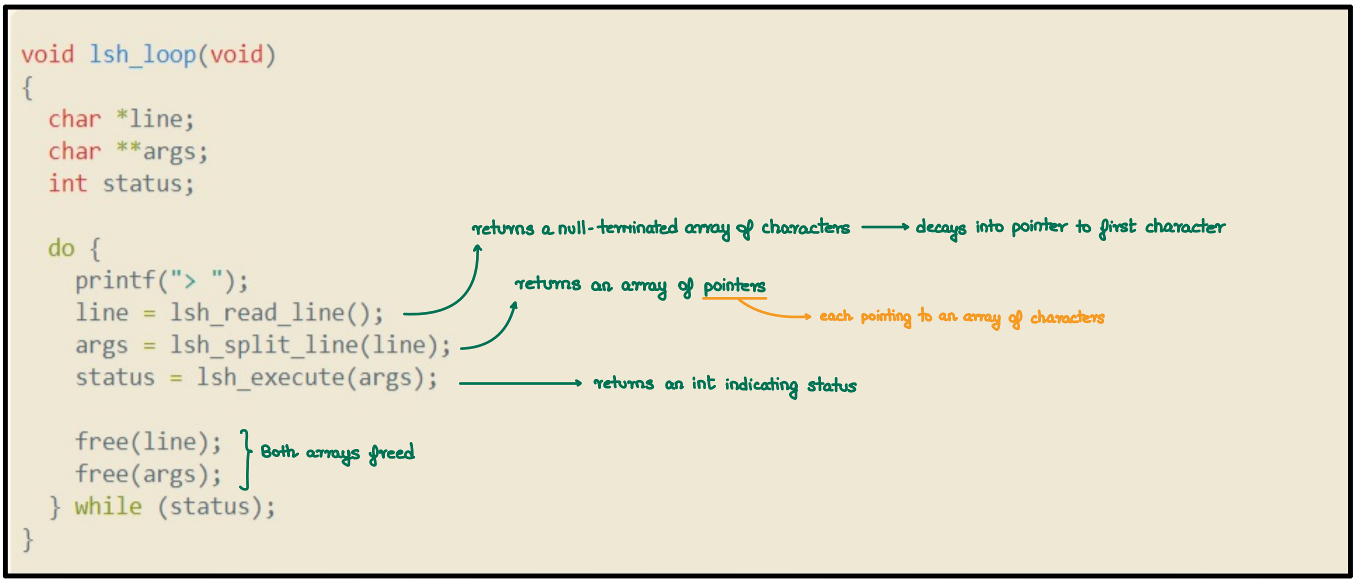
Array of Pointers → char *argv[] → decays into pointer as fn. parameter → argv[0] = program executed



```c
int main(int argc, char **argv)
{
  // Load config files, if any.

  // Run command loop.
  lsh_loop();

  // Perform any shutdown/cleanup.

  return EXIT_SUCCESS;
}
```

lsh_loop(); → Main loop

EXIT_SUCCESS → Macro → C Preprocessor → same meaning as zero



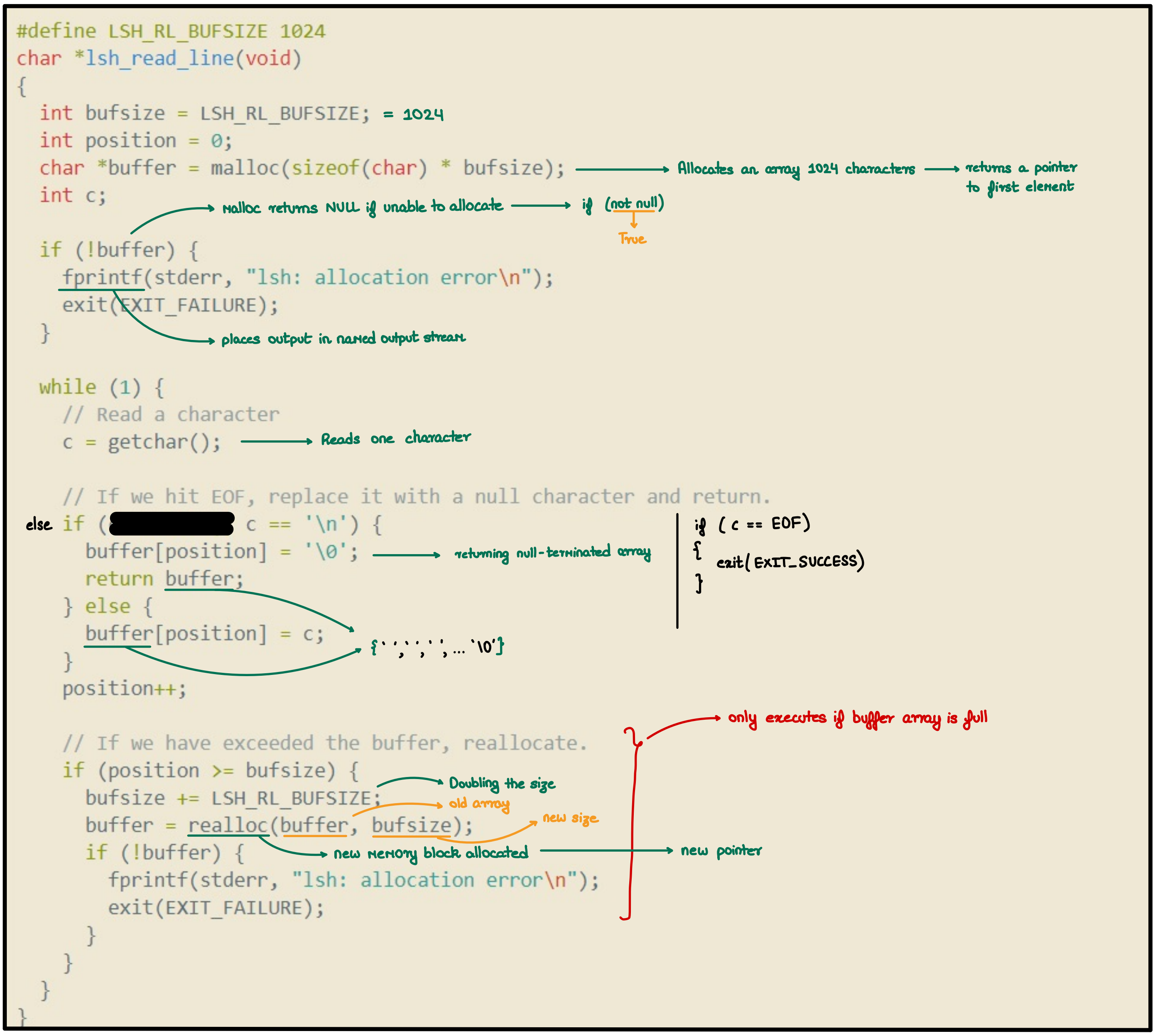
```c
void lsh_loop(void)
{
  char *line;
  char **args;
  int status;

  do {
    printf("> ");
    line = lsh_read_line();
    args = lsh_split_line(line);
    status = lsh_execute(args);

    free(line);
    free(args);
  } while (status);
}
```

returns a null-terminated array of characters → decays into pointer to first character

returns an array of pointers → each pointing to an array of characters

returns an int indicating status

Both arrays freed



```c
#define LSH_RL_BUFSIZE 1024
char *lsh_read_line(void)
{
  int bufsize = LSH_RL_BUFSIZE; = 1024
  int position = 0;
  char *buffer = malloc(sizeof(char) * bufsize);
  int c;

  if (!buffer) {
    fprintf(stderr, "lsh: allocation error\n");
    exit(EXIT_FAILURE);
  }

  while (1) {
    // Read a character
    c = getchar();

    // If we hit EOF, replace it with a null character and return.
else if ([redacted] c == '\n') {
      buffer[position] = '\0';
      return buffer;
    } else {
      buffer[position] = c;
    }
    position++;

    // If we have exceeded the buffer, reallocate.
    if (position >= bufsize) {
      bufsize += LSH_RL_BUFSIZE;
      buffer = realloc(buffer, bufsize);
      if (!buffer) {
        fprintf(stderr, "lsh: allocation error\n");
        exit(EXIT_FAILURE);
      }
    }
  }
}
```

Allocates an array 1024 characters → returns a pointer to first element

Malloc returns NULL if unable to allocate → if (not null) → True

places output in named output stream

Reads one character

returning null-terminated array

```
if ( c == EOF)
{
  exit( EXIT_SUCCESS)
}
```

{ ' ', ' ', ' ', ... '\0'}

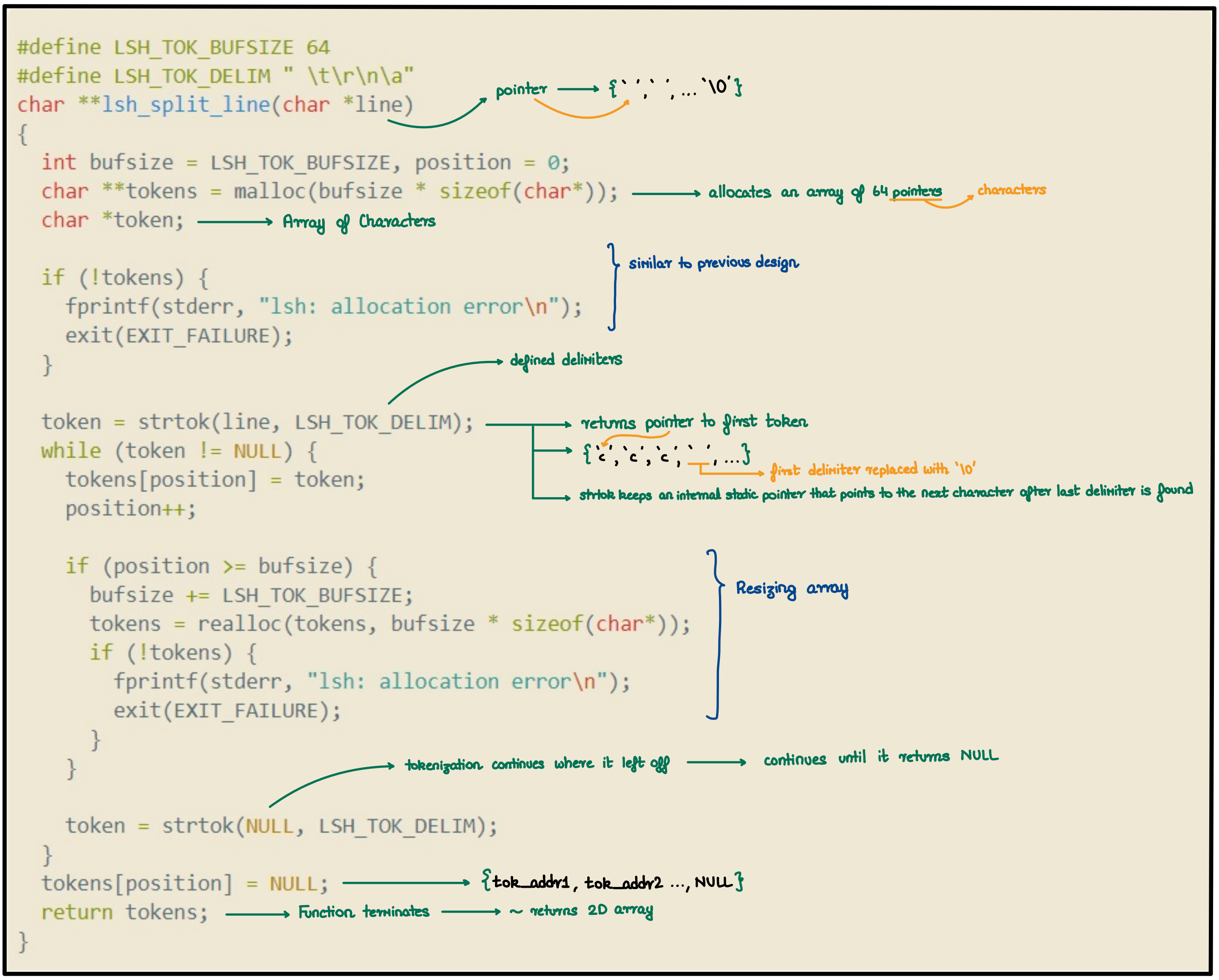
only executes if buffer array is full

Doubling the size

old array

new size

new memory block allocated → new pointer



```c
#define LSH_TOK_BUFSIZE 64
#define LSH_TOK_DELIM " \t\r\n\a"
char **lsh_split_line(char *line)
{
  int bufsize = LSH_TOK_BUFSIZE, position = 0;
  char **tokens = malloc(bufsize * sizeof(char*));
  char *token;

  if (!tokens) {
    fprintf(stderr, "lsh: allocation error\n");
    exit(EXIT_FAILURE);
  }

  token = strtok(line, LSH_TOK_DELIM);
  while (token != NULL) {
    tokens[position] = token;
    position++;

    if (position >= bufsize) {
      bufsize += LSH_TOK_BUFSIZE;
      tokens = realloc(tokens, bufsize * sizeof(char*));
      if (!tokens) {
        fprintf(stderr, "lsh: allocation error\n");
        exit(EXIT_FAILURE);
      }
    }

    token = strtok(NULL, LSH_TOK_DELIM);
  }
  tokens[position] = NULL;
  return tokens;
}
```

pointer → {` ', ` ', ... `\0'}

allocates an array of 64 pointers → characters

Array of Characters

similar to previous design

defined delimiters

returns pointer to first token

{`c', `c', `c', ` ', ...}

first delimiter replaced with `\0'

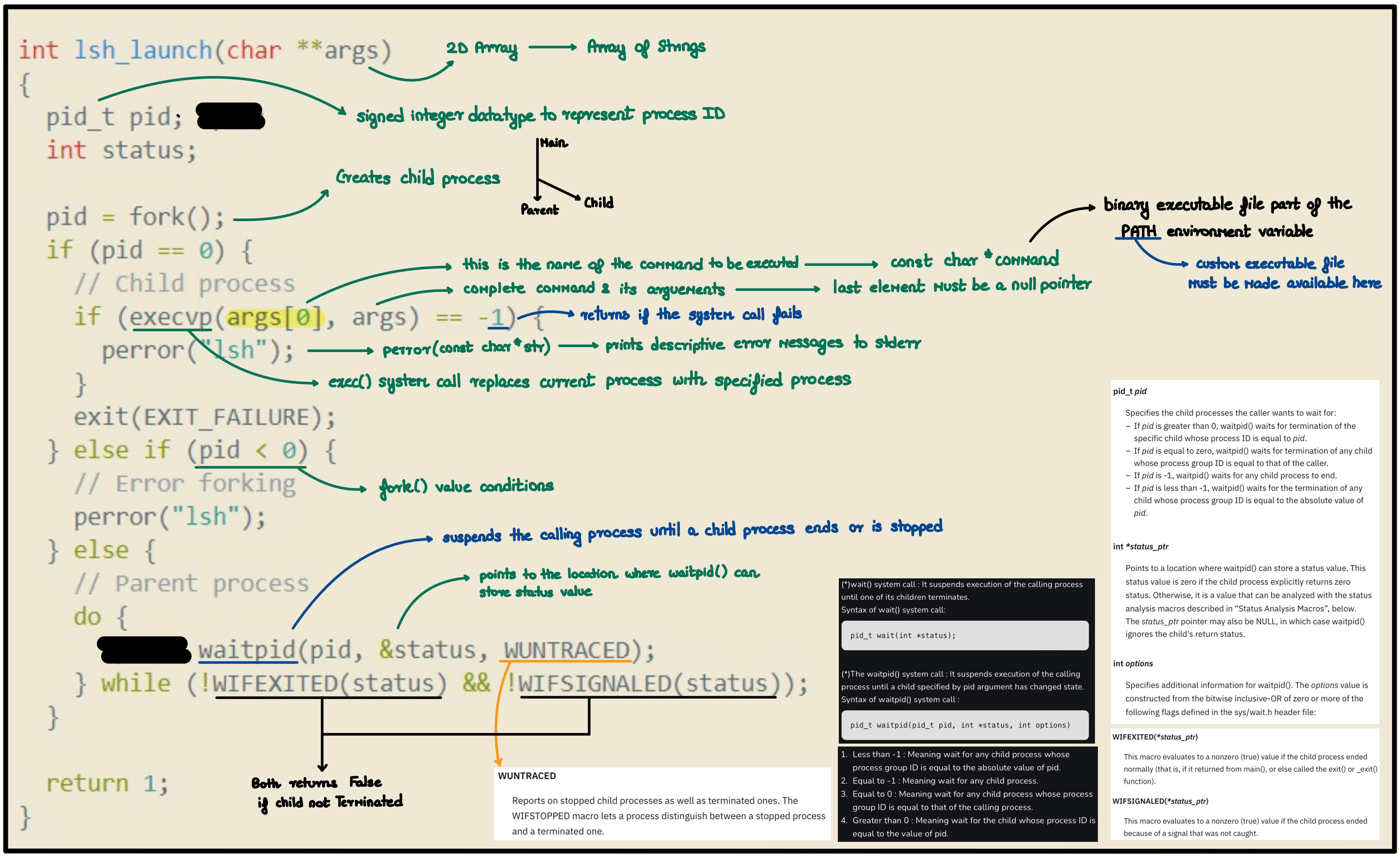
strtok keeps an internal static pointer that points to the next character after last delimiter is found

Resizing array

tokenization continues where it left off → continues until it returns NULL

{tok_addr1, tok_addr2 ..., NULL}

Function terminates → ~ returns 2D array



```c
int lsh_launch(char **args)
{
  pid_t pid; [redacted]
  int status;

  pid = fork();
  if (pid == 0) {
    // Child process
    if (execvp(args[0], args) == -1) {
      perror("lsh");
    }
    exit(EXIT_FAILURE);
  } else if (pid < 0) {
    // Error forking
    perror("lsh");
  } else {
    // Parent process
    do {
      [redacted] waitpid(pid, &status, WUNTRACED);
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
  }

  return 1;
}
```

2D Array → Array of Strings

signed integer datatype to represent process ID

Creates child process

binary executable file part of the PATH environment variable

custom executable file must be made available here

this is the name of the command to be executed → const char *command

complete command & its arguements → last element must be a null pointer

returns if the system call fails

perror(const char *str) → prints descriptive error messages to stderr

exec() system call replaces current process with specified process

fork() value conditions

suspends the calling process until a child process ends or is stopped

points to the location where waitpid() can store status value

Both returns False if child not Terminated

WUNTRACED

Reports on stopped child processes as well as terminated ones. The WIFSTOPPED macro lets a process distinguish between a stopped process and a terminated one.

(*)wait() system call : It suspends execution of the calling process until one of its children terminates.
Syntax of wait() system call:

```c
pid_t wait(int *status);
```

(*)The waitpid() system call : It suspends execution of the calling process until a child specified by pid argument has changed state.
Syntax of waitpid() system call :

```c
pid_t waitpid(pid_t pid, int *status, int options)
```

1. Less than -1 : Meaning wait for any child process whose process group ID is equal to the absolute value of pid.
2. Equal to -1 : Meaning wait for any child process.
3. Equal to 0 : Meaning wait for any child process whose process group ID is equal to that of the calling process.
4. Greater than 0 : Meaning wait for the child whose process ID is equal to the value of pid.

pid_t *pid*

Specifies the child processes the caller wants to wait for:

– If *pid* is greater than 0, waitpid() waits for termination of the specific child whose process ID is equal to *pid*.
– If *pid* is equal to zero, waitpid() waits for termination of any child whose process group ID is equal to that of the caller.
– If *pid* is -1, waitpid() waits for any child process to end.
– If *pid* is less than -1, waitpid() waits for the termination of any child whose process group ID is equal to the absolute value of *pid*.

int **status_ptr*

Points to a location where waitpid() can store a status value. This status value is zero if the child process explicitly returns zero status. Otherwise, it is a value that can be analyzed with the status analysis macros described in “Status Analysis Macros”, below. The *status_ptr* pointer may also be NULL, in which case waitpid() ignores the child's return status.

int *options*

Specifies additional information for waitpid(). The *options* value is constructed from the bitwise inclusive-OR of zero or more of the following flags defined in the sys/wait.h header file:

WIFEXITED(status_ptr*)**

This macro evaluates to a nonzero (true) value if the child process ended normally (that is, if it returned from main(), or else called the exit() or _exit() function).

WIFSIGNALED(status_ptr*)**

This macro evaluates to a nonzero (true) value if the child process ended because of a signal that was not caught.

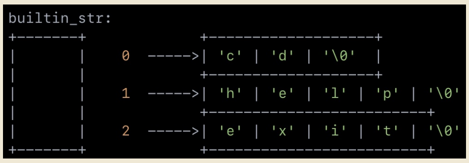
```c
/*
  Function Declarations for builtin shell commands:
 */
int lsh_cd(char **args);
int lsh_help(char **args);
int lsh_exit(char **args);
```

Declaration

each parameter is an array of strings

```c
/*
  List of builtin commands, followed by their corresponding functions.
 */
char *builtin_str[] = {
  "cd",
  "help",
  "exit"
};
```

An array of character pointers

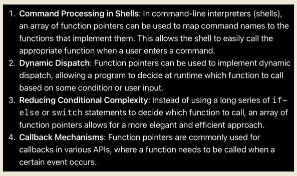


```
builtin_str:
+--------+             +--------------------+
|        |    0  ----->| 'c' | 'd' | '\0'  |
|        |             +--------------------+
|        |    1  ----->| 'h' | 'e' | 'l' | 'p' | '\0'
|        |             +--------------------------+
|        |    2  ----->| 'e' | 'x' | 'i' | 't' | '\0'
+--------+             +--------------------------+
```

```c
int (*builtin_func[]) (char **) = {
  &lsh_cd,
  &lsh_help,
  &lsh_exit
};
```

An array of function pointers → the expression means that char** is the arguement to fn. & returns an int

usage →



1. **Command Processing in Shells**: In command-line interpreters (shells), an array of function pointers can be used to map command names to the functions that implement them. This allows the shell to easily call the appropriate function when a user enters a command.
2. **Dynamic Dispatch**: Function pointers can be used to implement dynamic dispatch, allowing a program to decide at runtime which function to call based on some condition or user input.
3. **Reducing Conditional Complexity**: Instead of using a long series of `if-else` or `switch` statements to decide which function to call, an array of function pointers allows for a more elegant and efficient approach.
4. **Callback Mechanisms**: Function pointers are commonly used for callbacks in various APIs, where a function needs to be called when a certain event occurs.

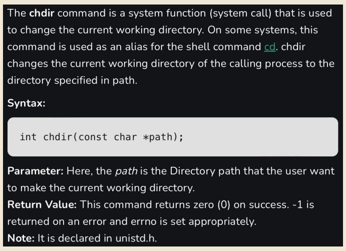
Finds the no. elements in buitlin_str array

```c
int lsh_num_builtins() {
  return sizeof(builtin_str) / sizeof(char *);
}
```

```c
/*
  Builtin function implementations.
*/
int lsh_cd(char **args)
{
  if (args[1] == NULL) {
    fprintf(stderr, "lsh: expected argument to \"cd\"\n");
  } else {
    if (chdir(args[1]) != 0) {
      perror("lsh");
    }
  }
  return 1;
}
```

checking the second element → the name of the directory → is given or not

checking if Not success



The **chdir** command is a system function (system call) that is used to change the current working directory. On some systems, this command is used as an alias for the shell command cd. chdir changes the current working directory of the calling process to the directory specified in path.

Syntax:

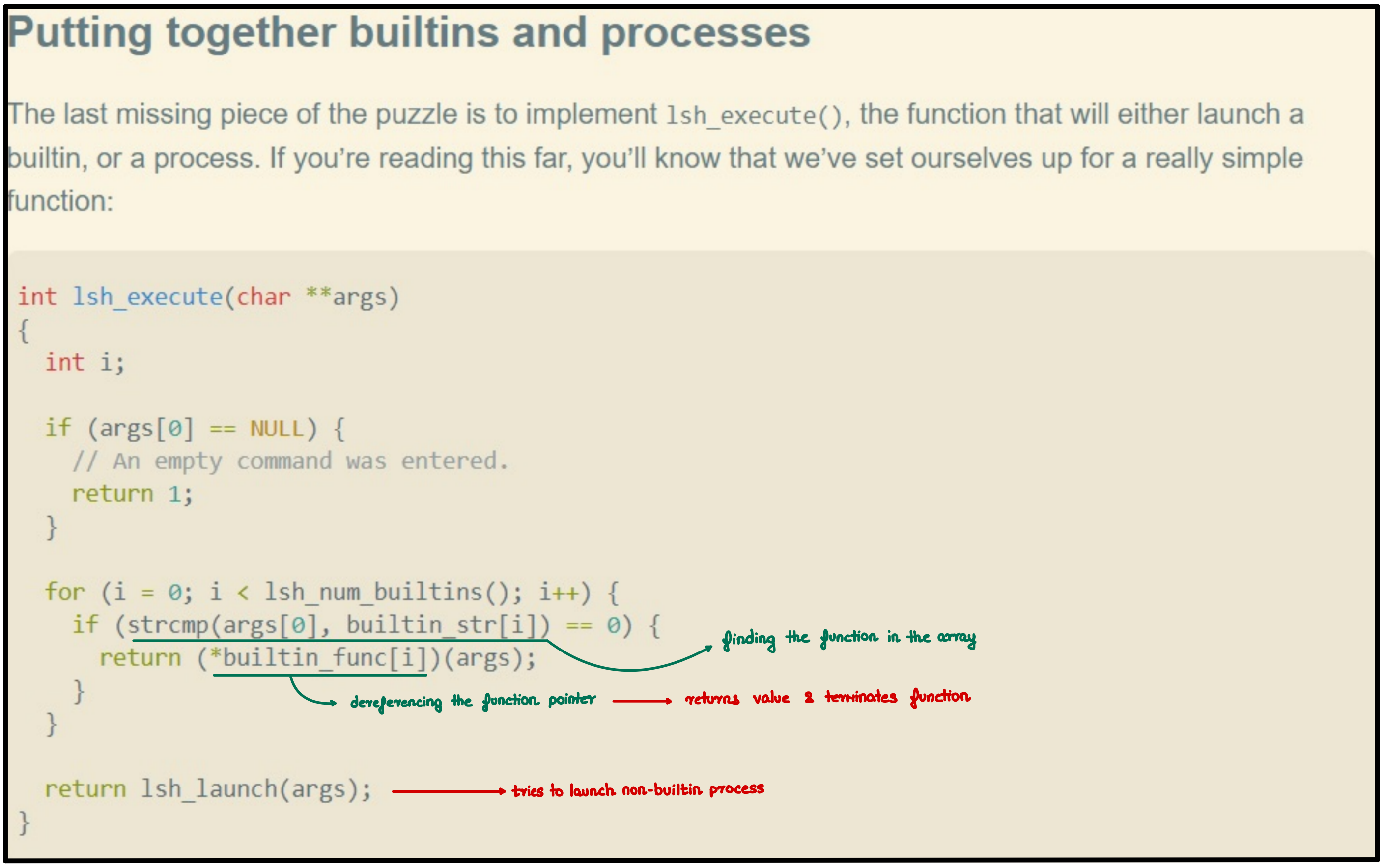
```c
int chdir(const char *path);
```

Parameter: Here, the *path* is the Directory path that the user want to make the current working directory.
Return Value: This command returns zero (0) on success. -1 is returned on an error and errno is set appropriately.
Note: It is declared in unistd.h.

```c
int lsh_help(char **args)
{
  int i;
  printf("Stephen Brennan's LSH\n");
  printf("Type program names and arguments, and hit enter.\n");
  printf("The following are built in:\n");

  for (i = 0; i < lsh_num_builtins(); i++) {
    printf("  %s\n", builtin_str[i]);
  }

  printf("Use the man command for information on other programs.\n");
  return 1;
}

int lsh_exit(char **args)
{
  return 0;
}
```

no. of commands

the name of each command



Putting together builtins and processes

The last missing piece of the puzzle is to implement `lsh_execute()`, the function that will either launch a builtin, or a process. If you're reading this far, you'll know that we've set ourselves up for a really simple function:

```
int lsh_execute(char **args)
{
  int i;

  if (args[0] == NULL) {
    // An empty command was entered.
    return 1;
  }

  for (i = 0; i < lsh_num_builtins(); i++) {
    if (strcmp(args[0], builtin_str[i]) == 0) {
      return (*builtin_func[i])(args);
    }
  }

  return lsh_launch(args);
}
```

finding the function in the array

dereferencing the function pointer → returns value & terminates function

→ tries to launch non-builtin process