<config.h>

```c
#ifndef CONFIG_H
#define CONFIG_H

#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>

#define PATH_MAX 4096
#define MAX_SEQ 16
#define MAX_PIPE 16
#define MAX_ARGS 64

typedef enum
{
    CMD_SIMPLE,
    CMD_SEQ,    // ;
    CMD_AND,    // &&
    CMD_OR,     // ||
    CMD_PIPE,   // |
} CmdType;
```

각종 헤더파일과 변수를 선언함. cmd를 parsing할 때 사용할 새로운 타입 정의 (CmdType)

```c
typedef struct
{
    char *cmd;
    CmdType type;
    int is_bg;

} ParsedCmd;

int main();
void print_prompt();
void sigchld_handler(int sig);
void trim(char **str);
int cmpfunc(const void *a, const void *b);
int ls();
int cd_cmd(char *input);
int pwd();
int parsing(char *input, ParsedCmd cmds[]);
void execute_exec(char *cmd);
void process_line(char *line);
int execute_cmd(char *cmd);
int execute_pipeline(char *cmds[], int n);

extern char *cwd;
extern int seq_cnt;
extern ParsedCmd cmds[MAX_SEQ];
extern int status;

#endif
```

마찬가지로 parsing할 때 주로 사용할 구조체를 만들었다. 명령어 하나 당 { 공백 제외 명령어, 명령어 타입, 백그라운드 여부 } 이런 식으로 정리하고 싶었기 때문이다. 그리고 모든 함수들을 선언하고, 전역 변수들을 사용하기 위해 extern으로 선언하였다.

<main.c>

```c
#include "config.h"

char *cwd = NULL;
int seq_cnt = 0;
ParsedCmd cmds[MAX_SEQ] = {0};
int status = 0;

int main ()
{
    char *input = NULL;
    size_t bufsize = 0;

    // 현재 디렉토리 받기
    cwd = getcwd(NULL, 0); // 자동 메모리 할당 및 디렉토리 경로 저장
    if (cwd == NULL) {
        // 실패 시 에러 출력
        perror("getcwd");
        return 1;
    }

    struct sigaction sa;
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
    if (sigaction(SIGCHLD, &sa, NULL) == -1)
    {
        perror("sigaction");
        return 1;
    }
```

main.c에는 main 함수가 들어있다. 우선 현재 디렉토리를 받아 전역변수 cwd에 저장한다.(프롬프트 출력 용으로) 이 때 실패 시 에러가 출력되도록 했다. 또한 sigchild_handler 함수(utils.c)로 좀비 프로세스를 정리할 수 있도록 하였다. Sigaction 구조체를 선언해 시그널을 처리할 수 있게 하고, SIGCHLD 시그널이 발생했을 때 실행할 함수 포인터를 만든다. (자식 프로세스 종료 등) 시그널 핸들러가 실행되는 동안 블록할 시그널을 비워둔다. (핸들러 실행 중에도 다른 시그널을 받음) SA_RESTART로 시그널 실행 중 중단된 시스템 호출을 자동 재시작하고, SA_NOCLDSTOP으로 자식이 종료되었을 때만 sigchld가 발생하도록 한다. 설정한 핸들러를 SIGCHLD 시그널에 등록하고 실패 시 에러 메시지를 출력하고 리턴한다.

```c
    while (1)
    {
        print_prompt();

        // input 크기 할당, 저장 및 개행 포함 길이 반환
        ssize_t nread = getline(&input, &bufsize, stdin);
        if (nread == -1)
        { // error
            perror("getline");
            break;
        }

        // \n 위치(인덱스) 반환 후 제거
        input[strcspn(input, "\n")] = '\0';

        // exit
        if (strcmp(input, "exit") == 0)
        {
            break;
        }

        process_line(input);

        for (int i = 0; i < seq_cnt; i++) {
            free(cmds[i].cmd);  // strdup() 된 문자열을 해제
            cmds[i].cmd = NULL; // 포인터 초기화
        }
    }

    free(input);
    free(cwd);

    return 0;
}
```

개략적인 구조는 print_prompt(prompt.c)로 프롬프트를 출력하고 input으로 입력 받고, 그럼 이 때 ₩n을 포함하게 되는데 strcspn으로 위치를 찾아 개행을 제거한 뒤, 명령어가 exit인지 확인한다. 그리고 아니라면 process_line으로 명령어를 처리하게 된다. 그리고 프로세스에서 사용한 메모리를 해제한다. 다시 처음으로 돌아가 명령어를 받고 … 이렇게 진행된다. 프로세스가 종료되면 계속 사용했던 input과 cwd를 해제하고 마무리된다.

<prompt.c>

```c
#include "config.h"

void print_prompt ()
{
    const char *color_reset = "\033[0m";
    const char *color_user = "\033[1;32m";  // green
    const char *color_host = "\033[1;34m";  // blue
    const char *color_dir = "\033[1;36m";   // turquoise
    const char *color_symbol = "\033[1;33m"; // yellow

    // cwd 유효성 검사
    if (cwd == NULL) {
        cwd = getcwd(NULL, 0);
        if (cwd == NULL) {
            perror("getcwd");
            printf("unknown$ ");
            return;
        }
    }

    // 유저명 받기
    char *username = getenv("USER");
    if (username == NULL)
    { // username이 없을 때
        username = "unknown_user";
    }

    // 호스트명 받기
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
```

Prompt.c에는 print_prompt 함수가 있다. 우선 프롬프트 색상에 사용할 색을 정의하고, chdir용으로 cwd를 받는다. 마찬가지로 실패했을 경우에는 에러를 출력하도록 했다. 이제 유저명과 호스트명을 받는다.

```c
    // ~ 설정
    char *home_dir = getenv("HOME");
    if (strncmp(cwd, home_dir, strlen(home_dir)) == 0)
    { // home_dir 건너뛰고 출력
        printf("%s%s@%s%s:%s~%s%s$ ", color_user, username, color_host, hostname, color_dir, cwd + strlen(home_dir),
    }
    else
    {
        printf("%s%s@%s%s:%s%s%s$ ", color_user, username, color_host, hostname, color_dir, cwd, color_symbol);
    }

    printf("%s", color_reset);
}
```

홈 디렉토리를 받고, cwd가 홈 디렉토리 경로 내라면 /home/usename을 ~로 출력하고 cwd에서 홈 디렉토리를 건너뛰고 출력한다. 아니라면 그대로 cwd를 출력한다. 프롬프트를 출력했다면 다시 color를 reset한다.

<executor.c>

```c
#include "config.h"

void execute_exec(char *cmd)
{
    char *argv[MAX_ARGS];
    int argc = 0;
    char *token = strtok(cmd, " ");

    while (token != NULL)
    {
        if (argc >= MAX_ARGS - 1)
        {
            fprintf(stderr, "Too many arguments\n");
            exit(1);
        }

        argv[argc++] = token;
        token = strtok(NULL, " ");
    }

    argv[argc] = NULL;
    execvp(argv[0], argv);
    perror("execvp failed");
    exit(1);
}
```

Executor.c에는 execute_exec, process_line, execute_cmd, execute_pipe가 있다. Execvp를 실행시키기 위해 명령어를 공백 기준으로 분할한다. 이때 argc는 인자를 세기 위한 변수이다. Strtok로 들어온 cmd를 공백 기준으로 자르고 while로 넘어가 argv에 token을 저장한다. 그리고 다음 토큰으로 넘어가기를 반복하여 token이 NULL이면 멈춘다. Execvp는 마지막 인자로 NULL이 필요하기 때문에 NULL은 따로 추가해준다. 또한, argv가 오버플로우되기 전에 argc의 개수를 확인해 막는다. Execvp가 실패했을 경우도 에러 메시지를 출력하고 중단하게 했다. 이 함수는 process_line, execute_cmd와 execute_pipe에서 사용하려고 만들었다.

```c
void process_line (char *input)
{
    seq_cnt = parsing(input, cmds); // parsing cmds
    int i = 0;

    while (i < seq_cnt)
    {
        ParsedCmd current = cmds[i];

        // and or 처리
        if (i > 0)
        {
            CmdType prev_type = cmds[i - 1].type;
            if ((prev_type == CMD_AND && status != 0) || (prev_type == CMD_OR && status == 0))
            {
                i++;
                continue;
            }
        }
```

다음은 process_line으로, 명령어를 처리하는 총괄 함수로 볼 수 있고 다양한 함수를 호출하게 된다. Input을 parsing함수(parser.c)를 통해 명령어를 뽑아내 cmds에 저장하고, seq_숫로 명령어 개수를 반환하게 된다. While로 들어가면 현재 다룰 명령어가 cmds[i]가 되는데 이를 current라 칭하겠다. 사이 사이에 있는 연산자는 앞의 명령어에 귀속되도록 했기 때문에 (parsing 함수에서 확인할 수 있음 / ls | grep out 이면 cmds[0].type = CMD_PIPE, cmds[1].type = CMD_SIMPLE) 앞의 명령어의 type에 따라 움직이게 설정했다. prev_type이 and이고 실패했거나, prev_type이 or이고 성공했을 시에는 이번 명령을 건너뛰도록 하였다. 이로써 &&과 ||이 작동하게 된다.

```c
        // pipe 명령 수 세고 execute_pipeline 호출
        if (current.type == CMD_PIPE)
        {
            int j = i;
            char *pipe_cmds[MAX_PIPE];
            int pipe_cnt = 0;

            pipe_cmds[pipe_cnt++] = current.cmd;

            while (cmds[j].type == CMD_PIPE)
            {
                j++;
                pipe_cmds[pipe_cnt++] = cmds[j].cmd;
            }

            status = execute_pipeline(pipe_cmds, pipe_cnt);
            i = j + 1;
            continue;
        }
```

다음은 pipe를 검사하게 된다. 파이프 타입이라면 연결되어 있는 파이프의 개수를 셈과 동시에 pipe_cmds[]에 명령어를 저장한다. 그리고 execute_pipeline을 호출하여 파이프라인을 실행하고, 성공여부를 반환한다. 그리고 현재 cmd위치를 가리키는 i를 j+1로 초기화한 후 continue하여 파이프에 묶인 명령어 바로 다음으로 넘어가게 했다.

```
        else if (current.is_bg) // case of bg
        {
            pid_t pid = fork();
            if (pid == 0)
            {
                // child
                execute_exec(current.cmd);
            }
            else
            {
                // parent
                printf("[bg] pid: %d\n", pid);
                status = 0;
            }
        }
        else // 일반
        {
            status = execute_cmd(current.cmd);
        }

        i++;
    }
}
```

이제 백그라운드 여부를 판단한다. Else if로 한 이유는 파이프라인 내에서 백그라운드 명령어가 실행될 수 없기 때문이다. Fork를 호출해 자식을 만들고 자식에서는 execute_exec을 이용해 실행을 한다. Else로 나눠진 부분은 부모가 실행할 부분이다. 백그라운드 실행이기 때문에 waitpid하지 않고 Pid를 출력한다. 보통 백그라운드 실행은 성공으로 보기 때문에 status = 0으로 해준다. 또한 마지막에 else로 취급되는 경우는 CMD_SEQ이거나 CMD_AND인데 앞의 명령어가 성공했다거나 등이 있겠다. 이 경우에는 execute_cmd를 호출하여 명령을 실행하고 status를 반환한다. 이 일련의 과정들이 끝나면 i를 증가시켜 다음 명령어로 넘어간다.

```
int execute_cmd (char *exeCmd)
{
    if (strncmp(exeCmd, "cd", 2) == 0) return cd_cmd(exeCmd);
    else if (strncmp(exeCmd, "pwd", 3) == 0) return pwd();
    else if (strncmp(exeCmd, "ls", 2) == 0) return ls();
    else
    {
        // 외부 명령일 경우
        pid_t pid = fork();
        if (pid < 0)
        {
            perror("fork");
            return 1;
        }
        else if (pid == 0)
        {
            // child
            execute_exec(exeCmd);
            return 0;
        }
```

다음은 execute_cmd 함수이다. Cmd가 cd로 시작하면 cd_cmd를 호출하고 pwd라면 pwd, ls라면 ls를 호출하고 성공 여부를 반환한다. 이도저도 아니고 외부 명령이라면 fork 함수를 호출하여 분기한다. Fork에 실패한 경우에는 에러를 출력하고 종료시킨다. 분기에

성공하면 execute_exec으로 명령을 실행한다.

```
        else
        {
            int wstatus;
            waitpid(pid, &wstatus, 0);
            if (WIFEXITED(wstatus)) {
                return WEXITSTATUS(wstatus);
            } else {
                return 1;
            }
        }
    }
}
```

이번엔 포그라운드이기 때문에 부모 프로세스가 waitpid()를 통해 자식이 끝날 때까지 기다린다. 자식 프로세스의 종료 코드를 받아 status를 반환하게 된다.

```
int execute_pipeline(char *pipeCmds[], int count)
{
    int pipefd[2];
    int prev_fd = -1;
    pid_t pids[MAX_PIPE];

    for (int i = 0; i < count; i++)
    {
        if (pipe(pipefd) == -1)
        {
            perror("pipe");
            exit(1);
        }
```

마지막으로는 execute_pipeline 함수이다. 읽는 용, 쓰는 용으로 쓰기 위해 두 칸짜리 pipefd를 선언한다. 파이프 명령어 수만큼 돌아가며 파이프를 생성하고 만약 실패하면 오류 메시지를 띄우고 exit한다.

```
        pid_t pid = fork();
        if (pid == 0)
        {
            if (i > 0)
            {
                dup2(prev_fd, 0); // 이전 명령의 출력을 현재 명령의 입력으로
                close(prev_fd);
            }

            if (i < count - 1)
            {
                dup2(pipefd[1], 1); // 현재 명령의 출력을 pipe로
                close(pipefd[0]);
                close(pipefd[1]);
            }

            execute_exec(pipeCmds[i]);
        }
```

자식을 생성하고 성공하면 자식에서 파이프를 엮는 역할과 실행을 한다. 이전 명령이 있을 경우(i>0) prev_fd를 표준 입력(0)으로 복제한 후 이제 사용하지 않는 prev_fd를 닫는다. 첫 명령을 포함해 마지막 명령의 이전 명령까지는 표준 출력(1)을 다음 입력으로 연결하게 된다. 입력도 이전 출력과 연결하고 현재 출력도 다음 입력으로 연결하면 읽기 쓰기가 완료됐기 때문에 파이프fd를 닫는다. 두 파이프의 흐름을 제어했으면 execute_exec으로 execvp를 실행한다.

```
        else if (pid > 0)
        {
            pids[i] = pid;
            if (prev_fd != -1) close(prev_fd);
            if (i < count - 1)
            {
                close(pipefd[1]);
                prev_fd = pipefd[0];
            }
        }
        else
        {
            perror("fork failed");
            return 1;
        }
    }
```

부모에서는 생성된 자식 프로세스의 pid를 pids 배열에 저장한다. (나중에 waitpid로 각 자식 프로세스를 기다리기 위해 필요함) 또한, 이전 명령의 출력 파이프 읽기는 이제 더 이상 필요 없으므로 닫는다. (리소스 누수 방지) 마지막 명령 전이라면(I<count-1) 다음 명령을 위해 파이프를 유지해야하므로 pipefd[1]은 필요없으니 닫고, pipefd[0]을 prev_fd에 저장해 다음 루프에서 사용할 수 있게 한다. Pid < 0인 경우에는 에러 메시지를 출력하고 status = 1을 반환한다.

```
    for (int i = 0; i < count; i++)
    {
        int wstatus;
        waitpid(pids[i], &wstatus, 0);

        if (i == count - 1)
        { // 마지막 명령의 status만 기록
            if (WIFEXITED(wstatus))
            {
                status = WEXITSTATUS(wstatus);
            }
            else
            {
                status = 1;
            }
        }
    }

    return status;
}
```

여기서는 각 자식의 프로세스가 끝날 때까지 기다리는 역할을 한다. Pids[i]에 해당하는 자식이 종료될 때까지 기다리고 그 종료 상태를 wstatus에 반영한다. 파이프 덩어리 뒤에 명령이 있을 때 덩어리의 마지막 명령어만 의미가 있으므로 그때 WIFEXITED(wstatus)로 정상 종료됐는지 확인한다. 정상 종료됐다면 exit code를 status에 저장하고 아니라면 status = 1로 설정한 후 status를 반환한다.

<parser.c>

```c
#include "config.h"

int parsing(char *input, ParsedCmd cmds[])
{
    int count = 0;
    char *p = input;
    trim(&p);

    while (*p && count < MAX_SEQ)
    {
        int bg = 0;
        char *op = strpbrk(p, "&|;");

        // nontype
        if (!op)
        {
            cmds[count++] = (ParsedCmd){ .cmd = strdup(p), .type = CMD_SIMPLE, .is_bg = bg };
            break;
        }
```

파싱한 명령어 개수를 세기 위해 count, input을 가리키는 포인터를 만들고 p를 trim(utils.c)한다. While이 돌아갈 때 백그라운드 여부를 판단할 변수 bg를 0으로 초기화 한다. 포인터 op는 p에서 & 또는 | 또는 ;이 처음 등장하는 위치를 가리킨다. 만약 그게 없다면 cmds[count]의 cmd는 p자체가 되고 CMD_SIMPLE을 부여하며 백그라운드 실행이 아님으로 기록된다.

```c
        // check type
        CmdType type;
        if (op[0] == '&' && op[1] == '&')
        {
            type = CMD_AND;
        }
        else if (op[0] == '|' && op[1] == '|')
        {
            type = CMD_OR;
        }
        else if (op[0] == '|')
        {
            type = CMD_PIPE;
        }
        else if (op[0] == ';')
        {
            type = CMD_SEQ;
        }
        else
        {
            type = CMD_SIMPLE; // bg
            bg = 1;
        }
```

여기는 본격적으로 type을 결정하는 부분인데, 만약 &&이 연속으로 나오면 CMD_AND,

else if ||가 나오면 CMD_OR, else if |라면 CMD_PIPE, else if ;가 나오면 CMD_SEQ, 이 모든게 아닐 경우에는 CMD_SIMPLE로 하고 백그라운드라고 표시해준다. 정상적으로 명령어가 입력됐을 경우에는 (echo hi& && cd 같은 것이 아닐 때) 잘 구분된다.

```
        char *end = op;
        *end = '\0';
        trim(&p);
        cmds[count++] = (ParsedCmd){ .cmd = strdup(p), .type = type, .is_bg = bg };

        if (cmds[count - 1].type == CMD_AND || cmds[count - 1].type == CMD_OR)
        {
            p = op + 2;
        } else p = op + 1;

        trim(&p);
    }

    return count;
}
```

이제 널문자로 명령어까지만 자르고, trim하여 공백을 없애준다. Cmds[count]에 정보를 저장하고 +1한 후, +1하기 전 cmd의 type이 and거나 or일 경우에는 위치를 두 칸 이동시키고 아닐 경우에는 한 칸만 이동한다. 그리고 다시 trim한 후 다음 반복으로 넘어간다. 마지막으로는 명령어 수 count를 반환한다.

<builtin.c>

```c
#include "config.h"

int ls()
{
    struct dirent *entry;
    DIR *dp = opendir(".");
    if (dp == NULL)
    {
        perror("opendir");
        return 1;
    }

    // 파일 이름들 저장
    char **files = NULL;
    int count = 0;

    while ((entry = readdir(dp)) != NULL)
    {
        // 숨김 파일 제외
        if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0 && entry->d_name[0] != '.')
        {
            files = realloc(files, sizeof(char *) * (count + 1));
            if (files == NULL)
            {
                perror("realloc");
                closedir(dp);
                return 1;
            }
```

Built.c에는 ls, cd_cmd, pwd 함수가 있다. ls는 cd랑 pwd말고 추가로 구현해본 것이다. 디렉토리를 열고 실패하면 오류를 띄우고 리턴한다. 알파벳 정렬을 하기 위해 파일 이름들을 저장할 files를 만들었고, 개수를 초기화했다. ., .. 와 숨김 파일이 아닐 경우 files의 크기를 동적으로 하나씩 증가시킨다. (실패하면 리턴)

```c
            files[count] = strdup(entry->d_name);
            if (files[count] == NULL)
            {
                perror("strdup");
                closedir(dp);
                return 1;
            }

            count++;
        }
    }

    closedir(dp);
```

파일 이름을 복사해서 files에 넣는다. (실패하면 리턴) 마지막으로 count를 증가시키고 while을 다 돌면 디렉토리를 닫는다.

```
    qsort(files, count, sizeof(char *), cmpfunc);

    for (int i = 0; i < count; i++)
    {
        printf("%s  ", files[i]);
        free(files[i]);
    }

    printf("\n");

    free(files);
    return 0;
}
```

이제 cmpfunc에 따라 qsort로 files를 알파벳 순으로 정렬하고 files[i]를 프린트함과 동시에 free한다. (realloc) 마지막으로 files 자체도 free하고 0을 반환한다. (strdup)

```
int cd_cmd (char *input)
{
    char *cmd = input + 2;
    trim(&cmd);
    char path[PATH_MAX];
    char *home_dir = getenv("HOME");

    // cd 일 경우
    if (*cmd == '\0')
    {
        if(home_dir == NULL)
        {
            fprintf(stderr, "cd: HOME not set\n");
            return 1;
        }

        cmd = home_dir;
    }
```

input으로 들어오는 것이 이미 parsing되어 trim된 상태의 명령어이기 때문에 맨앞 두자리가 cd이므로 cd를 제거해준다. 그리고 또 trim한다. 명령어가 cd뿐이었다면(cmd는 널문자가 됨) cmd를 홈디렉토리로 만든다. 이때 홈디렉토리가 없다면 오류를 띄우고 리턴한다.

```
if (cmd[0] == '~')
{
    if (home_dir == NULL)
    {
        fprintf(stderr, "cd: HOME not set\n");
        return 1;
    }

    if (cmd[1] == '\0')
    {
        snprintf(path, sizeof(path), "%s", home_dir);
    }
    else if (cmd[1] == '/')
    {
        if (snprintf(path, sizeof(path), "%s%s", home_dir, cmd + 1) >= sizeof(path))
        {
            fprintf(stderr, "cd path too long\n");
            return 1;
        }
    }
    else
    {
        fprintf(stderr, "cd: unsupported ~ syntax\n");
        return 1;
    }

    cmd = path;
}
```

다음은 상대경로 ~를 사용했을 경우이다. 마찬가지로 홈디렉토리를 받아오지 못했다면 리턴하고, if 첫번째 분기는 'cd ~'일 경우이다. 이때 path에 홈디렉토리를 붙여넣는다. 'cd ~/.....' 이런 형식일 경우에는 경로를 홈디렉토리+입력받은 ~제외하고 /밑에주소를 붙여서 path에 넣는다. 만약 path의 PATH_MAX를 초과하면 에러를 띄우고 리턴한다. 만약 이도 저도 아닐 경우에는 잘못된 명령이므로 syntex error를 띄우고 리턴한다. 마지막으로 cmd = path로 갱신해준다.

```
    // 디렉토리 변경
    if (chdir(cmd) != 0)
    {
        perror("cd");
        return 1;
    }

    // cwd 갱신
    char *new_cwd = getcwd(NULL, 0);
    if (new_cwd == NULL)
    {
        perror("getcwd");
        return 1;
    }

    free(cwd);
    cwd = new_cwd;
    return 0;
}
```

이제 갱신한 cmd로 chdir하고 만약 실패하면 오류를 띄운다. 변수 cwd도 갱신해야하므로 new_cmd로 현재 cwd를 받고 (실패시 에러 출력 후 리턴) 기존 cwd를 해제하고 new_cwd를 저장한다.

```
int pwd ()
{
    printf("%s\n", cwd);
    return 0;
}
```

마지막으로 pwd 함수이다. 현재 cwd를 출력해주고 0을 반환한다.

<utils.c>

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>

void sigchld_handler(int sig) {
    int saved_errno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0);
    errno = saved_errno;
}
```

Utils.c에는 sigchld_handler, trim, cmpfunc 함수가 있다. Sigchld_handler 함수는 자식 프로세스가 종료되었을 때 호출되어 좀비 프로세스를 없애는 역할을 한다. 시스템 호출로 중단될 수 있기 때문에 실행 전에 errno 값을 먼저 저장해놓고, 모든 자식 프로세스(-1)에 대해 (종료된 자식이 없으면 안기다림(WHOHANG)) 모든 종료된 자식을 회수한다. 핸들러 실행 중 시스템 콜 같은게 실패하며 errno가 바뀔 수 있으므로 호출 이전 상태로 복원한다.

```c
void trim(char **str)
{
    if (*str == NULL || **str == '\0') return;

    // 앞쪽 공백 제거
    while(**str == ' ') (*str)++;

    // 뒷쪽 공백 제거
    char *end = *str + strlen(*str) - 1;
    while (end > *str && *end == ' ')
    {
        *end = '\0';
        end--;
    }
}
```

Trim은 앞뒤 공백을 제거해주는 함수이다. **str이 공백이라면 *str을 한칸씩 옮기면서 앞쪽 공백을 제거하게 된다. 포인터 end를 선언하여 마지막 위치를 가리키게 하고, 공백일 경우 널 문자로 바꿔가며 차례차례 지운다.

```c
int cmpfunc(const void *a, const void *b)
{
    const char **str1 = (const char **)a;
    const char **str2 = (const char **)b;
    return strcmp(*str1, *str2);
}
```

이 함수는 ls에서 qsort하기 위해 만들어졌다. Qsort는 void*형 요소가 필요하므로 const void *a, const void * b로 선언한다. 내부에서 char**로 형변환하여 각각 str1과 str2로 둔 후 Strcmp로 두 문자열을 사전 순으로 비교한다.