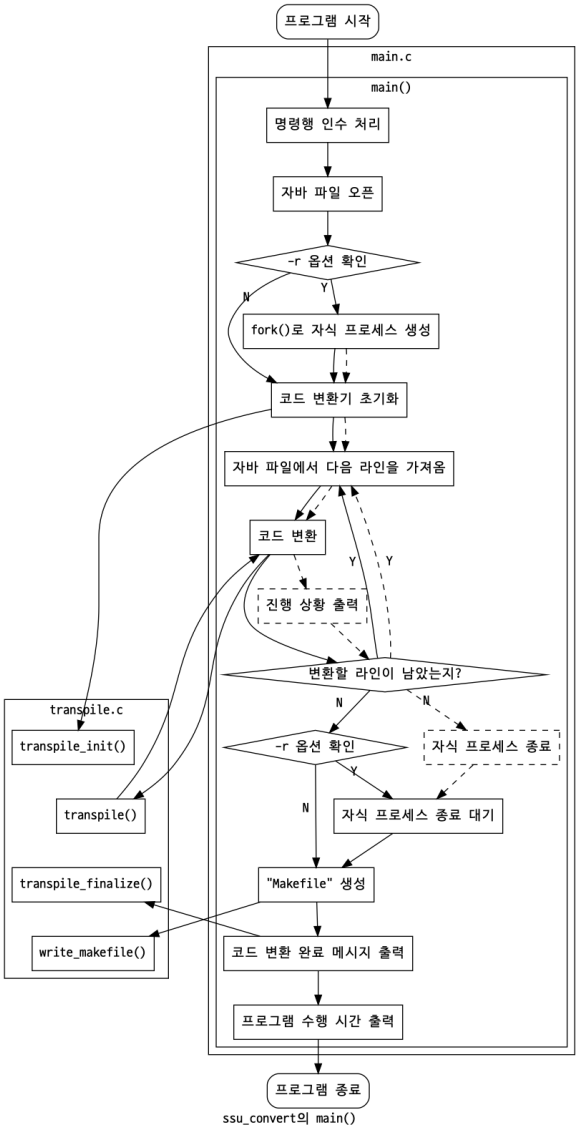
컴퓨터학부 20142314 김승호

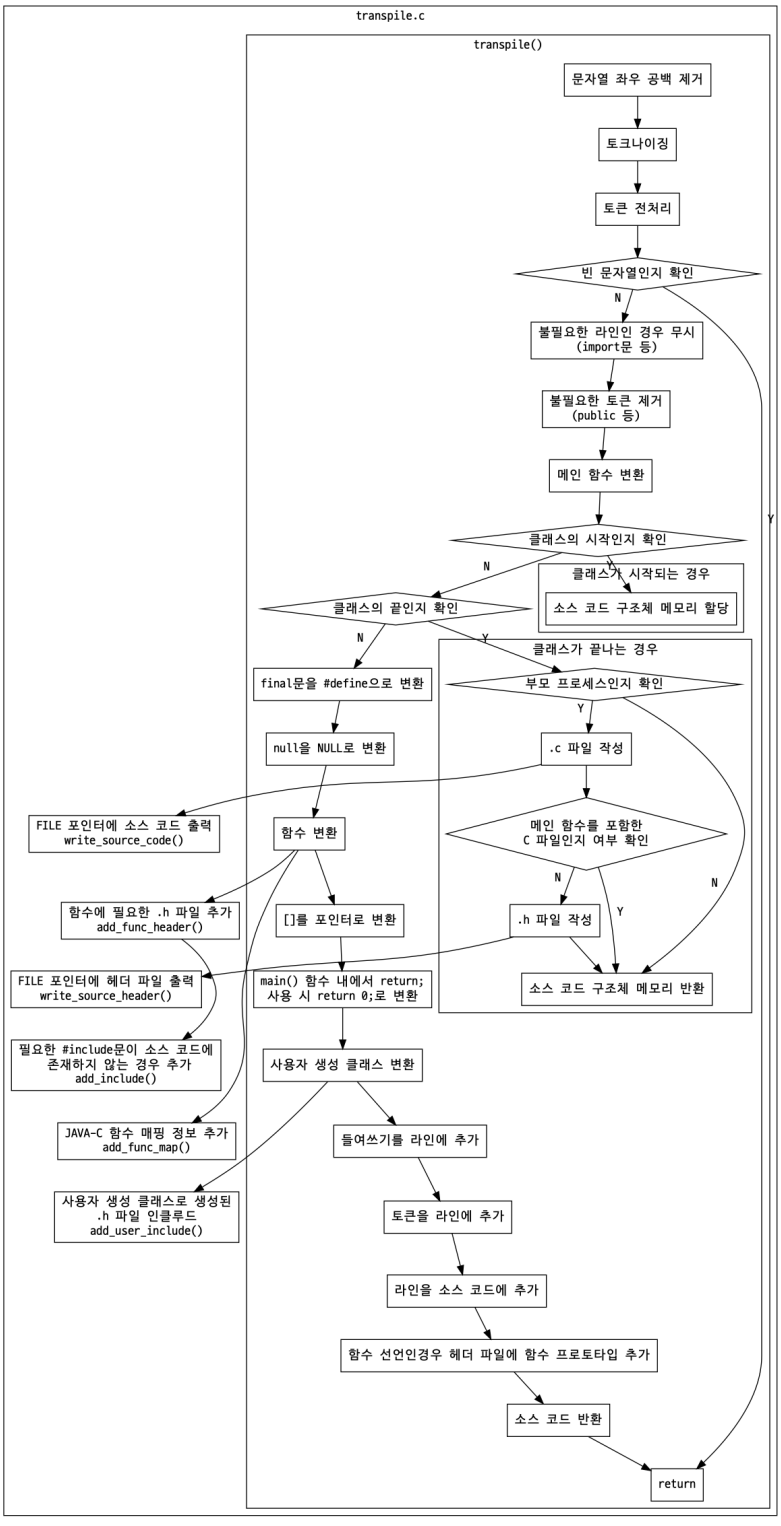
1. 과제 개요

ssu_convert는 Java 언어 프로그램을 C 언어 프로그램으로 자동 변환하는 프로그램이다.
프로그램은 한줄씩 Java 코드를 읽어 C 코드로 변환한다. 여러 개의 클래스로 이루어진 자파 파일의 경우에는 클래스에 따라 자동으로 분할되며, 프로그램이 자동적으로 Makefile 파일을 생성하기 때문에 편리하게 생성된 C 파일을 컴파일 할 수 있다. 명령행 인수로 주어지는 옵션에 따라 변환하는 과정을 보여주거나, 변환할 파일 또는 변환된 파일 및 그 파일에 관련된 정보들을 표시하도록 할 수 있다.

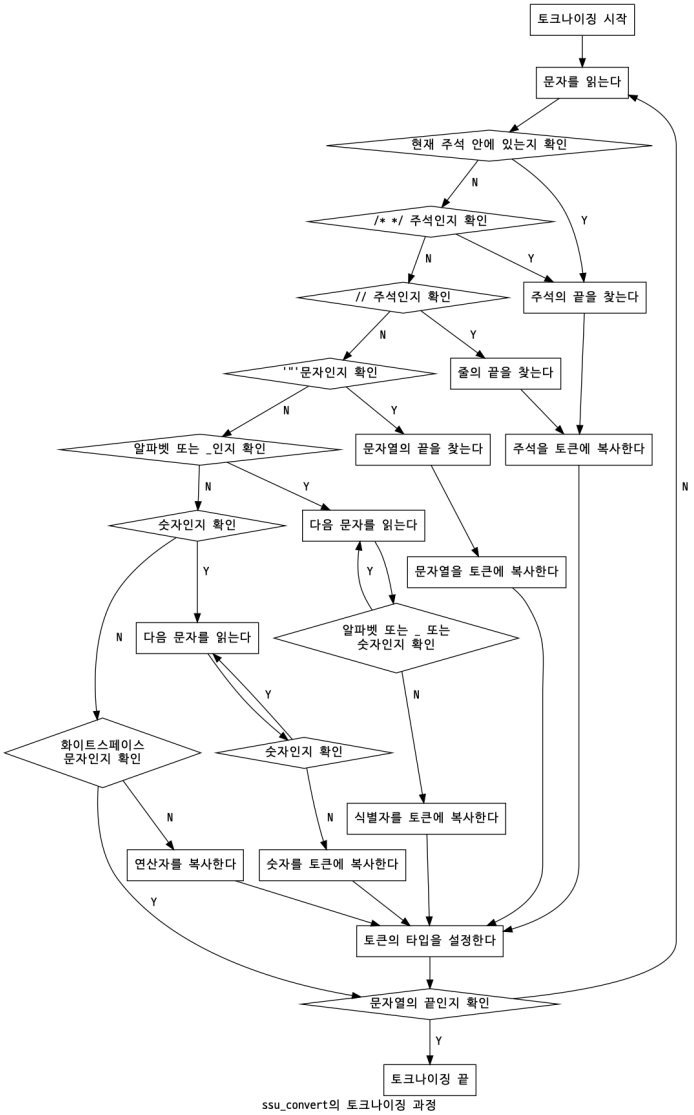
2. 설계



위 그림에서는 main() 함수에서 일어나는 대략적인 흐름을 볼 수 있다.
main.c에서는 명령행 인자를 처리하고, 파일을 오픈하며 상황에 맞게 transpile.c의 함수들을 호출하여 파일을 변환한다.
이 프로그램에서는 fork를 통해 자식 프로세스가 생성 될 수 있는데, 자식 프로세스의 흐름은 점선을 통해 표현되었다.

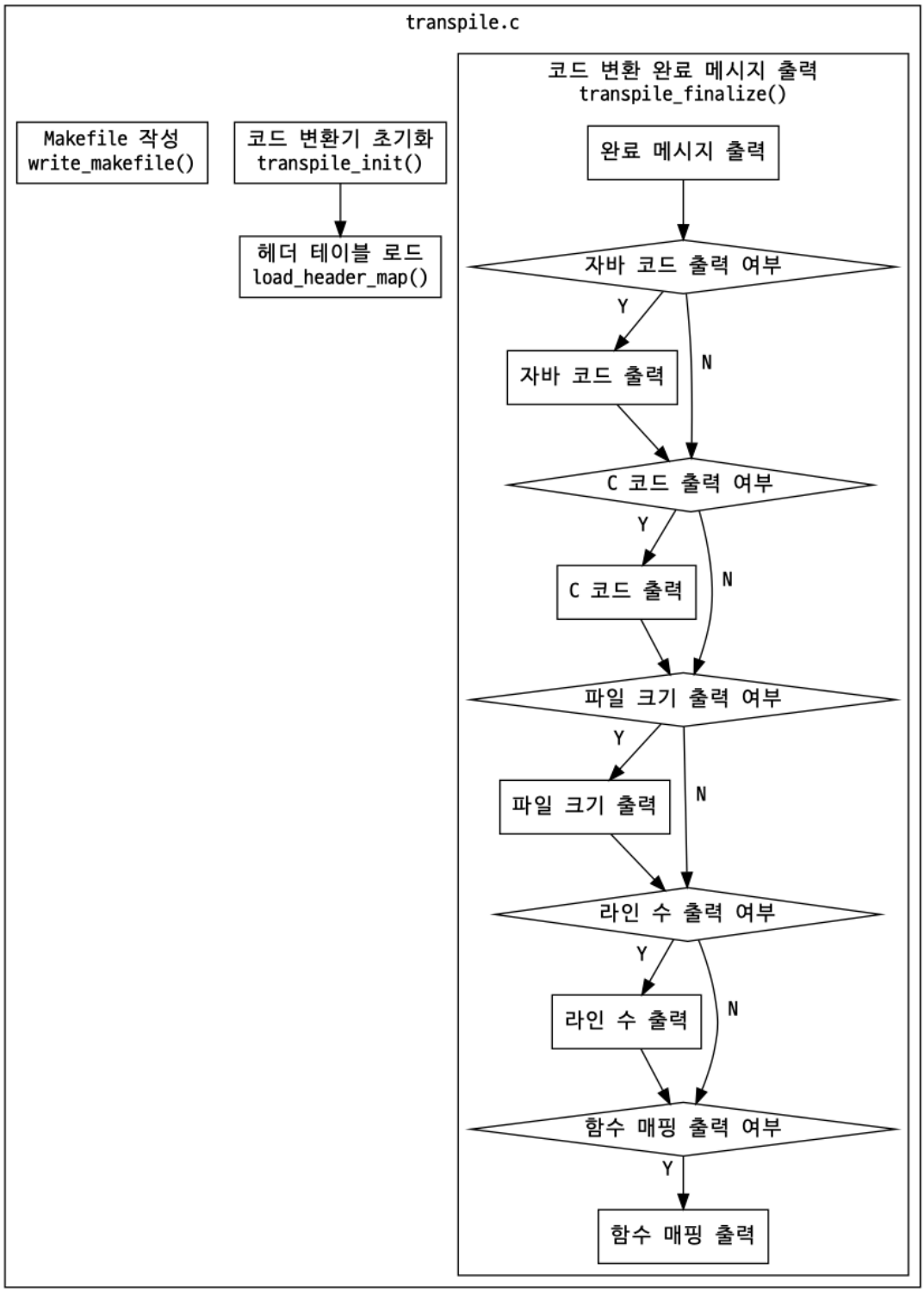


ssu_convert의 transpile.c



ssu_convert의 토크나이징 과정

위 두 사진에서는 transpile.c에서 호출된 transpile()함수와, transpile() 함수 내부에서 토크나이징이 이루어지는 과정을 보여주고 있다.



ssu_convert의 transpile.c

위 사진에서는 transpile.c의 나머지 부분의 흐름에 대하여 설명하고 있다.

3. 구현

<transpile.h>

```
/*
 * 코드 변환기 관련 함수
 */
// 코드 변환기를 초기화 하는 함수
void transpile_init();
// 코드를 한 줄 변환하는 함수
source_code *transpile(const char *file_name, const char *output_path,
                       const char *line, pid_t pid);
// 코드 변환기를 종료하고 필요한 메시지를 출력하는 함수
void transpile_finalize(const char *output_path, const char *file_name,
                        bool print_java, bool print_c, bool print_size,
                        bool print_line_num, bool print_func_map);
/*
 * 코드 출력 관련 함수
 */
// 소스 코드를 쓰는 함수
void write_source_code(source_code *code, FILE *fp);
// 소스 코드의 헤더 파일을 쓰는 함수
void write_source_header(source_code *code, FILE *fp);
// 'Makefile'을 쓰는 함수
void write_makefile(const char *output_path, const char *file_name);

/*
 * 기타 함수
 */
// 헤더 테이블을 불러오는 함수
header_map_node *load_header_map();
// 함수에 필요한 헤더를 인클루드하는 함수
void add_func_header(const char *func_name);
// 헤더 파일이 존재하지 않으면 추가하는 함수
void add_include(const char *include_str);
// 사용자 정의 헤더 파일을 인클루드 하는 함수
void add_user_include(const char *include_str);
// JAVA 함수 - C 함수의 매핑을 추가하는 함수
void add_func_map(const char *java_func, const char *c_func);
```

<util.h>

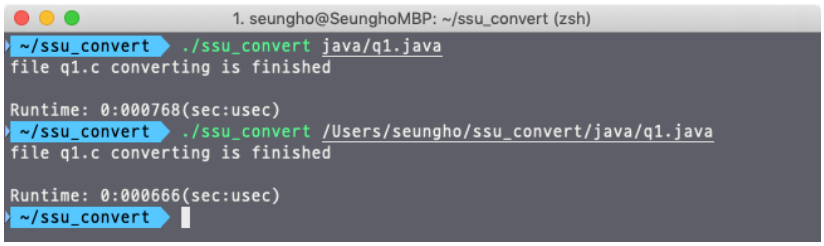
```
/*
```

```
 * 출력 관련 함수
 */
// 에러 메시지를 출력하고 종료한다.
void print_fatal_error(const char *format, ...);
// 코드를 출력한다.
void print_code(const char *title, const char *code, int line_limit);
// 프로그램의 실행 시간을 출력한다.
void ssu_runtime(struct timeval *begin_t, struct timeval *end_t);

/*
 * 파일 관련 함수
 */
// 파일이 존재하는지 확인한다.
bool file_exists(const char *pathname);
// 파일을 오픈하여 읽어들인다.
char *file_get_data(const char *pathname);
// 파일의 라인 수를 가져온다.
int file_get_line(const char *pathname);
// 파일의 크기를 가져온다.
off_t file_get_size(const char *pathname);
// 코드 파일을 읽어들여서 출력한다.
void file_print_code(const char *title, const char *pathname);

/*
 * 문자열 관련 함수
 */
// 문자열 좌우의 공백을 제거하여 반환한다.
char *string_trim(const char *str);
// 문자열이 특정 문자열로 끝나는지 확인한다.
bool string_ends_with(const char *str, const char *suffix);
```

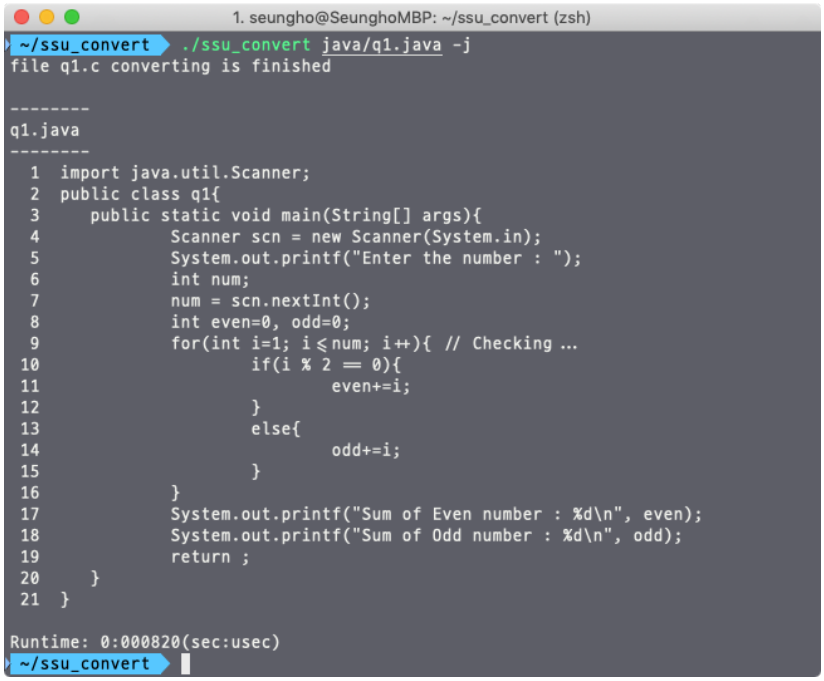
프로그램이 시작되면 main 함수에서는 가장 먼저 명령행 인수를 처리한다.
명령행 인수의 처리가 끝나게 되면 프로그램은 변환할 자바 코드 파일을 오픈하고, transpile_init() 함수를 통해 코드 변환기를 초기화한다. 이 때 헤더 테이블을 읽어들여 C 함수에 대응하는 헤더 파일의 정보를 가져온다.
그 뒤 strtok_r() 함수를 이용해 라인별로 코드 파일을 분할하고 transpile()함수를 호출하여 한줄씩 프로그램을 변환하게 된다.
transpile() 함수에서는 가장 먼저 라인의 어휘를 분석하여 토큰화한다. 토큰화 된 이후에는 토큰을 한번 먼저 순회하여 두 글자로 이루어진 연산자(예를 들면 '++'나 '==' 같은)를 하나로 합친다. 그 이후에는 여러 규칙에 맞게 토큰을 다른 토큰으로 분류한다. 파일을 쓰는 과정은 클래스가 끝날 때 쓰게 되는데, 이를 통해 여러 클래스로 이루어진 파일은 여러 파일로 나누어 변환할 수 있게 된다. 이 나누어진 파일들은 나중에 생성되는 Makefile을 통해 컴파일 할 수 있게 된다.
'-r' 옵션을 통해서는 파일의 변환 과정을 볼 수 있는데, main 함수에서 fork()가 일어나 프로세스가 하나 더 실행되며, 자식 프로세스에서는 파일 쓰기가 일어나지 않고 변환과정 출력 - 1초 대기 - 화면 클리어의 과정이 순서대로 일어나게 된다. 이후 부모 프로세스는 먼저 파일의 변환을 수행한 뒤 wait() 함수를 통해 자식 프로세스가 끝날 때 까지 대기하게 되며, 변환 과정을 모두 출력한 자식 프로세스는 exit() 함수를 통해 종료된다.

변환이 모두 종료되면 transpile_finalize() 함수가 호출되게 되는데, 앞에서 명령행 인수에서 받은 옵션을 반영해 파일이 변환되고 난 뒤 출력되어야 하는 여러 정보를 출력하게 된다.
이후 프로그램은 실행 시간을 출력한 뒤 종료된다.

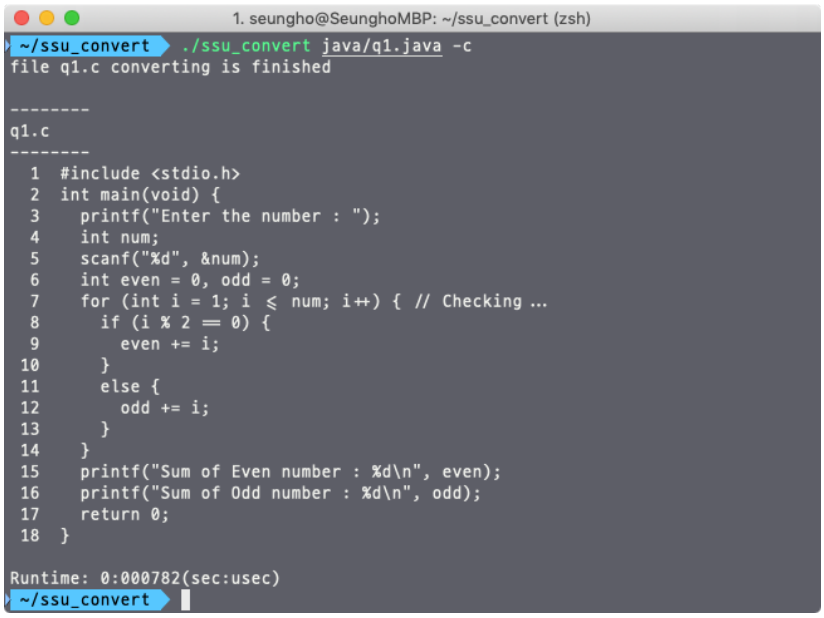
4. 테스트 및 결과



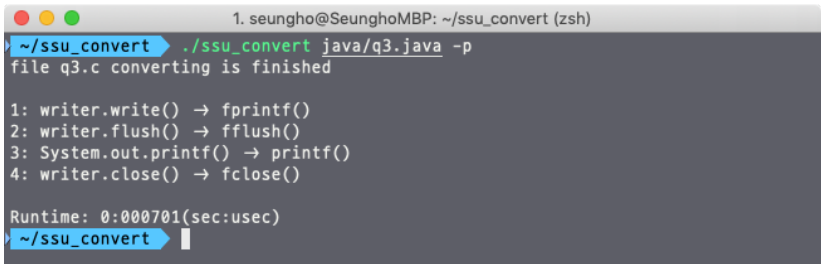
상대 경로와 절대 경로를 통해 파일을 변환한 모습



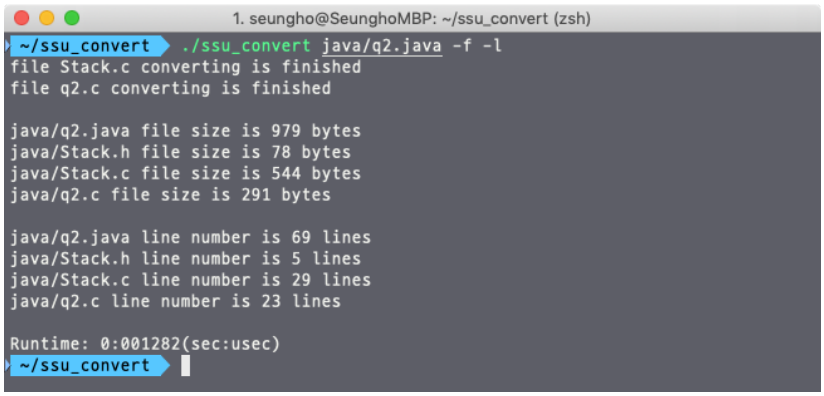
변환할 자바 파일을 출력한 모습



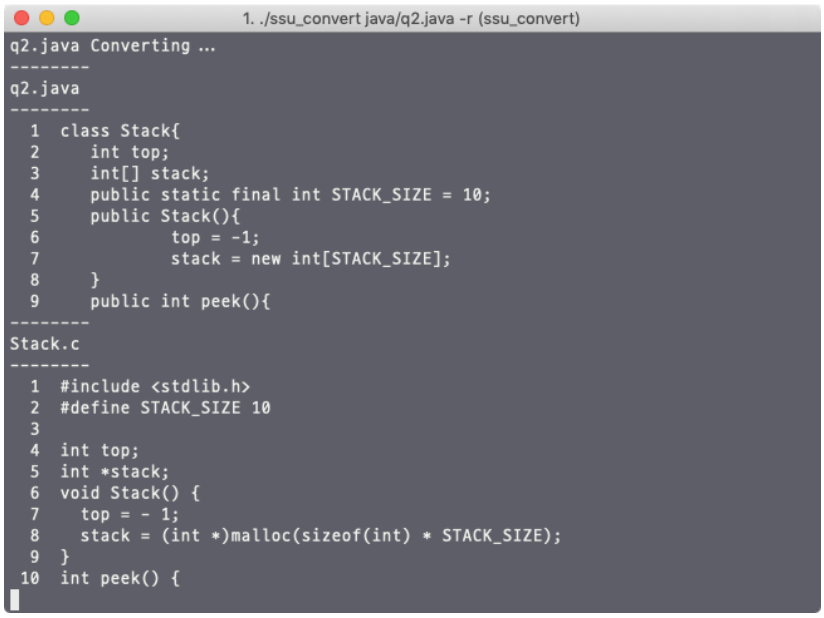
변환된 C 파일을 출력한 모습



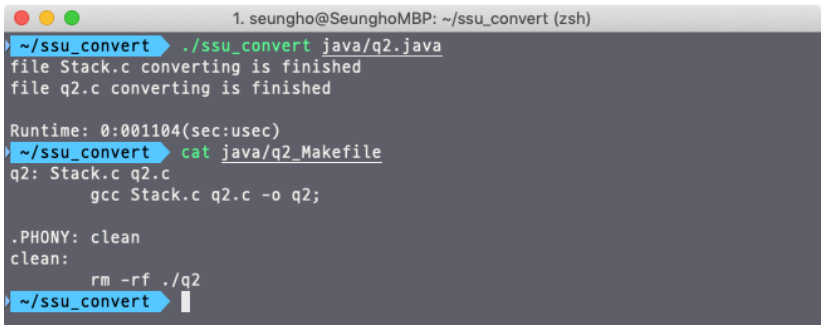
-p 옵션을 통해 변환 된 함수를 표시한 모습



-f -l 옵션을 동시에 사용하여 파일의 크기와 라인 수를 동시에 표시한 모습



-r 옵션을 사용하여 변환 과정을 출력하고 있는 모습



만들어진 Makefile의 모습

5. 소스 코드

<main.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <sys/wait.h>
#include <sys/time.h>
#include "util.h"
#include "transpile.h"
```

```
// flag_j: 변환할 java 파일을 출력할지 여부
// flag_c: 변환된 c 파일을 출력할지 여부
// flag_p: java 함수에 대응되는 c 함수를 출력할지 여부
// flag_f: 프로그램 파일의 크기를 출력할지 여부
// flag_l: 프로그램 파일의 라인 수를 출력할지 여부
// flag_r: 코드가 변환되는 과정을 출력할지 여부
bool flag_j = false, flag_c = false, flag_p = false, flag_f = false,
      flag_l = false, flag_r = false;

// 명령행 인수를 처리하는 함수
void process_args(int argc, char **argv);

int main(int argc, char *argv[])
{
  struct timeval begin_t, end_t;
  gettimeofday(&begin_t, NULL);

  // 명령행 인수 처리
  process_args(argc, argv);

  // pathname: 파일의 전체 경로
  const char *pathname = argv[1];
  // file_name: 경로와 확장자를 제외한 파일의 순수한 이름
  char file_name[NAME_MAX];
  // output_path: C 파일이 출력될 경로
  char output_path[LINE_MAX];

  // pathname에서 file_name과 output_path를 구한다.
  if (strrchr(argv[1], '/') != NULL) {
    // pathname이 경로를 포함하고 있는 경우
    strncpy(output_path, pathname, (strrchr(argv[1], '/') - pathname) + 1);
    output_path[(strrchr(argv[1], '/') - pathname) + 1] = '\0';
    strcpy(file_name, strrchr(argv[1], '/') + 1);
  }
  else {
    // pathname이 경로를 포함하고 있지 않은 경우
    strcpy(output_path, "");
    strcpy(file_name, pathname);
  }

  // file_name이 확장자를 포함하는 경우 제거한다.
  if (strrchr(file_name, '.') != NULL) {
    *strrchr(file_name, '.') = '\0';
  }
```

```
// 자바 파일을 읽어들인다.
char *file_src = file_get_data(pathname);

// 자바 파일이 존재하지 않는 경우
if (file_src == NULL) {
  // 에러 메시지를 출력하고 종료한다.
  print_fatal_error("file '%s' is not exist.", pathname);
}
// 원본 코드 출력에 사용하기 위한 자바 파일을 복제한다.
char *file_src_dup = strdup(file_src);
// 출력할 라인 수
int line_num = 1;
// 현재 pid를 구한다.
pid_t pid = getpid();

char *save_ptr;
// strtok_r을 이용하여 line 변수에 자바 파일의 첫번째 줄을 가져온다.
char *line = strtok_r(file_src, "\n", &save_ptr);

// 코드 변환기를 초기화한다.
transpile_init();

// '-r'옵션이 지정되어 있는 경우
if (flag_r) {
  // 자식 프로세스를 생성한다.
  pid = fork();
  if (pid < 0) {
    // 에러가 발생하면 에러 메시지를 출력하고 종료한다.
    print_fatal_error("fork error");
  }
}

// 줄의 끝까지 도달할 때까지 반복한다.
while (line != NULL) {
  // line을 변환한다.
  source_code *transpiled_source_code =
      transpile(file_name, output_path, line, pid);
  if (pid == 0) {
    // '-r' 옵션을 받아 fork로 생성된 자식 프로세스의 경우 변환 과정을
    // 출력한다.
    // '변환중...' 메시지 출력
    char java_file[FILENAME_MAX];
    sprintf(java_file, "%s.java", file_name);
```

```
      printf("%s Converting...\n", java_file);
      // 변환될 자바 코드 출력
      print_code(java_file, file_src_dup, line_num++);
      // 변환된 C 코드 출력
      write_source_code(transpiled_source_code, stdout);
      // 변환 과정을 보여주기 위해 1초 대기
      sleep(1);
      // 화면 클리어
      system("clear");
    }
    // line에 다음 줄의 시작 위치를 저장한다.
    line = strtok_r(NULL, "\n", &save_ptr);
}

// 'Makefile'을 생성한다.
write_makefile(output_path, file_name);

// '-r' 옵션이 지정된 경우
if (flag_r) {
    // 자식 프로세스인경우 종료한다.
    if (pid == 0) {
      exit(0);
    }

    int status;
    // 부모 프로세스는 자식 프로세스의 종료를 대기한다.
    if (wait(&status) != pid) {
      print_fatal_error("wait error");
    }
}

// 자바 파일을 메모리에서 해제한다.
free(file_src);
free(file_src_dup);

// 코드 변환기를 종료하고, 필요에 따라 메시지를 출력한다.
transpile_finalize(output_path, file_name, flag_j, flag_c, flag_f, flag_l,
                   flag_p);

// 프로그램의 수행 시간을 출력한다.
gettimeofday(&end_t, NULL);
ssu_runtime(&begin_t, &end_t);

exit(0);
```

```
}

// 명령행 인수를 처리하는 함수
void process_args(int argc, char **argv)
{
  // 인수의 수가 너무 적으면 에러를 출력하고 프로그램을 종료한다.
  if (argc < 2) {
    fprintf(stderr, "usage: %s <FILENAME> [option]\n", argv[0]);
    exit(1);
  }

  for (int i = 2; i < argc; ++i) {
    // 명령행 인수에 따라 플래그를 설정한다.
    if (strcmp(argv[i], "-j") == 0) {
      flag_j = true;
    }
    else if (strcmp(argv[i], "-c") == 0) {
      flag_c = true;
    }
    else if (strcmp(argv[i], "-p") == 0) {
      flag_p = true;
    }
    else if (strcmp(argv[i], "-f") == 0) {
      flag_f = true;
    }
    else if (strcmp(argv[i], "-l") == 0) {
      flag_l = true;
    }
    else if (strcmp(argv[i], "-r") == 0) {
      flag_r = true;
    }
    else {
      // 올바르지 않은 명령행 인수가 들어오면 오류를 출력한다.
      print_fatal_error("unknown option '%s'\n"
                        "usage: %s <FILENAME> [option]",
                        argv[i], argv[0]);
    }
  }
}
```

<transpile.h>

```
#ifndef SSU_CONVERT_TRANSPILE_H
```

```
#define SSU_CONVERT_TRANSPILE_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>
#include <limits.h>
#include "util.h"

// 한 토큰의 길이
#define TOKEN_LEN 256
// 한 라인의 길이
#define LINE_LEN 1024
// 변환할 수 있는 클래스의 최댓값
#define CLASS_LIMIT 16
// 변환할 수 있는 함수의 종류의 최댓값
#define FUNC_LIMIT 128

//토큰의 종류
typedef enum _TOK_TYPE {
  // 연산자
  TOK_OPERATOR,
  // 숫자
  TOK_NUMBER,
  // 식별자
  TOK_IDENTIFIER,
  // 문자열
  TOK_STRING,
  // 주석
  TOK_COMMENT
} TOK_TYPE;

// 토큰의 종류와 내용을 저장하는 구조체
typedef struct _token_type {
  char content[TOKEN_LEN];
  TOK_TYPE type;
} token_type;

// 라인을 저장하는 링크드 리스트
typedef struct _line_node {
  struct _line_node *next;
  char line[LINE_LEN];
} line_node;
```

```
// 소스 코드를 저장하는 구조체
typedef struct _source_code {
  // 표준 헤더
  line_node *header;
  // 사용자 정의 클래스의 헤더
  line_node *user_header;
  // '.h' 파일을 저장하는 링크드 리스트
  line_node *header_file;
  // #define을 저장하는 링크드 리스트
  line_node *define;
  // 코드 부분을 저장하는 링크드 리스트
  line_node *body;
} source_code;

// 헤더 테이블에서 헤더 파일의 이름을 저장하는 데 사용되는 링크드 리스트
typedef struct _header_node {
  struct _header_node *next;
  char header[TOKEN_LEN];
} header_node;

// 헤더 테이블에서 함수 이름과 헤더 노드를 저장하는 데 사용되는 링크드 리스트
typedef struct _header_map_node {
  struct _header_map_node *next;
  header_node *header;
  char func[TOKEN_LEN];
} header_map_node;

// 헤더 테이블의 헤드를 저장하는 구조체
typedef struct _header_map {
  header_map_node *head;
} header_map;

// JAVA 함수 - C 함수의 매핑을 저장하는 구조체
typedef struct _func_map {
  char c[TOKEN_LEN];
  char java[TOKEN_LEN];
} func_map;

// '{' '}'의 깊이
int brace_level;
// 현재 클래스
char class_context[TOKEN_LEN];
// 사용자 정의 객체의 리스트
```

```
char object_list[CLASS_LIMIT][TOKEN_LEN];
// 사용자 정의 객체의 수
int object_list_count;
// 생성된 파일의 리스트
char file_list[CLASS_LIMIT][TOKEN_LEN];
// 생성된 파일의 수
int file_list_count;
// 매핑된 함수의 리스트
func_map func_list[FUNC_LIMIT];
// 매핑된 함수의 수
int func_list_count;
// 현재 문자열의 위치가 주석인지 여부
bool is_comment;
// 다음 행을 한단계 더 들여쓰기할지 여부
bool indent_next;
// 다음 토큰을 건너뛸지의 여부
bool skip_next;
// '.h' 파일의 프로토 타입을 생성 할 때 사용되는 변수
// ')'을 기다리고 있는지 여부
bool is_waiting_paren;
// 현재 문자열의 위치가 메인 함수인지 여부
bool is_main_func;
// 변환 된 소스 파일
source_code *source_file;
// 헤더 테이블
header_map header_table;

/*
 * 코드 변환기 관련 함수
 */
// 코드 변환기를 초기화 하는 함수
void transpile_init();
// 코드를 한 줄 변환하는 함수
source_code *transpile(const char *file_name, const char *output_path,
                       const char *line, pid_t pid);
// 코드 변환기를 종료하고 필요한 메시지를 출력하는 함수
void transpile_finalize(const char *output_path, const char *file_name,
                        bool print_java, bool print_c, bool print_size,
                        bool print_line_num, bool print_func_map);
/*
 * 코드 출력 관련 함수
 */
// 소스 코드를 쓰는 함수
void write_source_code(source_code *code, FILE *fp);
```

```
// 소스 코드의 헤더 파일을 쓰는 함수
void write_source_header(source_code *code, FILE *fp);
// 'Makefile'을 쓰는 함수
void write_makefile(const char *output_path, const char *file_name);

/*
 * 기타 함수
 */
// 헤더 테이블을 불러오는 함수
header_map_node *load_header_map();
// 함수에 필요한 헤더를 인클루드하는 함수
void add_func_header(const char *func_name);
// 헤더 파일이 존재하지 않으면 추가하는 함수
void add_include(const char *include_str);
// 사용자 정의 헤더 파일을 인클루드 하는 함수
void add_user_include(const char *include_str);
// JAVA 함수 - C 함수의 매핑을 추가하는 함수
void add_func_map(const char *java_func, const char *c_func);

#endif
```

<transpile.c>

```
#include "transpile.h"

/*
 * 코드 변환기 관련 함수
 */

// 코드 변환기를 초기화 하는 함수
void transpile_init()
{
  brace_level = 0;
  class_context[0] = 0;
  file_list_count = 0;
  func_list_count = 0;
  is_comment = false;
  indent_next = false;
  skip_next = false;
  is_waiting_paren = false;
  is_main_func = false;
  source_file = NULL;
  header_table.head = load_header_map();
```

```
}

// 코드를 한 줄 변환하는 함수
source_code *transpile(const char *file_name, const char *output_path,
                       const char *line, pid_t pid)
{
  // 문자열 좌우 공백 제거
  char *line_trim = string_trim(line);
  int line_trim_len = strlen(line_trim);

  // 토큰 변수 초기화
  token_type token[TOKEN_LEN];
  memset(&token, 0, sizeof(token));
  int token_count = 0;

  // 들여쓰기 레벨을 설정한다.
  int indent_level = brace_level;

  /*
   * 토큰 분리
   */
  for (int i = 0; i < line_trim_len;) {
    // 주석인 경우
    if (is_comment) {
      // '/* */' 주석의 끝을 만나지 못한 경우 끝을 만날때까지 추가
      int count = 0;
      while (i + count < line_trim_len &&
             strncmp(line_trim + i + count, "*/", 2) != 0) {
        count++;
      }
      strncat(token[token_count].content, line_trim + i, count);
      token[token_count].type = TOK_COMMENT;
      i += count;
      if (strncmp(line_trim + i, "*/", 2) == 0) {
        strcat(token[token_count].content, "*/");
        token[token_count].type = TOK_COMMENT;
        i += 2;
        is_comment = false;
      }
      ++token_count;
    }
    else if (strncmp(line_trim + i, "/*", 2) == 0) {
      // '/* */' 주석의 시작인 경우
      strcpy(token[token_count].content, "/*");
```

```
  token[token_count].type = TOK_COMMENT;
  is_comment = true;
  i += 2;
}
else if (strncmp(line_trim + i, "//", 2) == 0) {
  // '//' 주석인 경우
  strcpy(token[token_count].content, line_trim + i);
  token[token_count].type = TOK_COMMENT;
  i = line_trim_len - 1;
  ++token_count;
}
else if (line_trim[i] == '"') {
  // 문자열을 만난 경우
  int end = i + 1;
  bool is_escaped = false;
  while (!(!is_escaped && line_trim[end] == '"')) {
    if (is_escaped) {
      is_escaped = false;
    }
    else if (line_trim[end] == '\\') {
      is_escaped = true;
    }
    ++end;
  }
  strncpy(token[token_count].content, line_trim + i, end - i + 1);
  token[token_count].content[end - i + 1] = '\0';
  token[token_count].type = TOK_STRING;
  i = end;
  ++token_count;
}
else if (isalpha(line_trim[i]) || line_trim[i] == '_') {
  // 식별자를 만난 경우
  int end = i + 1;
  while (isalnum(line_trim[end]) || line_trim[end] == '_') {
    ++end;
  }
  strncpy(token[token_count].content, line_trim + i, end - i);
  token[token_count].content[end - i] = '\0';
  token[token_count].type = TOK_IDENTIFIER;
  i = end - 1;
  ++token_count;
}
else if (isdigit(line_trim[i])) {
  // 숫자를 만난 경우
```

```
    int end = i + 1;
    while (isdigit(line_trim[end])) {
      ++end;
    }
    strncpy(token[token_count].content, line_trim + i, end - i);
    token[token_count].content[end - i] = '\0';
    token[token_count].type = TOK_NUMBER;
    i = end - 1;
    ++token_count;
  }
  else if (!isspace(line_trim[i])) {
    // 연산자를 만난 경우
    token[token_count].content[0] = line_trim[i];
    token[token_count].content[1] = 0;
    token[token_count].type = TOK_OPERATOR;
    // 들여쓰기 레벨 설정
    if (line_trim[i] == '{')
      brace_level++;
    else if (line_trim[i] == '}') {
      brace_level--;
      indent_level--;
    }
    ++token_count;
  }
  ++i;
}

/*
 * 토큰 전처리
 */
int i, j;
// 여러 토큰으로 이루어진 연산자를 합친다.
for (i = 0, j = 0; i < token_count; ++i, ++j) {
  char *repeats[] = {"&", "|", "+", "-", ">", "<", "="};
  char *equals[] = {"+", "-",  "*",  "/", "%", "!", "<",
                    ">", "<<", ">>", "^", "|", "~"};
  bool matched = false;
  for (int k = 0; k < 7; ++k) {
    char *sym = repeats[k];
    if (j > 0 && strcmp(token[i].content, sym) == 0 &&
        strcmp(token[j - 1].content, sym) == 0) {
      matched = true;
      --j;
      strcat(token[j].content, sym);
```

```
      break;
    }
  }
  if (matched)
    continue;
  for (int k = 0; k < 13; ++k) {
    char *sym = equals[k];
    if (j > 0 && strcmp(token[i].content, "=") == 0 &&
        strcmp(token[j - 1].content, sym) == 0) {
      matched = true;
      --j;
      strcat(token[j].content, "=");
      break;
    }
  }
  if (matched)
    continue;
  if (j > 0 && strcmp(token[i].content, ".") == 0 &&
      token[j - 1].type == TOK_IDENTIFIER) {
    bool object_matched = false;
    for (int k = 0; k < object_list_count; ++k) {
      if (strcmp(object_list[k], token[j - 1].content) == 0) {
        object_matched = true;
        break;
      }
    }
    if (object_matched) {
      j -= 2;
    }
    else {
      --j;
      strcat(token[j].content, ".");
    }
    continue;
  }
  if (j > 0 && token[i].type == TOK_IDENTIFIER &&
      string_ends_with(token[j - 1].content, ".") &&
      token[j - 1].type == TOK_IDENTIFIER) {
    --j;
    strcat(token[j].content, token[i].content);
    continue;
  }
  if (i > 0 && strcmp(token[i].content, ">") == 0 &&
      strcmp(token[j - 1].content, "-") == 0) {
```

```
      --j;
      strcat(token[j].content, ">");
    }
    else {
      if (i != j) {
        strcpy(token[j].content, token[i].content);
        token[j].type = token[i].type;
      }
    }
  }
  token_count = j;

  // 문자열의 길이가 0인 경우 변환을 마친다.
  if (strlen(line_trim) == 0)
    return source_file;

  /*
   * 토큰 변환
   */
  token_type token_result[TOKEN_LEN];
  memset(&token_result, 0, sizeof(token_result));
  int token_result_count = 0;

  // 토큰을 규칙에 따라 변환한다.
  for (i = 0, j = 0; i < token_count; ++i, ++j) {
    // 다음 토큰을 생략할 필요가 있는 경우 생략한다.
    if (skip_next) {
      --j;
      skip_next = false;
      continue;
    }

    // 다음과 같은 행으로 시작하는 문자열은 무시한다.
    if (strcmp(token[i].content, "import") == 0 ||
        strcmp(token[i].content, "Scanner") == 0) {
      break;
    }

    // 지원하지 않거나 불필요한 토큰은 제거한다.
    if (strcmp(token[i].content, "public") == 0 ||
        strcmp(token[i].content, "private") == 0 ||
        strcmp(token[i].content, "default") == 0 ||
        strcmp(token[i].content, "protected") == 0 ||
        strcmp(token[i].content, "static") == 0 ||
```

```
      strcmp(token[i].content, "throws") == 0 ||
      strcmp(token[i].content, "IOException") == 0) {
    --j;
    continue;
}

// 메인 함수 변환
if (strcmp(token[i].content, "main") == 0 &&
      strcmp(token[0].content, "public") == 0) {
    strcpy(token_result[0].content, "int");
    token_result[0].type = TOK_IDENTIFIER;
    strcpy(token_result[1].content, "main");
    token_result[1].type = TOK_IDENTIFIER;
    strcpy(token_result[2].content, "(");
    token_result[2].type = TOK_OPERATOR;
    strcpy(token_result[3].content, "void");
    token_result[3].type = TOK_IDENTIFIER;
    strcpy(token_result[4].content, ")");
    token_result[4].type = TOK_OPERATOR;
    j = 4;
    while (strcmp(token[i].content, ")") != 0) {
      ++i;
    }
    is_main_func = true;
    continue;
}

// 새로운 클래스가 시작되는 경우
if (i > 0 && strcmp(token[i - 1].content, "class") == 0 &&
      token[i].type == TOK_IDENTIFIER) {
    // 클래스 컨텍스트를 변환한다.
    // 파일 저장은 클래스 컨텍스트 단위로 저장된다.
    strcpy(class_context, token[i].content);
    // 파일 리스트에 추가
    // Makefile을 만들때 사용된다.
    sprintf(file_list[file_list_count], "%s.c", class_context);
    ++file_list_count;
    // 소스 파일 초기화
    source_file = (source_code *)malloc(sizeof(source_code));
    source_file->header = (line_node *)malloc(sizeof(line_node));
    source_file->header->next = NULL;
    source_file->user_header = (line_node *)malloc(sizeof(line_node));
    source_file->user_header->next = NULL;
    source_file->define = (line_node *)malloc(sizeof(line_node));
```

```
    source_file->define->next = NULL;
    source_file->body = (line_node *)malloc(sizeof(line_node));
    source_file->body->next = NULL;
    source_file->header_file = (line_node *)malloc(sizeof(line_node));
    source_file->header_file->next = NULL;
    skip_next = true;
    j = -1;
    memset(object_list, 0, sizeof(object_list));
    object_list_count = 0;
    continue;
}

// 함수가 끝나는 경우
if (strcmp(token[i].content, "}") == 0 && indent_level == 1) {
    is_main_func = false;
}

// 클래스 파일이 끝나는경우
if (strcmp(token[i].content, "}") == 0 && indent_level == 0) {
    // 메인 함수를 포함한 클래스가 아닌 경우
    // 헤더 파일을 생성한다.
    if (strcmp(class_context, file_name) != 0) {
        char class_name[TOKEN_LEN];
        sprintf(class_name, "\"%s.h\"", class_context);
        add_user_include(class_name);
    }

    // 자식 프로세스가 아닌 경우 실제로 파일을 생성한다.
    if (pid > 0) {
        char output_file_name[PATH_MAX];
        strcpy(output_file_name, output_path);
        strcat(output_file_name, class_context);
        strcat(output_file_name, ".c");

        FILE *fp;

        if ((fp = fopen(output_file_name, "w")) == NULL) {
            print_fatal_error("file open error for '%s'", output_file_name);
        }
        // '.c' 파일 기록
        write_source_code(source_file, fp);
        fclose(fp);

        if (strcmp(class_context, file_name) != 0) {
```

```
      output_file_name[strlen(output_file_name) - 1] = 'h';
      if ((fp = fopen(output_file_name, "w")) == NULL) {
        print_fatal_error("file open error for '%s'", output_file_name);
      }
      // 필요한 경우 '.h' 파일 기록
      write_source_header(source_file, fp);
      fclose(fp);
    }
}

// 할당된 메모리 회수

class_context[0] = 0;

line_node *header = source_file->header;
while (header != NULL) {
  line_node *tmp = header->next;
  free(header);
  header = tmp;
}

line_node *user_header = source_file->user_header;
while (user_header != NULL) {
  line_node *tmp = user_header->next;
  free(user_header);
  user_header = tmp;
}

line_node *header_file = source_file->header_file;
while (header_file != NULL) {
  line_node *tmp = header_file->next;
  free(header_file);
  header_file = tmp;
}

line_node *define = source_file->define;
while (define != NULL) {
  line_node *tmp = define->next;
  free(define);
  define = tmp;
}

line_node *body = source_file->body;
while (body != NULL) {
```

```
      line_node *tmp = body->next;
      free(body);
      body = tmp;
    }

    free(source_file);
    source_file = NULL;

    --j;
    break;
}

// System.out.printf -> printf 변환
if (strcmp(token[i].content, "System.out.printf") == 0) {
    strcpy(token_result[j].content, "printf");
    add_func_map("System.out.printf", "printf");
    add_func_header("printf");
    token_result[j].type = token[i].type;
    continue;
}

// final 변수의 경우 #define 처리
if (i > 4 && strcmp(token[i - 5].content, "final") == 0 &&
      token[i - 4].type == TOK_IDENTIFIER &&
      token[i - 3].type == TOK_IDENTIFIER &&
      strcmp(token[i - 2].content, "=") == 0 &&
      strcmp(token[i].content, ";") == 0) {
    line_node *new_define_line = malloc(sizeof(line_node));
    new_define_line->next = NULL;
    sprintf(new_define_line->line, "#define %s %s", token[i - 3].content,
            token[i - 1].content);
    line_node *cur = source_file->define;
    while (cur->next != NULL) {
      cur = cur->next;
    }
    cur->next = new_define_line;
    j -= 6;
    continue;
}

// null -> NULL 변환
if (strcmp(token[i].content, "null") == 0) {
    strcpy(token_result[j].content, "NULL");
    token_result[j].type = token[i].type;
```

```
    continue;
}

// 파일 이름 타입 변환
if (i > 0 && strcmp(token[i - 1].content, "File") == 0 &&
      token[i].type == TOK_IDENTIFIER) {
    strcpy(token_result[j - 1].content, "char");
    strcpy(token_result[j].content, "*");
    strcat(token_result[j].content, token[i].content);
    token[j].type = TOK_IDENTIFIER;
    continue;
}

// 파일 이름 타입 변환
if (i > 4 && strcmp(token[i - 4].content, "new") == 0 &&
      strcmp(token[i - 3].content, "File") == 0 &&
      strcmp(token[i - 2].content, "(") == 0 &&
      token[i - 1].type == TOK_STRING && strcmp(token[i].content, ")") == 0) {
    j -= 4;
    strcpy(token_result[j].content, token[i - 1].content);
    token[j].type = TOK_STRING;
    continue;
}

// FileWriter -> fopen 변환
if (i > 0 && strcmp(token[i - 1].content, "FileWriter") == 0 &&
      token[i].type == TOK_IDENTIFIER) {
    strcpy(token_result[j - 1].content, "FILE");
    strcpy(token_result[j].content, "*");
    strcat(token_result[j].content, token[i].content);
    token[j].type = TOK_IDENTIFIER;
    continue;
}

// FileWriter -> fopen 변환
if (i > 8 && strcmp(token[i - 7].content, "new") == 0 &&
      strcmp(token[i - 6].content, "FileWriter") == 0 &&
      strcmp(token[i - 5].content, "(") == 0 &&
      strcmp(token[i - 3].content, ",") == 0 &&
      token[i - 2].type == TOK_IDENTIFIER &&
      strcmp(token[i - 1].content, ")") == 0 &&
      strcmp(token[i].content, ";") == 0) {
    j -= 7;
    strcpy(token_result[j].content, "fopen");
```

```
    token[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, "(");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, token[i - 4].content);
    token[j].type = token[i - 4].type;
    strcpy(token_result[++j].content, ",");
    token[j].type = TOK_OPERATOR;
    char *mode = strcmp(token[i - 2].content, "false") ? "\"a\"" : "\"w\"";
    strcpy(token_result[++j].content, mode);
    token[j].type = TOK_STRING;
    strcpy(token_result[++j].content, ")");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, ";");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, "if");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, "(");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, token[i - 9].content);
    token[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, "==");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, "NULL");
    token[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, ")");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, "exit");
    token[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, "(");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, "1");
    token[j].type = TOK_NUMBER;
    strcpy(token_result[++j].content, ")");
    token[j].type = TOK_OPERATOR;
    strcpy(token_result[++j].content, ";");
    token[j].type = TOK_OPERATOR;
    // 에러 처리 추가
    add_func_header("exit");
    continue;
}

// .write -> fprintf 변환
if (i > 0 && string_ends_with(token[i - 1].content, ".write") &&
    strcmp(token[i].content, "(") == 0) {
```

```
    strcpy(token_result[j - 1].content, "fprintf");
    strcpy(token_result[j].content, "(");
    token_result[j].type = TOK_OPERATOR;
    strncpy(token_result[++j].content, token[i - 1].content,
            strlen(token[i - 1].content) - 6);
    token_result[j].content[strlen(token[i - 1].content) - 6] = '\0';
    token_result[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, ",");
    token_result[j].type = TOK_OPERATOR;
    add_func_map(token[i - 1].content, "fprintf");
    add_func_header("fprintf");
    continue;
}

// .flush -> fflush 변환
if (i > 0 && string_ends_with(token[i - 1].content, ".flush") &&
    strcmp(token[i].content, "(") == 0) {
    strcpy(token_result[j - 1].content, "fflush");
    strcpy(token_result[j].content, "(");
    token_result[j].type = TOK_OPERATOR;
    strncpy(token_result[++j].content, token[i - 1].content,
            strlen(token[i - 1].content) - 6);
    token_result[j].content[strlen(token[i - 1].content) - 6] = '\0';
    token_result[j].type = TOK_IDENTIFIER;
    add_func_map(token[i - 1].content, "fflush");
    add_func_header("fflush");
    continue;
}

// .close -> fclose 변환
if (i > 0 && string_ends_with(token[i - 1].content, ".close") &&
    strcmp(token[i].content, "(") == 0) {
    strcpy(token_result[j - 1].content, "fclose");
    strcpy(token_result[j].content, "(");
    token_result[j].type = TOK_OPERATOR;
    strncpy(token_result[++j].content, token[i - 1].content,
            strlen(token[i - 1].content) - 6);
    token_result[j].content[strlen(token[i - 1].content) - 6] = '\0';
    token_result[j].type = TOK_IDENTIFIER;
    add_func_map(token[i - 1].content, "fclose");
    add_func_header("fclose");
    continue;
}
```

```
// .nextInt -> scanf 변환
if (i > 0 && strcmp(token[i - 1].content, "=") == 0 &&
    string_ends_with(token[i].content, ".nextInt")) {
  add_func_map(token[i].content, "scanf");
  add_func_header("scanf");
  j = -1;
  if (strcmp(token[0].content, "int") == 0) {
    strcpy(token_result[++j].content, "int");
    token_result[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, token[i - 2].content);
    token_result[j].type = TOK_IDENTIFIER;
    strcpy(token_result[++j].content, ";");
    token_result[j].type = TOK_OPERATOR;
  }
  strcpy(token_result[++j].content, "scanf");
  token_result[j].type = TOK_IDENTIFIER;
  strcpy(token_result[++j].content, "(");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, "\"%d\"");
  token_result[j].type = TOK_STRING;
  strcpy(token_result[++j].content, ",");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, "&");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, token[i - 2].content);
  token_result[j].type = TOK_IDENTIFIER;
  strcpy(token_result[++j].content, ")");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, ";");
  token_result[j].type = TOK_OPERATOR;
  ++j;
  break;
}

// 배열 -> 포인터 변환
if (i > 2 && token[i - 3].type == TOK_IDENTIFIER &&
    strcmp(token[i - 2].content, "[") == 0 &&
    strcmp(token[i - 1].content, "]") == 0 &&
    token[i].type == TOK_IDENTIFIER) {
  j -= 2;
  strcpy(token_result[j].content, "*");
  strcat(token_result[j].content, token[i].content);
  token_result[j].type = TOK_IDENTIFIER;
  continue;
```

```
}

// 메인 함수 리턴 변환
if (i > 0 && strcmp(token[i - 1].content, "return") == 0 &&
    strcmp(token[i].content, ";") == 0 && is_main_func) {
  strcpy(token_result[j].content, "0");
  token_result[j].type = TOK_IDENTIFIER;
  strcpy(token_result[++j].content, ";");
  token_result[j].type = TOK_OPERATOR;
  continue;
}

// new [] -> malloc 변환
if (i > 3 && strcmp(token[i - 4].content, "new") == 0 &&
    token[i - 3].type == TOK_IDENTIFIER &&
    strcmp(token[i - 2].content, "[") == 0 &&
    token[i - 1].type == TOK_IDENTIFIER &&
    strcmp(token[i].content, "]") == 0) {
  j -= 4;
  strcpy(token_result[j].content, "(");
  strcat(token_result[j].content, token[i - 3].content);
  strcat(token_result[j].content, " *)");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, "malloc");
  token_result[j].type = TOK_IDENTIFIER;
  strcpy(token_result[++j].content, "(");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, "sizeof(");
  strcat(token_result[j].content, token[i - 3].content);
  strcat(token_result[j].content, ")");
  token_result[j].type = TOK_IDENTIFIER;
  strcpy(token_result[++j].content, "*");
  token_result[j].type = TOK_OPERATOR;
  strcpy(token_result[++j].content, token[i - 1].content);
  token_result[j].type = TOK_IDENTIFIER;
  strcpy(token_result[++j].content, ")");
  token_result[j].type = TOK_OPERATOR;
  add_func_header("malloc");
  continue;
}

// 클래스 생성자 변환
if (i > 2 && strcmp(token[i - 2].content, class_context) == 0 &&
    strcmp(token[i - 1].content, "(") == 0 &&
```

```
        strcmp(token[i].content, ")") == 0) {
      j -= 2;
      strcpy(token_result[j].content, "void");
      token_result[j].type = TOK_IDENTIFIER;
      strcpy(token_result[++j].content, token[i - 2].content);
      token_result[j].type = TOK_IDENTIFIER;
      strcpy(token_result[++j].content, "(");
      token_result[j].type = TOK_OPERATOR;
      strcpy(token_result[++j].content, ")");
      token_result[j].type = TOK_OPERATOR;
      continue;
    }

    // 사용자 정의 객체 생성 변환
    if (i > 5 && token[i - 6].type == TOK_IDENTIFIER &&
        token[i - 5].type == TOK_IDENTIFIER &&
        strcmp(token[i - 4].content, "=") == 0 &&
        strcmp(token[i - 3].content, "new") == 0 &&
        token[i - 2].type == TOK_IDENTIFIER &&
        strcmp(token[i - 1].content, "(") == 0 &&
        strcmp(token[i].content, ")") == 0 &&
        strcmp(token[i - 6].content, token[i - 2].content) == 0) {
      j -= 6;
      strcpy(token_result[++j].content, "(");
      token_result[j].type = TOK_OPERATOR;
      strcpy(token_result[++j].content, ")");
      token_result[j].type = TOK_OPERATOR;
      strcpy(object_list[object_list_count], token[i - 5].content);
      ++object_list_count;
      char class_name[TOKEN_LEN];
      sprintf(class_name, "\"%s.h\"", token[i - 6].content);
      add_user_include(class_name);
      continue;
    }
    strcpy(token_result[j].content, token[i].content);
    token_result[j].type = token[i].type;
  }

  token_result_count = j;

  // 토큰 길이가 0보다 크면 새 라인 노드를 생성한다.
  if (token_result_count > 0) {
    line_node *new_line = (line_node *)malloc(sizeof(line_node));
    new_line->next = NULL;
```

```
new_line->line[0] = 0;

// 인덴트 구현
for (i = 0; i < indent_level - 1; ++i) {
  strcat(new_line->line, "    ");
}
if (indent_next) {
  strcat(new_line->line, "    ");
  indent_next = false;
}

// 토큰 간 띄어쓰기 구현
for (i = 0; i < token_result_count; ++i) {
  if (i > 0) {
    if (strcmp(token_result[i].content, ";") != 0 &&
        strcmp(token_result[i].content, ",") != 0 &&
        strcmp(token_result[i].content, ")") != 0 &&
        !(string_ends_with(token_result[i - 1].content, ")") &&
          token_result[i].type != TOK_OPERATOR) &&
        strcmp(token_result[i - 1].content, "(") != 0 &&
        strcmp(token_result[i].content, "[") != 0 &&
        strcmp(token_result[i].content, "]") != 0 &&
        strcmp(token_result[i - 1].content, "[") != 0 &&
        strcmp(token_result[i - 1].content, "&") != 0 &&
        !((strcmp(token_result[i].content, "++") == 0 ||
           strcmp(token_result[i].content, "--") == 0) &&
          token_result[i - 1].type == TOK_IDENTIFIER) &&
        !((strcmp(token_result[i - 1].content, "++") == 0 ||
           strcmp(token_result[i - 1].content, "--") == 0) &&
          token_result[i].type == TOK_IDENTIFIER) &&
        !(strcmp(token_result[i].content, "(") == 0 &&
          (strcmp(token_result[i - 1].content, "for") != 0 &&
           strcmp(token_result[i - 1].content, "if") != 0 &&
           strcmp(token_result[i - 1].content, "while") != 0))) {
      strcat(new_line->line, " ");
    }
  }
  // 토큰 삽입
  strcat(new_line->line, token_result[i].content);

  // 라인의 끝이 ')'로 끝나면 다음 라인을 들여쓰기한다.
  if (i == token_result_count - 1 &&
      strcmp(token_result[i].content, ")") == 0) {
    indent_next = true;
```

```
    }
  }
  // 라인을 소스 파일에 삽입
  line_node *body = source_file->body;
  while (body->next != NULL) {
    body = body->next;
  }
  body->next = new_line;
}

// 토큰 길이가 0보다 크면 헤더 파일 노드를 생성한다.
if (token_result_count > 0) {
  line_node *header_file_line = malloc(sizeof(line_node));
  header_file_line->next = NULL;
  // 함수 프로토타입인 경우
  if (token_result_count > 3 && token_result[0].type == TOK_IDENTIFIER &&
      token_result[1].type == TOK_IDENTIFIER &&
      strcmp(token_result[2].content, "(") == 0) {
    i = 0;
    is_waiting_paren = true;
  }
  // 이전 라인에서 ')'을 만나지 못한 경우
  else if (is_waiting_paren) {
    i = 0;
  }
  // 아닌 경우 건너뛰기
  else {
    i = token_result_count;
  }

  // 프로토타입 토큰 간 띄어쓰기 구현
  for (; i < token_result_count; ++i) {
    if (i > 0 && strcmp(token_result[i].content, ";") != 0 &&
        strcmp(token_result[i].content, ",") != 0 &&
        strcmp(token_result[i].content, ")") != 0 &&
        !(string_ends_with(token_result[i - 1].content, ")") &&
          token_result[i].type != TOK_OPERATOR) &&
        strcmp(token_result[i - 1].content, "(") != 0 &&
        strcmp(token_result[i].content, "[") != 0 &&
        strcmp(token_result[i].content, "]") != 0 &&
        strcmp(token_result[i - 1].content, "[") != 0 &&
        strcmp(token_result[i - 1].content, "&") != 0 &&
        !(strcmp(token_result[i].content, "(") == 0 &&
          (strcmp(token_result[i - 1].content, "for") != 0 &&
```

```
            strcmp(token_result[i - 1].content, "if") != 0 &&
            strcmp(token_result[i - 1].content, "while") != 0))) {
        strcat(header_file_line->line, " ");
      }
      strcat(header_file_line->line, token_result[i].content);
      // ')'를 만난 경우
      if (strcmp(token_result[i].content, ")") == 0) {
        is_waiting_paren = false;
        // 세미콜론 삽입
        strcat(header_file_line->line, ";");
        break;
      }
    }

    // 헤더 파일 라인 길이가 0보다 크면 라인을 삽입한다.
    if (strlen(header_file_line->line) > 0) {
      line_node *header_file = source_file->header_file;
      while (header_file->next != NULL) {
        header_file = header_file->next;
      }
      header_file->next = header_file_line;
    }
    else {
      // 길이가 0인 경우 메모리 반환
      free(header_file_line);
    }
  }

  // 트림된 라인 메모리 반환
  free(line_trim);

  // 소스 코드 리턴
  // 한 줄씩 변환하여 출력할 때 사용
  return source_file;
}

// 코드 변환기를 종료하고 필요한 메시지를 출력하는 함수
void transpile_finalize(const char *output_path, const char *file_name,
                        bool print_java, bool print_c, bool print_size,
                        bool print_line_num, bool print_func_map)
{
  // 파일 변환이 종료되었다는 메시지를 출력한다.
  for (int i = 0; i < file_list_count; ++i) {
    printf("file %s converting is finished\n", file_list[i]);
```

```
  }
  // 변환한 자바 파일 출력
  if (print_java) {
    char path[PATH_MAX];
    char title[FILENAME_MAX];
    sprintf(path, "%s%s.java", output_path, file_name);
    sprintf(title, "%s.java", file_name);
    printf("\n");
    file_print_code(title, path);
  }
  // 변환된 C 파일 출력
  if (print_c) {
    char path[PATH_MAX];
    char title[FILENAME_MAX];
    for (int i = 0; i < file_list_count; ++i) {
      if (strncmp(file_list[i], file_name, strlen(file_name)) != 0) {
        sprintf(path, "%s%s", output_path, file_list[i]);
        path[strlen(path) - 1] = 'h';
        sprintf(title, "%s", file_list[i]);
        title[strlen(title) - 1] = 'h';
        printf("\n");
        file_print_code(title, path);
      }
      sprintf(path, "%s%s", output_path, file_list[i]);
      sprintf(title, "%s", file_list[i]);
      printf("\n");
      file_print_code(title, path);
    }
  }
  // 파일 크기 출력
  if (print_size) {
    // 변환한 자바 파일 크기 출력
    char path[PATH_MAX];
    sprintf(path, "%s%s.java", output_path, file_name);
    printf("\n");
#ifdef __APPLE__
    printf("%s file size is %lld bytes\n", path, file_get_size(path));
#else
    printf("%s file size is %ld bytes\n", path, file_get_size(path));
#endif
    // 변환된 C 파일 크기 출력
    for (int i = 0; i < file_list_count; ++i) {
      if (strncmp(file_list[i], file_name, strlen(file_name)) != 0) {
        // 헤더 파일이 존재하는 경우 헤더 파일 크기 출력
```

```
        sprintf(path, "%s%s", output_path, file_list[i]);
        path[strlen(path) - 1] = 'h';
#ifdef __APPLE__
        printf("%s file size is %lld bytes\n", path, file_get_size(path));
#else
        printf("%s file size is %ld bytes\n", path, file_get_size(path));
#endif
      }
      sprintf(path, "%s%s", output_path, file_list[i]);
#ifdef __APPLE__
      printf("%s file size is %lld bytes\n", path, file_get_size(path));
#else
      printf("%s file size is %ld bytes\n", path, file_get_size(path));
#endif
    }
  }
  // 파일 라인 수 출력
  if (print_line_num) {
    // 변환한 자바 파일 라인 수 출력
    char path[PATH_MAX];
    sprintf(path, "%s%s.java", output_path, file_name);
    printf("\n");
    printf("%s line number is %d lines\n", path, file_get_line(path));
    for (int i = 0; i < file_list_count; ++i) {
      // 변환된 C 파일 라인 수 출력
      if (strncmp(file_list[i], file_name, strlen(file_name)) != 0) {
        // 헤더 파일이 존재하는 경우 헤더 파일 라인 수 출력
        sprintf(path, "%s%s", output_path, file_list[i]);
        path[strlen(path) - 1] = 'h';
        printf("%s line number is %d lines\n", path, file_get_line(path));
      }
      sprintf(path, "%s%s", output_path, file_list[i]);
      printf("%s line number is %d lines\n", path, file_get_line(path));
    }
  }
  // 함수 매핑 출력
  if (print_func_map) {
    printf("\n");
    for (int i = 0; i < func_list_count; ++i) {
      printf("%d: %s() -> %s()\n", (i + 1), func_list[i].java, func_list[i].c);
    }
  }
}
```

```
/*
 * 코드 출력 관련 함수
 */

// 소스 코드를 쓰는 함수
void write_source_code(source_code *code, FILE *fp)
{
  // 표준 출력에 출력하고 코드가 NULL인 경우
  // == 현재 어느 클래스에도 포함되지 않은 경우
  // 빈 칸 출력
  if (code == NULL) {
    if (fp == stdout) {
      fprintf(fp, "--------\n\n--------\n");
    }
    return;
  }
  // 표준 출력에 출력하는 경우 포맷을 다르게 출력
  if (fp == stdout) {
    fprintf(fp, "--------\n%s.c\n--------\n", class_context);
  }
  // 표준 출력에는 라인 수 출력
  int line_num = 0;
  // 헤더 인클루드 출력
  line_node *header = code->header;
  while (header->next != NULL) {
    if (fp == stdout) {
      fprintf(fp, "%3d  ", ++line_num);
    }
    fprintf(fp, "%s\n", header->next->line);
    header = header->next;
  }
  // 사용자 정의 헤더 인클루드 출력
  line_node *user_header = code->user_header;
  while (user_header->next != NULL) {
    if (fp == stdout) {
      fprintf(fp, "%3d  ", ++line_num);
    }
    fprintf(fp, "%s\n", user_header->next->line);
    user_header = user_header->next;
  }
  // DEFINE 전처리문 출력
  line_node *define = code->define;
  while (define->next != NULL) {
    if (fp == stdout) {
```

```
      fprintf(fp, "%3d  ", ++line_num);
    }
    fprintf(fp, "%s\n", define->next->line);
    define = define->next;
  }
  // 전처리문과 코드 사이 공백 삽입
  if (fp == stdout) {
    fprintf(fp, "%3d  ", ++line_num);
  }
  fprintf(fp, "\n");
  // 코드 출력
  line_node *body = code->body;
  while (body->next != NULL) {
    if (fp == stdout) {
      fprintf(fp, "%3d  ", ++line_num);
    }
    fprintf(fp, "%s\n", body->next->line);
    body = body->next;
  }
}

// 소스 코드의 헤더 파일을 쓰는 함수
void write_source_header(source_code *code, FILE *fp)
{
  line_node *header_file = code->header_file;
  while (header_file->next != NULL) {
    fprintf(fp, "%s\n", header_file->next->line);
    header_file = header_file->next;
  }
}

// 'Makefile'을 쓰는 함수
void write_makefile(const char *output_path, const char *file_name)
{
  FILE *fp;
  char makefile_path[PATH_MAX];
  sprintf(makefile_path, "%s%s_Makefile", output_path, file_name);
  if ((fp = fopen(makefile_path, "w")) == NULL) {
    print_fatal_error("open file error for %s", makefile_path);
  }
  fprintf(fp, "%s:", file_name);
  for (int i = 0; i < file_list_count; ++i) {
    fprintf(fp, " %s", file_list[i]);
  }
```

```
  fprintf(fp, "\n\tgcc");
  for (int i = 0; i < file_list_count; ++i) {
    fprintf(fp, " %s", file_list[i]);
  }
  fprintf(fp, " -o %s:\n\n.PHONY: clean\nclean:\n\trm -rf ./%s\n", file_name,
          file_name);
  fclose(fp);
}

/*
 * 기타 함수
 */

// 헤더 테이블을 불러오는 함수
header_map_node *load_header_map()
{
  header_map_node *ret = (header_map_node *)malloc(sizeof(header_map_node));
  ret->next = NULL;

  // header_table.txt 로딩
  char *header_table_file = file_get_data("header_table.txt");
  if (header_table_file == NULL) {
    // 헤더 테이블 파일이 존재하지 않은 경우 에러 출력
    print_fatal_error("header table not exist.");
  }

  char *line;
  char *save_ptr;

  // 헤더 테이블 라인 간 분리
  line = strtok_r(header_table_file, "\n", &save_ptr);
  while (line != NULL) {

    // 링크드 리스트 생성
    header_map_node *node = (header_map_node *)malloc(sizeof(header_map_node));
    node->header = (header_node *)malloc(sizeof(header_node));
    node->header->next = NULL;
    node->next = NULL;

    char *func;
    char *includes;
    char *save_ptr_2;

    // 띄어쓰기 간 분리
```

```
    func = strtok_r(line, " ", &save_ptr_2);
    strcpy(node->func, func);
    while (strtok_r(NULL, " ", &save_ptr_2) != NULL) {
      includes = strtok_r(NULL, " ", &save_ptr_2);
      if (includes == NULL) {
        // 헤더 테이블 포맷이 올바르지 않은 경우 에러 출력
        print_fatal_error("header table format error");
      }
      header_node *header = (header_node *)malloc(sizeof(header_node));
      header->next = NULL;
      strcpy(header->header, includes);
      header_node *cur = node->header;
      while (cur->next != NULL) {
        cur = cur->next;
      }
      cur->next = header;
    }
    header_map_node *cur = ret;
    while (cur->next != NULL) {
      cur = cur->next;
    }
    cur->next = node;
    line = strtok_r(NULL, "\n", &save_ptr);
  }
  free(header_table_file);
  return ret;
}

// 함수에 필요한 헤더를 인클루드하는 함수
void add_func_header(const char *func_name)
{
  header_map_node *cur = header_table.head->next;
  while (cur != NULL) {
    if (strcmp(cur->func, func_name) == 0) {
      header_node *cur_header = cur->header->next;
      while (cur_header != NULL) {
        add_include(cur_header->header);
        cur_header = cur_header->next;
      }
      return;
    }
    else {
      cur = cur->next;
    }
```

```
  }
}

// 헤더 파일이 존재하지 않으면 추가하는 함수
void add_include(const char *include_str)
{
  line_node *cur = source_file->header;
  while (cur->next != NULL) {
    // 헤더 파일 중복 검사
    if (string_ends_with(cur->next->line, include_str)) {
      return;
    }
    cur = cur->next;
  }
  // 중복이 없는 경우 헤더 파일 추가
  line_node *new_line = (line_node *)malloc(sizeof(line_node));
  sprintf(new_line->line, "#include %s", include_str);
  new_line->next = NULL;
  cur->next = new_line;
}

// 사용자 정의 헤더 파일을 인클루드 하는 함수
void add_user_include(const char *include_str)
{
  line_node *cur = source_file->user_header;
  while (cur->next != NULL) {
    // 헤더 파일 중복 검사
    if (string_ends_with(cur->next->line, include_str)) {
      return;
    }
    cur = cur->next;
  }
  // 중복이 없는 경우 헤더 파일 추가
  line_node *new_line = (line_node *)malloc(sizeof(line_node));
  sprintf(new_line->line, "#include %s", include_str);
  new_line->next = NULL;
  cur->next = new_line;
}

// JAVA 함수 - C 함수의 매핑을 추가하는 함수
void add_func_map(const char *java_func, const char *c_func)
{
  for (int i = 0; i < func_list_count; ++i) {
    if (strcmp(java_func, func_list[i].java) == 0) {
```

```
        return;
      }
    }
    strcpy(func_list[func_list_count].java, java_func);
    strcpy(func_list[func_list_count].c, c_func);
    ++func_list_count;
}
```

<util.h>

```
#ifndef SSU_CONVERT_UTIL_H
#define SSU_CONVERT_UTIL_H

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdbool.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/time.h>

/*
 * 출력 관련 함수
 */
// 에러 메시지를 출력하고 종료한다.
void print_fatal_error(const char *format, ...);
// 코드를 출력한다.
void print_code(const char *title, const char *code, int line_limit);
// 프로그램의 실행 시간을 출력한다.
void ssu_runtime(struct timeval *begin_t, struct timeval *end_t);

/*
 * 파일 관련 함수
 */
// 파일이 존재하는지 확인한다.
bool file_exists(const char *pathname);
// 파일을 오픈하여 읽어들인다.
char *file_get_data(const char *pathname);
// 파일의 라인 수를 가져온다.
int file_get_line(const char *pathname);
// 파일의 크기를 가져온다.
off_t file_get_size(const char *pathname);
```

```
// 코드 파일을 읽어들여서 출력한다.
void file_print_code(const char *title, const char *pathname);

/*
 * 문자열 관련 함수
 */
// 문자열 좌우의 공백을 제거하여 반환한다.
char *string_trim(const char *str);
// 문자열이 특정 문자열로 끝나는지 확인한다.
bool string_ends_with(const char *str, const char *suffix);

#endif
```

<util.c>

```
#include "util.h"

/*
 * 출력 관련 함수
 */

// 에러 메시지를 출력하고 종료한다.
void print_fatal_error(const char *format, ...)
{
  fprintf(stderr, "\033[0;31m");
  fprintf(stderr, "error: ");
  va_list args;
  va_start(args, format);
  vfprintf(stderr, format, args);
  va_end(args);
  fprintf(stderr, "\n");
  fprintf(stderr, "\033[0m");
  exit(1);
}

// 코드를 출력한다.
void print_code(const char *title, const char *code, int line_limit)
{
  printf("--------\n%s\n--------\n", title);
  char *code_dup = strdup(code);
  char *line;
  char *save_ptr;
  int line_num = 0;
```

```
  line = strtok_r(code_dup, "\n", &save_ptr);

  while (line != NULL && (line_limit == 0 || line_num < line_limit)) {
    printf("%3d  %s\n", ++line_num, line);
    line = strtok_r(NULL, "\n", &save_ptr);
  }

  free(code_dup);
}

// 프로그램의 실행 시간을 출력한다.
void ssu_runtime(struct timeval *begin_t, struct timeval *end_t)
{
  end_t->tv_sec -= begin_t->tv_sec;
  if (end_t->tv_usec < begin_t->tv_usec) {
    end_t->tv_sec--;
    end_t->tv_usec += 1000000;
  }
  end_t->tv_usec -= begin_t->tv_usec;
#ifdef __APPLE__
  printf("\nRuntime: %ld:%06d(sec:usec)\n", end_t->tv_sec, end_t->tv_usec);
#else
  printf("\nRuntime: %ld:%06ld(sec:usec)\n", end_t->tv_sec, end_t->tv_usec);
#endif
}

/*
 * 파일 관련 함수
 */

// 파일이 존재하는지 확인한다.
bool file_exists(const char *pathname)
{
  return access(pathname, F_OK) == 0;
}

// 파일을 오픈하여 읽어들인다.
char *file_get_data(const char *pathname)
{
  if (!file_exists(pathname)) {
    return NULL;
  }
  int fd;
```

```
  if ((fd = open(pathname, O_RDONLY)) < 0) {
    print_fatal_error("file open error for %s", pathname);
  }
  off_t fsize;
  if ((fsize = lseek(fd, 0, SEEK_END)) < 0) {
    print_fatal_error("lseek error");
  }
  char *buf = (char *)malloc(fsize + 1);
  if (lseek(fd, 0, SEEK_SET) < 0) {
    print_fatal_error("lseek error");
  }
  ssize_t len;
  if ((len = read(fd, buf, fsize)) < 0) {
    print_fatal_error("file read error for %s", pathname);
  }
  close(fd);
  buf[len] = 0;
  if (strlen(buf) == 0) {
    return NULL;
  }
  return buf;
}

// 파일의 라인 수를 가져온다.
int file_get_line(const char *pathname)
{
  char *file_src = file_get_data(pathname);
  if (file_src == NULL) {
    return -1;
  }
  int line = 0;
  for (unsigned long i = 0; i < strlen(file_src); ++i) {
    if (file_src[i] == '\n')
      ++line;
  }
  free(file_src);
  return line;
}

// 파일의 크기를 가져온다.
off_t file_get_size(const char *pathname)
{
  if (!file_exists(pathname)) {
    return -1;
```

```
  }
  int fd;
  if ((fd = open(pathname, O_RDONLY)) < 0) {
    print_fatal_error("file open error for %s", pathname);
  }
  off_t fsize;
  if ((fsize = lseek(fd, 0, SEEK_END)) < 0) {
    print_fatal_error("lseek error");
  }
  return fsize;
}

// 코드 파일을 읽어들여서 출력한다.
void file_print_code(const char *title, const char *pathname)
{
  char *file_src = file_get_data(pathname);
  if (file_src == NULL) {
    return;
  }

  print_code(title, file_src, 0);

  free(file_src);
}

/*
 * 문자열 관련 함수
 */

// 문자열 좌우의 공백을 제거하여 반환한다.
char *string_trim(const char *str)
{
  char *tmp = strdup(str);
  char *cur = tmp;
  while (*cur == ' ' || *cur == '\t')
    cur++;
  if (*tmp == 0) {
    free(tmp);
    return NULL;
  }
  char *end = cur + strlen(cur) - 1;
  while (end > str && (*end == ' ' || *end == '\t'))
    end--;
  end[1] = '\0';
```

```
  char *ret = strdup(cur);
  free(tmp);
  return ret;
}

// 문자열이 특정 문자열로 끝나는지 확인한다.
bool string_ends_with(const char *str, const char *suffix)
{
  const int len_diff = (int)strlen(str) - (int)strlen(suffix);
  return (len_diff >= 0) && (strcmp(str + len_diff, suffix) == 0);
}
```