4장 C 표준 파일 입출력

리눅스 시스템 프로그래밍

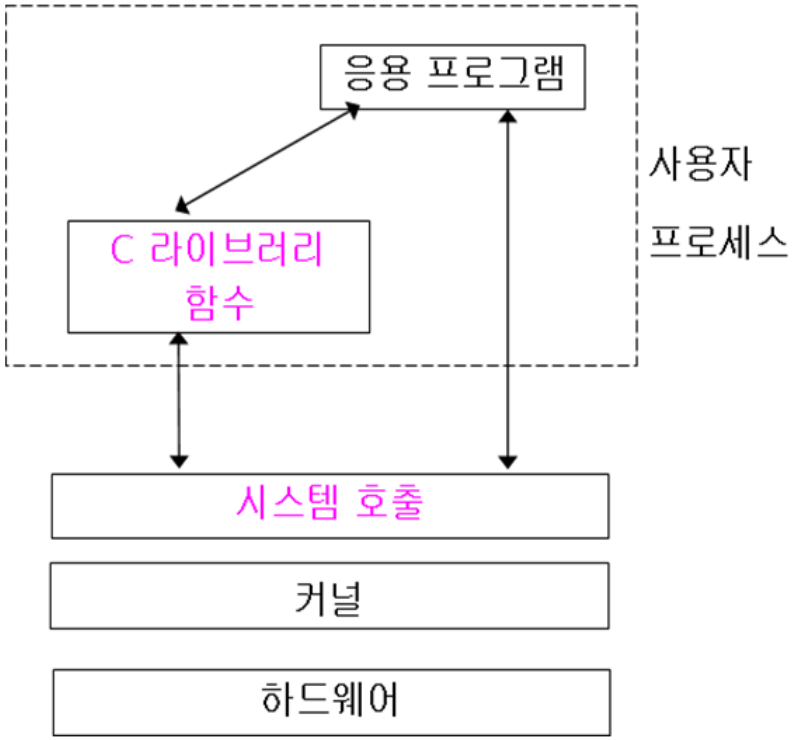
청주대학교 전자공학과
한철수

목차

- 파일 및 파일 포인터
- 텍스트 파일
- 이진 파일
- 임의 접근
- 버퍼 입출력
- 기타 함수

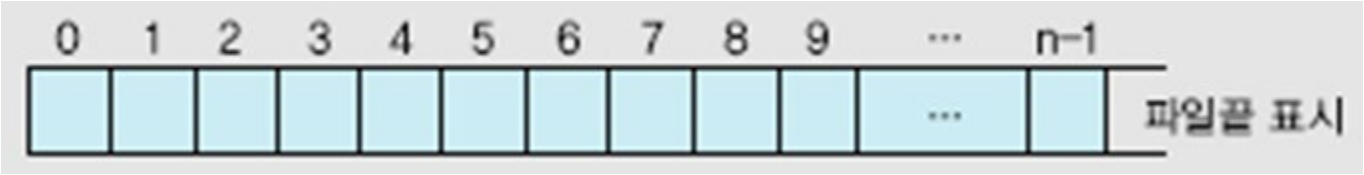
시스템 호출과 C 라이브러리 함수

- 응용 프로그램이 커널에 서비스를 요청하는 방법
  - 방법 1: 시스템 호출(system call)을 이용하여 요청.
  - 방법 2: C 언어가 제공하는 라이브러리 함수를 이용하여 요청.
    - 라이브러리 함수는 내부적으로 시스템 호출을 이용함.
    - 시스템 호출을 쉽게 사용할 수 있도록 함수를 만들었음.



C의 파일

- C 언어가 제공하는 파일 입출력을 이해하기 위해서는 C 파일에 대한 이해가 필요함.
- C 파일은 모든 데이터를 연속된 바이트 형태로 저장함
  - 저장된 데이터의 종류에 따라 **텍스트 파일**과 **이진 파일**로 구분됨.



텍스트 파일과 이진 파일

- 텍스트 파일(text file)
  - 문자들만으로 이루어진 파일로서 한글, 영문, 숫자 등의 문자들을 포함하고 있음.
  - 여러 개의 줄로 이루어지며 매 줄마다 줄의 끝을 나타내는 개행 문자(‘₩n’)를 포함하고 있음.
- 이진 파일(binary file)
  - 모든 데이터를 컴퓨터 내부의 이진수 표현 그대로 저장한 파일로 이미지 파일이나 실행 파일 등이 이진 파일임.
  - 또한, 메모리에 저장된 변수 값을 이진수 표현 그대로 이진 파일로 저장하는 경우가 많음.

C 언어의 파일 입출력 과정

1. 파일 열기
   - fopen( ) 함수를 사용함.
2. 파일 입출력을 실시
   - 다양한 파일 입출력 함수를 사용하여 파일의 내용을 읽거나 저장함.
3. 파일 닫기
   - fclose( ) 함수를 사용함.

파일 열기

- 파일을 사용하기 위해서는 반드시 fopen() 함수를 사용하여 파일을 열어야 함.
  - 파일 열기 함수 fopen()을 사용하면 운영체제는 그 파일에 대한 각종 정보를 FILE 구조체 변수로 구성하고, 그 변수의 포인터(주소 값)를 반환함.
  - **FILE 구조체 변수**에 대한 포인터를 간단히 **FILE 포인터**라 함.
- fopen() 함수가 반환하는 FILE 포인터는 잘 보관해야 함.
  - 파일 입출력 또는 파일 닫기를 하기 위해서는 FILE 포인터가 필요하기 때문.

FILE 구조체

- FILE 구조체 안에는 열린 파일을 위한 여러 변수들이 선언되어 있음.
  - 버퍼에 대한 정보, 파일 입출력 모드, 파일 디스크립터 등을 포함함.
- FILE 구조체의 예

```
typedef struct {
    int cnt;                    // 버퍼의 남은 문자 수
    unsigned char *base;        // 버퍼 시작
    unsigned char *ptr;         // 버퍼의 현재 포인터
    unsinged flag;              // 파일 입출력 모드
    int fd;                     // 파일 디스크립터
} FILE;
```

파일 열기 함수 fopen()

- 함수 프로토타입
  - FILE *fopen(const char *filename, const char *mode);
    - filename
      - 열고자 하는 파일 이름
    - mode
      - 파일 입출력 모드
    - 파일 열기가 성공하면 FILE 포인터를 반환하고, 실패하면 NULL을 반환함.
- 텍스트 파일에 사용할 수 있는 입출력 모드

모드	의미	파일이 없으면	파일이 있으면
"r"	읽기 전용(read)	NULL 반환	정상 동작
"w"	쓰기 전용(write)	새로 생성	기존 내용 삭제
"a"	추가 쓰기(append)	새로 생성	기존 내용 뒤에 추가
"r+"	읽기와 쓰기	NULL 반환	정상 동작
"w+"	읽기와 쓰기	새로 생성	기존 내용 삭제
"a+"	추가를 위한 읽기와 쓰기	새로 생성	기존 내용 뒤에 추가

파일 열기 예

- 파일을 열때에는 fopen() 함수가 반환하는 FILE 포인터를 반드시 저장해야 함.
- 사용 예 1

```
FILE *fp = fopen(“~/sp/text.txt”, “r”);        // 파일을 읽기 모드로 염.

if (fp == NULL) {                              // 읽기 모드인 경우
    printf("파일 열기 오류₩n");
}
```

- 사용 예 2

```
FILE *fp2 = fopen("outdata.txt", "w");      // 파일을 쓰기 모드로 염.
```

표준 입출력 스트림

- 스트림(stream)
  - fopen() 함수를 통해 열린 파일을 스트림이라고 함.
- 표준 입출력 스트림
  - C 프로그램이 실행될 때 자동으로 열리고 종료될 때 닫히는 스트림임.
  - 표준 헤더 파일 <stdio.h>에 3개의 스트림이 정의되어 있음.
    - stdin, stdout, stderr
    - 모두 FILE 포인터임. (FILE 포인터≈열린 파일≈스트림)

표준 입출력 포인터	설명	가리키는 장치
stdin	표준 입력에 대한 FILE 포인터	키보드
stdout	표준 출력에 대한 FILE 포인터	모니터
stderr	표준 오류에 대한 FILE 포인터	모니터

파일 닫기

- 열린 파일을 더 이상 사용하지 않는다면 fclose() 함수를 사용하여 닫아야 함.
- fclose() 함수의 프로토타입
  - int fclose(FILE *fp);
    - fp
      - FILE 포인터
    - 파일 닫기에 성공하면 0을 반환하고, 실패하면 -1을 반환함.
- 사용 예

```
FILE *fp3 = fopen("name.txt", “r"); // 파일을 읽기 모드로 엶.

if (fp == NULL) {                   // 읽기 모드인 경우
    printf("파일 열기 오류₩n");
}

fclose(fp3);                        // 파일을 닫음.
```

텍스트 파일 입출력 함수

- C 언어는 텍스트 파일에 입출력하기 위한 함수들을 제공함.

표준 입출력함수	파일 입출력 함수	기능
getchar()	fgetc(), getc()	문자단위로 입력하는 함수
putchar()	fputc(), putc()	문자단위로 출력하는 함수
gets()	fgets()	문자열을 입력하는 함수
puts()	fputs()	문자열을 출력하는 함수
scanf()	fscanf()	자료형에 따라 자료를 입력하는 함수
printf()	fprintf()	자료형에 따라 자료를 출력하는 함수

문자 단위 파일 입출력 함수

- 문자 단위 파일 입출력 함수

함수	기능
fgetc(), getc()	문자 단위로 입력하는 함수
fputc(), putc()	문자 단위로 출력하는 함수

- 함수 프로토타입
  - int fgetc(FILE *fp);
    - fp가 가리키는 파일에서 한 문자를 읽어서 반환함.
    - 파일 끝에 도달했을 경우에는 EOF를 반환함.
  - int fputc(int c, FILE *fp);
    - fp가 가리키는 파일에 한 문자(c)를 출력하고 출력된 문자를 반환함.
    - 문자 출력 시 오류가 발생하면 EOF를 반환함.

cat.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *fp=NULL;

    if (argc < 2)
        fp = stdin;
    else
        fp = fopen(argv[1],"r");

    int c = getc(fp);

    while (c != EOF) {
        putc(c, stdout);
        c = getc(fp);
    }

    fclose(fp);
    return 0;
}
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o cat cat.c
cju@cju-VirtualBox:~/chap4$ ./cat cat.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *fp=NULL;

    if (argc < 2)
        fp = stdin;
    else
        fp = fopen(argv[1],"r");

    int c = getc(fp);

    while (c != EOF) {
        putc(c, stdout);
        c = getc(fp);
    }

    fclose(fp);
    return 0;
}

cju@cju-VirtualBox:~/chap4$
```

명령줄 인수

- 명령줄 인수(command line argument)란 커맨드라인에서 프로그램을 실행시킬 때 프로그램 내부로 전달하는 인수를 말함.
- 명령줄 인수를 사용하지 않을 때
  - main 함수의 형태: int main(void)
  - 프로그램 실행 방법 : ./실행파일명 ./mycat
- 명령줄 인수를 사용할 때
  - main 함수의 형태: int main(int argc, char *argv[])
  - 프로그램 실행 방법 : ./실행파일명 인수1 인수2 … ./mycat 1 2
    ./mycat -w

copy.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc !=3) {
        fprintf(stderr, "사용법: %s 파일1 파일2₩n", argv[0]);
        return 1;
    }
    FILE *fp1 = fopen(argv[1], "r");
    if (fp1 == NULL) {
        fprintf(stderr, "파일 %s 열기 오류₩n", argv[1]);
        return 2;
    }
    FILE *fp2 = fopen(argv[2], "w");
    char c;
    while ((c = fgetc(fp1)) != EOF)
        fputc(c, fp2);

    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o copy copy.c
cju@cju-VirtualBox:~/chap4$ ./copy copy.c copy.bak
cju@cju-VirtualBox:~/chap4$ ls -l copy*
-rwxrwxr-x 1 cju cju 16176  3월 30 05:45 copy
-rw-rw-r-- 1 cju cju   474  3월 30 05:46 copy.bak
-rw-rw-r-- 1 cju cju   474  3월 30 05:45 copy.c
cju@cju-VirtualBox:~/chap4$
```

기타 파일 관련 함수

- int feof(FILE *fp)
  - fp가 가리키는 파일의 끝에 도달하면 0이 아닌 값을 반환하고, 파일 끝이 아니면 0을 반환함.
- int ungetc(int c, FILE *p)
  - c에 저장된 문자를 스트림에 반환함.
  - 마치 문자를 읽지 않은 것처럼 파일 위치 포인터를 1 감소시키고 그 위치에 c를 저장함.

질문

Q&A

줄 단위 파일 입출력 함수

- 줄 단위 파일 입출력 함수

함수	기능
fgets()	한 줄을 입력하는 함수
fputs()	한 줄을 출력하는 함수

함수 프로토타입

- char *fgets(char *s, int n, FILE *fp);
  - fp가 가리키는 열린 파일로부터 한 줄을 읽어서 문자열 포인터 s가 가리키는 곳에 저장하고 s를 반환함.
  - 개행 문자('₩n')나 EOF를 만날 때까지 파일로부터 최대 n-1 개의 문자를 읽고, 읽어온 데이터의 끝에는 NULL 문자를 붙임.
  - 오류가 발생하면 NULL을 반환함.
- int fputs(const char *s, FILE *fp);
  - 문자열 s를 fp가 가리키는 열린 파일에 출력(저장)함.
  - 문자열 끝을 나타내는 NULL 문자는 출력하지 않음.
  - 실행이 성공하면 파일에 출력한 바이트 수를 반환하고, 실패하면 EOF를 반환함.

line.c

```c
#include <stdio.h>
#define MAXLINE 80
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "사용법: line 파일이름₩n");
        return 1;
    }
    FILE *fp;
    if ((fp = fopen(argv[1],"r")) == NULL){
        fprintf(stderr, "파일 열기 오류₩n");
        return 2;
    }
    int line = 0;
    char buffer[MAXLINE];
    while (fgets(buffer, MAXLINE, fp) != NULL) {
        line++;
        printf("%3d %s", line, buffer);
    }
    fclose(fp);
    return 0;
}
```

명령줄 인수가 1개가 아니면

열린 파일 fp로부터 한 줄을 읽어와 배열 buffer에 저장함. 이 때 buffer의 길이는 MAXLINE임. 읽기가 실패하지 않았다면

```
cju@cju-VirtualBox:~/chap4$ gcc -o line line.c
cju@cju-VirtualBox:~/chap4$ ./line line.c
 1 #include <stdio.h>
 2 #define MAXLINE 80
 3 int main(int argc, char *argv[])
 4 {
 5     if (argc != 2) {
 6         fprintf(stderr, "사용법: line 파일이름\n");
 7         return 1;
 8     }
 9     FILE *fp;
10     if ((fp = fopen(argv[1],"r")) == NULL){
11         fprintf(stderr, "파일 열기 오류\n");
12         return 2;
13     }
14     int line = 0;
15     char buffer[MAXLINE];
16     while (fgets(buffer, MAXLINE, fp) != NULL) {
17         line++;
18         printf("%3d %s", line, buffer);
19     }
20     fclose(fp);
21     return 0;
22 }
23
cju@cju-VirtualBox:~/chap4$
```

포맷 입출력 함수

- 포맷 입출력 함수

함수	기능
fscanf()	자료형에 따라 자료를 입력하는 함수
fprintf()	자료형에 따라 자료를 출력하는 함수

함수 프로토타입

- int fprintf(FILE *fp, const char *format, ...);
  - fp는 열린 파일을 가리키는 FILE 포인터임.
  - 두 번째부터의 인자는 printf() 함수와 사용법이 동일함.
- int fscanf(FILE *fp, const char *format, ...);
  - fp는 열린 파일을 가리키는 FILE 포인터임.
  - 두 번째부터의 인자는 scanf() 함수와 사용법이 동일함.
  - 파일에서 읽기 성공한 항목의 개수를 반환함.
- 사용 예

```
int num;
fscanf(fp, “%d”, &num);
double d=3.5;
fprintf(fp, “%d %f”, num, d);
```

fprint.c

```c
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
    if (argc != 2) {          ← 명령줄 인수가 1개가 아니면
        fprintf(stderr, "사용법: %s 파일이름₩n", argv[0]);
        return 1;
    }
    FILE *fp = fopen(argv[1], "w");
    printf("%-9s %-7s %-4s₩n", "학번", "이름", "점수");

    struct student rec;       ← 구조체 변수 rec을 선언함.
    while (scanf("%d %s %d", &rec.id, rec.name, &rec.score)==3)  ← 3개가 입력되었다면
        fprintf(fp, "%d %s %d ", rec.id, rec.name, rec.score);

    fclose(fp);
    return 0;
}
```

student.h

```c
#define START_ID 1001001
struct student {
    int id;
    char name[20];
    int score;
};
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o fprint fprint.c
cju@cju-VirtualBox:~/chap4$ ./fprint stud.txt
학번    이름  점수
1001001 박연아 96
1001003 김태환 85
1001006 김현진 88
1001009 장샛별 75
cju@cju-VirtualBox:~/chap4$
```

fscan.c

```
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "사용법: %s 파일이름₩n", argv[0]);
        return 1;
    }
    FILE *fp = fopen(argv[1], "r");
    printf("%-9s %-7s %-4s₩n", "학번", "이름", "점수");

    struct student rec;
    while (fscanf(fp,"%d %s %d", &rec.id, rec.name, &rec.score)==3)
        printf("%d %s %d₩n", rec.id, rec.name, rec.score);

    fclose(fp);
    return 0;
}
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o fscan fscan.c
cju@cju-VirtualBox:~/chap4$ ./fscan stud.txt
학번     이름   점수
1001001 박연아 96
1001003 김태환 85
1001006 김현진 88
1001009 장샛별 75
cju@cju-VirtualBox:~/chap4$
```

이진 파일

- 이진 파일(binary file)은 데이터를 이진수 표현으로 저장한 파일임.
  - 이미지 파일이나 실행 파일이 대표적인 이진 파일임.
  - 또한, 메모리에 저장된 변수 값을 이진수 표현 그대로 이진 파일로 저장하는 경우가 많음.

이진 파일 열기

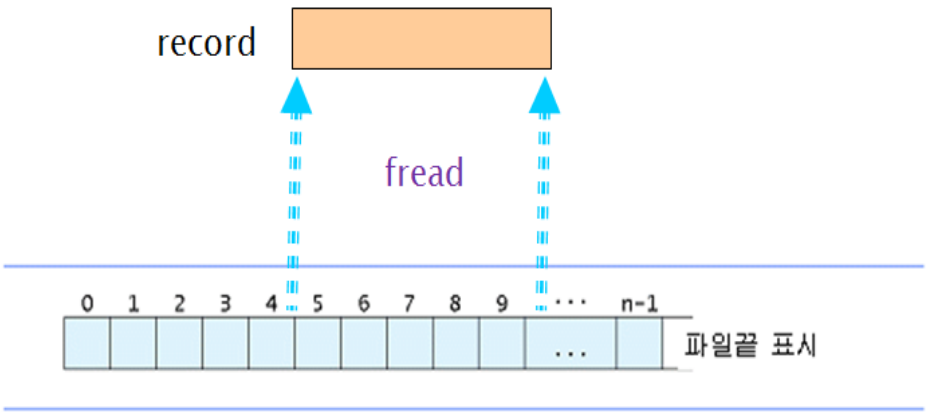
- 이진 파일로부터 입출력을 하기 위해서는 먼저 이진 파일을 열어야 함.
  - 이진 파일도 fopen() 함수를 호출하여 엶.

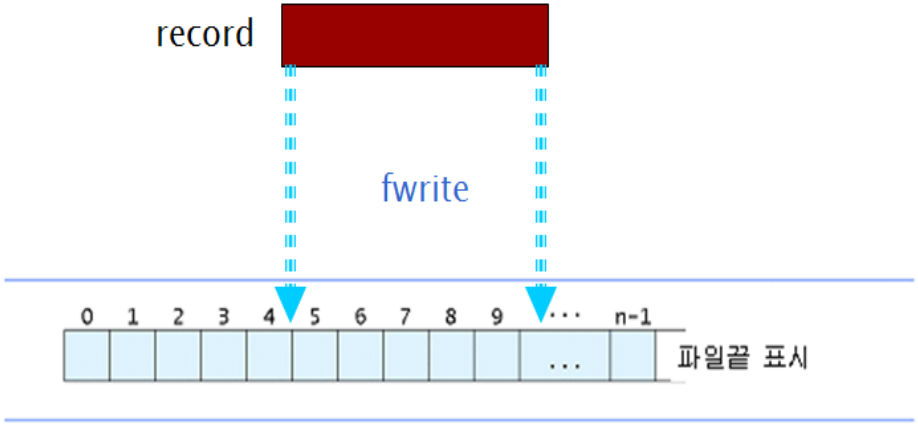
    FILE *fp = fopen("file", "**rb**");
  - 입출력 모드에 **b**가 붙고, 각 모드의 의미와 동작은 텍스트 파일과 같음.
- 이진 파일에 사용할 수 있는 입출력 모드

모드	의미	파일이 없으면	파일이 있으면
"rb"	읽기 전용(read)	NULL 반환	정상 동작
"wb"	쓰기 전용(write)	새로 생성	기존 내용 삭제
"ab"	추가 쓰기(append)	새로 생성	기존 내용 뒤에 추가
"rb+"	읽기와 쓰기	NULL 반환	정상 동작
"wb+"	읽기와 쓰기	새로 생성	기존 내용 삭제
"ab+"	추가를 위한 읽기와 쓰기	새로 생성	기존 내용 뒤에 추가

블록 단위 입출력

- 블록 단위 입출력이란 일정한 크기의 연속된 데이터를 한꺼번에 파일에서 읽거나 파일에 쓰는 것을 말함.
  - 예를 들어, 파일에서 한 번에 100 바이트씩 읽어오거나 저장하는 경우를 말함.
  - 또는, 한 번에 하나의 변수 크기만큼씩 읽어오거나 저장하는 경우도 많음.





함수 프로토타입

- int fread(void *buf, int size, int n, FILE *fp);
  - fp가 가리키는 열린 파일에서 size 바이트만큼의 데이터를 n번 읽어서 (결국 size*n바이트만큼의 데이터를), 포인터 buf가 가리키는 곳에 저장함.
  - 읽어온 바이트 수를 반환함.
- int fwrite(const void *buf, int size, int n, FILE *fp);
  - buf에 저장되어 있는 size 바이트만큼의 데이터를 n번 읽어서(결국 size*n바이트만큼의 데이터를) fp가 가리키는 열린 파일에 기록함.
  - 성공적으로 출력한 바이트 수를 반환함.
- 사용 예

```
int data1;
fread(&data1, sizeof(data1), 1, fp1);
double data2=3.5;
fwrite(&data2, sizeof(data2), 1, fp2);
```

← 열린 파일 fp1에서 sizeof(int) 바이트만큼의 데이터를 1번 읽어서 변수 data1에 저장함.

← 변수 data2에서 sizeof(double) 바이트만큼의 데이터를 1번 읽어서 열린 파일 fp2에 저장함.

stcreate1. c

```c
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "사용법: %s 파일이름₩n",argv[0]);
        return 1;
    }
    FILE * fp = fopen(argv[1], "wb");
    printf("%-9s %-7s %-4s₩n", "학번", "이름", "점수");

    struct student rec;
    while (scanf("%d %s %d", &rec.id, rec.name, &rec.score) == 3)
        fwrite(&rec, sizeof(rec), 1, fp);

    fclose(fp);
    return 0;
}
```

student.h

```c
#define START_ID 1001001
struct student {
    int id;
    char name[20];
    int score;
};
```

변수 rec에서 sizeof(rec) 크기만큼의 데이터를 1번 읽어서 열린 파일 fp에 저장함.

```
cju@cju-VirtualBox:~/chap4$ gcc -o stcreate1 stcreate1.c
cju@cju-VirtualBox:~/chap4$ ./stcreate1 stdb1
학번     이름  점수
1001001 박연아 96
1001003 김태환 85
1001006 김현진 88
1001009 장샛별 75
cju@cju-VirtualBox:~/chap4$
```

stprint.c

```c
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
   if (argc != 2) {
      fprintf(stderr, "사용법: %s 파일이름₩n", argv[0]);
      return 1;
   }
   FILE *fp=NULL;
   if ((fp = fopen(argv[1], "rb")) == NULL ) {
      fprintf(stderr, "파일 열기 오류₩n");
      return 2;
   }
   printf("----------------------------------₩n");
   printf("%12s %8s %8s₩n", "학번", "이름", "점수");
   printf("----------------------------------₩n");
   struct student rec;
   while (fread(&rec, sizeof(rec), 1, fp) > 0)
      if (rec.id != 0)
         printf("%10d %6s %6d₩n", rec.id,  rec.name,  rec.score);
   printf("----------------------------------₩n");
   fclose(fp);
   return 0;
}
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o stprint stprint.c
cju@cju-VirtualBox:~/chap4$ ./stprint stdb1
----------------------------------
      학번    이름    점수
----------------------------------
   1001001 박연아      96
   1001003 김태환      85
   1001006 김현진      88
   1001009 장샛별      75
----------------------------------
cju@cju-VirtualBox:~/chap4$
```

열린 파일 fp에서 sizeof(rec) 크기만큼의 데이터를 1번 읽어서 변수 rec에 저장하고, 읽어온 바이트가 있다면

파일에서 읽어와 저장한 rec 변수의 id 항목이 0이 아니라면, 문자열을 출력함. (rec.id != 0 문장은 4.4절 임의 접근에서 중요한 의미를 갖음.)

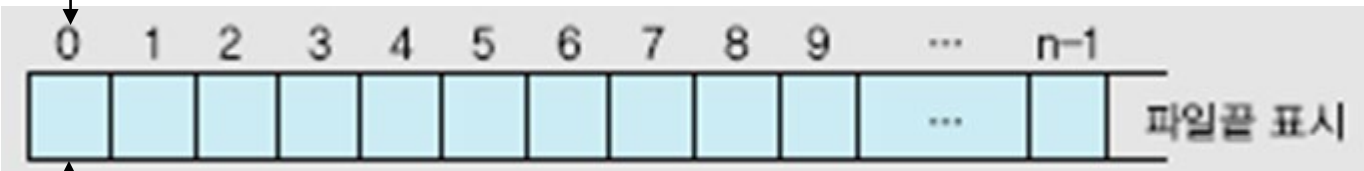
질문

Q&A

임의 접근의 필요성

- 순차 접근(sequential access)
  - 입출력 함수를 호출하여 데이터를 읽거나 쓰면, 읽거나 쓴 바이트 수만큼 “현재 파일 위치”가 자동으로 증가함.
  - 입출력 함수를 계속하여 호출하여 원하는 데이터에 접근하는 것을 순차 접근이라고 함.
  - 파일의 크기가 매우 크고 원하는 데이터가 파일의 중간 중간에 있을 때, 순차 접근은 불편하고 많은 시간을 낭비함.
- 임의 접근(random access)
  - 원하는 데이터가 있는 위치로 현재 파일 위치를 바로 변경하여 데이터에 접근하는 것을 말함.

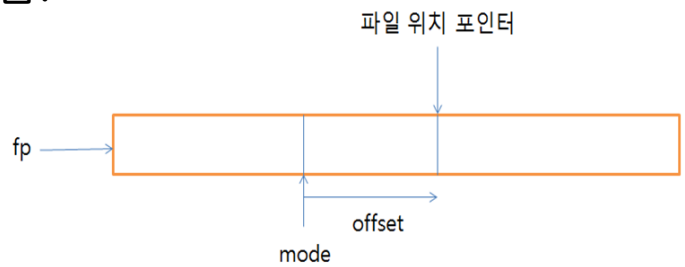
현재 파일 위치 (current file position, 열린 파일에서 다음으로 읽거나 쓸 파일 내의 위치)



파일 위치 포인터 (file position pointer, 현재 파일 위치를 저장하고 있는 포인터 변수)

임의 접근 관련 함수

- 함수 프로토타입
  - fseek(FILE *fp, long offset, int mode)
    - fp가 가리키는 열린 파일의 현재 파일 위치를 mode를 기준으로 offset만큼 이동시킴.



mode 기호	값	의미
SEEK_SET	0	파일 시작
SEEK_CUR	1	현재 위치
SEEK_END	2	파일 끝

  - rewind(FILE *fp)
    - fp가 가리키는 열린 파일의 현재 파일 위치를 파일 시작으로 이동시킴.
  - ftell(FILE *fp)
    - fp가 가리키는 열린 파일의 현재 파일 위치를 반환함.

fseek() 함수의 사용 예

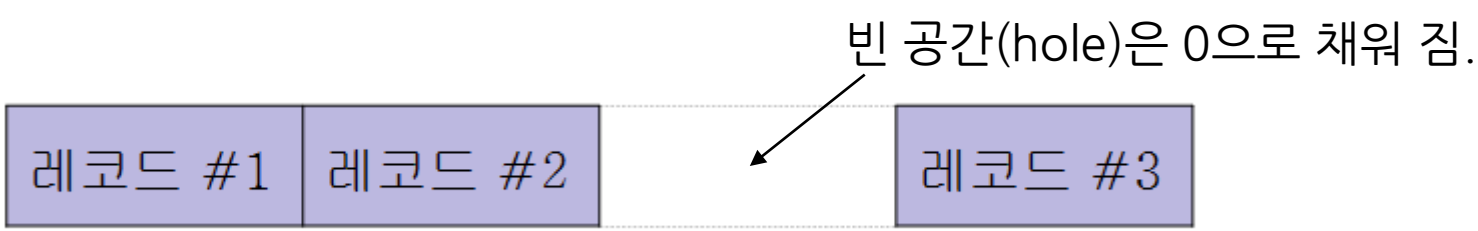
- 기본적인 현재 파일 위치의 이동 예
  - fseek(fd, 0L, SEEK_SET); // 파일 시작으로 이동(rewind()와 같음.)
  - fseek(fd, 100L, SEEK_SET); // 파일 시작에서 100바이트 뒤로 이동
  - fseek(fd, 0L, SEEK_END); // 파일 끝으로 이동
- 레코드 단위의 현재 파일 위치의 이동 예
  - fseek(fd, n * sizeof(record), SEEK_SET);
    - n+1번째 레코드의 시작 위치로 이동
  - fseek(fd, sizeof(record), SEEK_CUR);
    - 다음 레코드의 시작 위치로 이동
  - fseek(fd, -sizeof(record), SEEK_CUR);
    - 전 레코드의 시작 위치로 이동
- 파일 끝 뒤로도 이동 가능
  - fseek(fd, sizeof(record), SEEK_END);
    - 파일 끝에서 한 레코드 다음 위치로 이동

파일 끝 뒤로의 이동 예

- 빈 파일을 쓰기 모드로 열고, 2개의 레코드(record1, record2)를 파일에 저장한다. 그 다음, fseek() 함수를 이용해 파일 끝에서 한 레코드 만큼 다음 위치로 이동한 후, 1개의 레코드(record3)를 저장하였다.
  - 코드

```
fwrite(&record1, sizeof(record), 1, fp);
fwrite(&record2, sizeof(record), 1, fp);
fseek(fd, sizeof(record), SEEK_END);
fwrite(&record3, sizeof(record), 1, fp);
```

  - 결과



  - 파일 끝을 지나 현재 파일 위치를 자유롭게 이동시키고 데이터를 저장할 수 있는 기능은 **레코드를 계산된 위치에 저장하는 데 매우 유용함**.

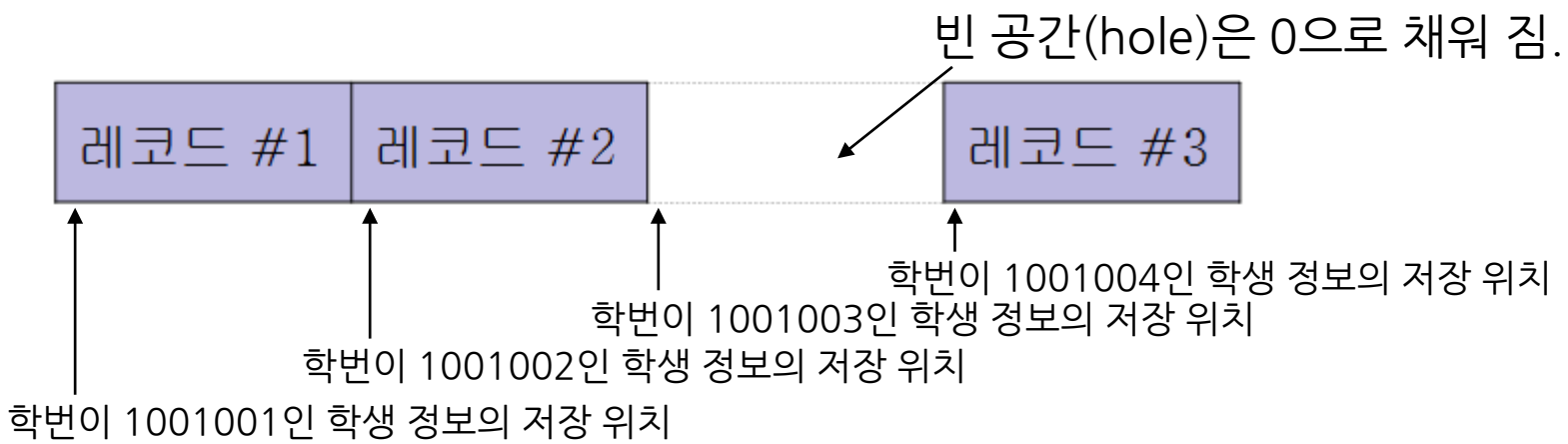
순차 접근의 단점

- 앞에서 학습한 학생 정보 입출력 프로그램들은 학생 정보를 입력된 순서대로 순차적으로 접근하여 처리하였음.
  - 프로그램 4.4~4.5
    - 텍스트 파일 형태로 학생 정보를 파일에 저장하고 읽음.
  - 프로그램 4.6~4.7
    - 이진 파일 형태로 학생 정보를 파일에 저장하고 읽음.
- 순차 접근을 이용할 경우, 특정 학생의 데이터를 검색하기 위해서는 모든 데이터를 처음부터 하나씩 순차적으로 읽으면서 해당 학생의 데이터인지 검사해야 함.
  - 최악의 경우, 파일 내의 모든 학생의 정보를 다 읽어 보아야 함.

임의 접근을 이용한 프로그램

- 프로그램 4.8은 블록 단위 입출력과 임의 접근을 통해 특정 학생의 데이터를 학번을 이용하여 바로 검색할 수 있게 함.
  - 학생 정보의 저장 위치를 시작 학번(START_ID)과 학번(rec.id)을 이용하여 정하였음.

```
#define START_ID 1001001
scanf("%d %s %d", &rec.id, rec.name, &rec.score);
fseek(fp, (rec.id - START_ID) * sizeof(rec), SEEK_SET);
fwrite(&rec, sizeof(rec), 1, fp);
```



stcreate2.c

```c
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "사용법: %s 파일이름₩n",argv[0]);
        return 1;
    }
    FILE * fp = fopen(argv[1], "wb");
    printf("%-9s %-7s %-4s₩n", "학번", "이름", "점수");

    struct student rec;
    while (scanf("%d %s %d", &rec.id, rec.name, &rec.score) == 3) {
        fseek(fp, (rec.id - START_ID) * sizeof(rec), SEEK_SET);
        fwrite(&rec, sizeof(rec), 1, fp);
    }
    fclose(fp);
    return 0;
}
```

저장할 위치로 이동함.

변수 rec에서 sizeof(rec)만큼의 데이터를 1번 읽어와서 열린 파일 fp에 저장함.

student.h

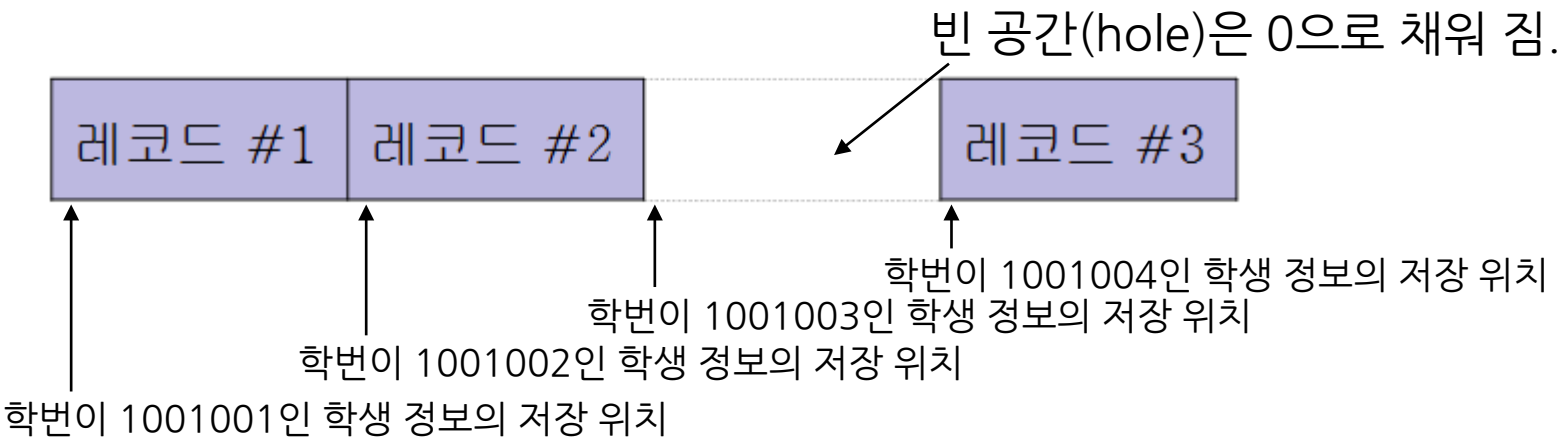
```c
#define START_ID 1001001
struct student {
    int id;
    char name[20];
    int score;
};
```

실행 결과

```
cju@cju-VirtualBox:~/chap4$ gcc -o stcreate2 stcreate2.c
cju@cju-VirtualBox:~/chap4$ ./stcreate2 stdb2
학번     이름  점수
1001001 박연아 96
1001003 김태환 85
1001006 김현진 88
1001009 장샛별 75
cju@cju-VirtualBox:~/chap4$
```

파일 읽기 시 주의 사항

- 프로그램 7.8에서 블록 입출력과 임의 접근을 통해 작성한 이진 파일은 중간 중간에 0으로 채워진 빈공간이 있을 수 있음.
  - 입력한 적이 없는 학생 데이터가 저장될 공간은 건너 띄고 저장하기 때문에 그 부분은 0이 채워 짐.
- 따라서 파일에서 읽어 온 데이터의 내용이 0이라면(예를 들어, **rec.id == 0**) 입력한 적이 없는 데이터이고, 0이 아니면 (**rec.id != 0**) 입력한 적이 있는 데이터임을 알 수 있음.



stquery.c (1/2)

```
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "사용법: %s 파일이름₩n", argv[0]);
        return 1;
    }

    FILE *fp=NULL;

    if ((fp = fopen(argv[1], "rb")) == NULL ) {
        fprintf(stderr, "파일 열기 오류₩n");
        return 2;
    }

    char c;
```

stquery.c (2/2)

```
    do {
        printf("검색할 학생의 학번 입력: ");
        struct student rec;
        int id;
        if (scanf("%d", &id) == 1) {
            fseek(fp, (id - START_ID) *sizeof(rec), SEEK_SET);
            if ((fread(&rec, sizeof(rec), 1, fp) > 0)  &&  (rec.id != 0))
                printf("학번: %8d 이름: %4s 점수: %4d₩n", rec.id, rec.name,
    rec.score);
            else printf("레코드 %d 없음₩n", id);
        }
        else printf("입력 오류");

        printf("계속하겠습니까?(Y/N)");
        scanf(" %c", &c);
    } while (c == 'Y');
    fclose(fp);
    return 0;
}
```

읽기가 성공 했고, 읽어 온 값에서 rec.id가 0이 아니라면

읽어 올 위치로 이동함.

읽어 온 값을 화면에 출력함.

읽어 온 값이 없거나, 읽어 온 값에서 rec.id가 0이라면, 문자를 출력함.

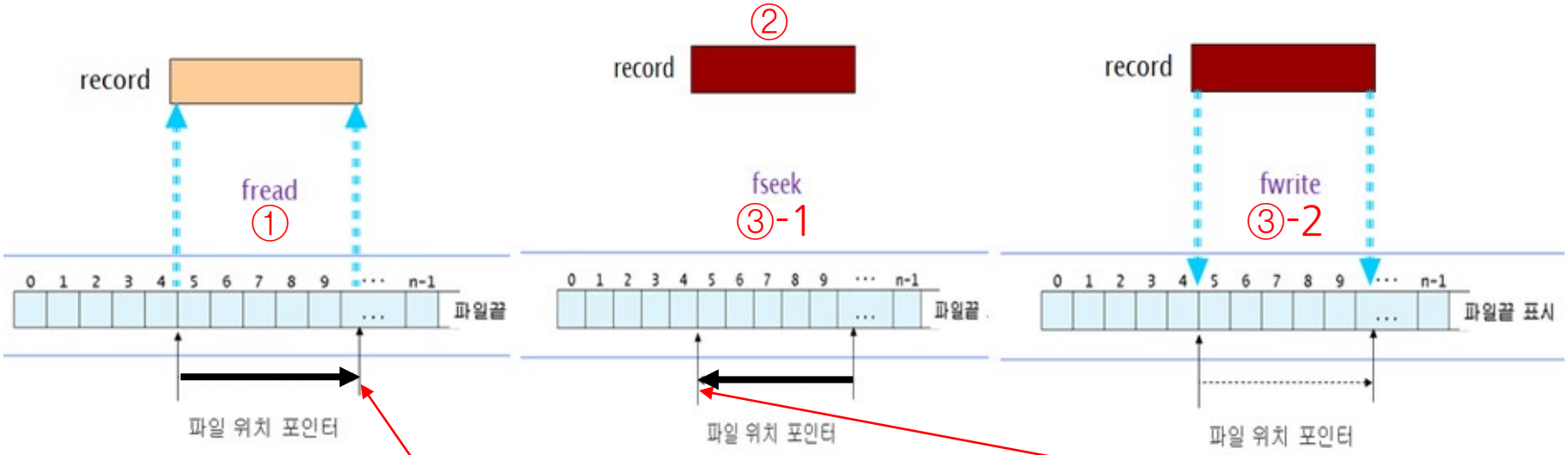
학번 1개가 제대로 입력되지 않았다면 문자를 출력함.

실행 결과

```
cju@cju-VirtualBox:~/chap4$ gcc -o stquery stquery.c
cju@cju-VirtualBox:~/chap4$ ./stquery stdb2
검색할 학생의 학번 입력: 1001003
학번:  1001003 이름: 김태환 점수:   85
계속하겠습니까?(Y/N)Y
검색할 학생의 학번 입력: 1001006
학번:  1001006 이름: 김현진 점수:   88
계속하겠습니까?(Y/N)N
cju@cju-VirtualBox:~/chap4$
```

파일의 특정 레코드 수정 시 주의 사항

- 파일에 저장된 레코드를 수정하는 과정
  ① 파일로부터 해당 레코드를 읽음.
  ② 이 레코드를 수정함.
  ③ 수정된 레코드를 파일의 **원래 위치에 저장**해야 함.



fread() 함수로 레코드를 읽으면 현재 파일 위치가 다음 레코드의 시작 위치로 이동함.

따라서 수정된 레코드를 파일에 쓰기전에 반드시 현재 파일 위치를 **원래 위치로 이동시킨 후 저장**해야 함.

① fread(&rec, sizeof(rec), 1, fp);

③-1 fseek(fp, -sizeof(rec), SEEK_CUR);
③-2 fwrite(&rec, sizeof(rec), 1, fp);

stupdate.c (1/2)

```
#include <stdio.h>
#include "student.h"
int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "사용법: %s 파일이름₩n", argv[0]);
        return 1;
    }

    FILE *fp=NULL;

    if ((fp = fopen(argv[1], "rb+")) == NULL ) {
        fprintf(stderr, "파일 열기 오류₩n");
        return 2;
    }

    char c;
```

stupdate.c (2/2)

```
do {
    printf("수정할 학생의 학번 입력: ");
    struct student rec;
    int id;
    if (scanf("%d", &id) == 1) {
        fseek(fp, (id - START_ID) *sizeof(rec), SEEK_SET);
        if ((fread(&rec, sizeof(rec), 1, fp) > 0)  &&  (rec.id != 0)){
            printf("학번: %8d 이름: %4s 점수: %4d₩n", rec.id, rec.name, rec.score);
            printf("새로운 점수 입력: ");
            scanf("%d", &rec.score);
            fseek(fp, -sizeof(rec), SEEK_CUR);
            fwrite(&rec, sizeof(rec), 1, fp);
        }
        else printf("레코드 %d 없음₩n", id);
    }
    else printf("입력 오류");
    printf("계속하겠습니까?(Y/N)");
    scanf(" %c", &c);
} while (c == 'Y');
fclose(fp);
return 0;
}
```

읽기가 성공 했고,
읽어 온 값 중 rec.id가 0이 아니라면

읽어 온 값을 화면에 출력함.

변수 rec의 점수(score)만 키보드로 새로 입력 받음.

현재 파일 위치를 rec 크기만큼 앞으로 이동시킴.

변수 rec의 값을 파일에 저장함.

실행 결과

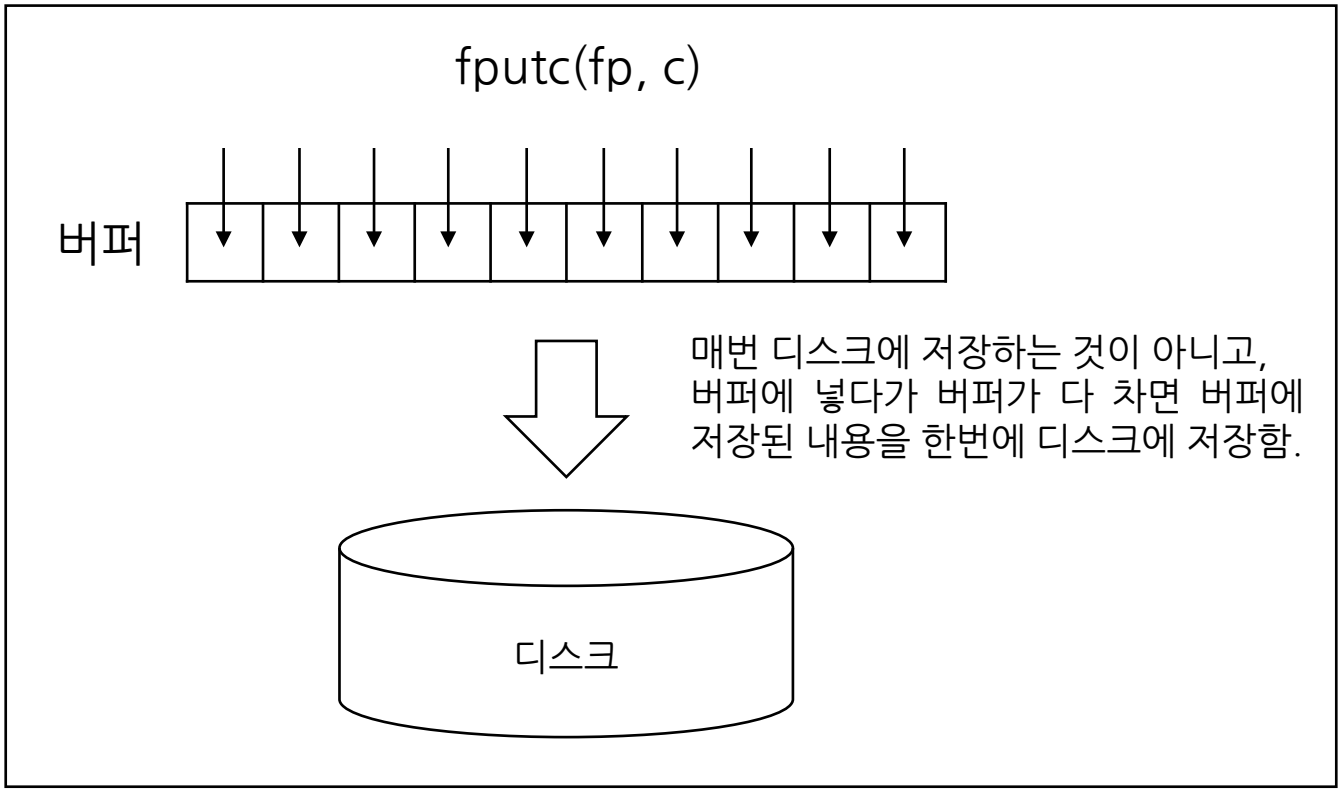
```
cju@cju-VirtualBox:~/chap4$ gcc -o stupdate stupdate.c
cju@cju-VirtualBox:~/chap4$ ./stupdate stdb2
수정할 학생의 학번 입력: 1001009
학번:  1001009 이름: 장샛별 점수:   75
새로운 점수 입력: 85
계속하겠습니까?(Y/N)N
cju@cju-VirtualBox:~/chap4$
```

질문

Q&A

버퍼 입출력

- C 파일 입출력 라이브러리 함수들은 입출력을 최적화하기 위해서 버퍼(buffer)를 할당하여 사용함.
- 버퍼를 사용함으로써 실제 디스크에 입출력하는 횟수를 획기적으로 줄일 수 있음.



버퍼 방식

- 완전 버퍼 방식
  - 버퍼가 꽉 찼을 때 실제 디스크 입출력을 수행하는 방식
  - 주로 파일 입출력에 사용됨.
- 줄 버퍼 방식
  - 개행 문자(‘₩n’)를 만났을 때 실제 디스크 입출력을 수행하는 방식
  - 표준 입출력(stdin과 stdout)이 대표적인 줄 버퍼 방식임.
- 버퍼 미사용 방식
  - 입출력에 버퍼를 사용하지 않는 방식
  - 표준 에러(stderr)가 대표적인 버퍼 미사용 방식임.

buffer.c (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define _IO_UNBUFFERED     0x0002
#define _IO_LINE_BUF       0x0200
int main(int argc, char *argv[])
{
    FILE *fp=NULL;
    if (!strcmp(argv[1], "stdin")) {
        fp = stdin;
        printf("한 글자 입력:");
        if (getchar() == EOF) perror("getchar");
    }
    else if (!strcmp(argv[1], "stdout"))
        fp = stdout;
    else if (!strcmp(argv[1], "stderr"))
        fp = stderr;
    else if ((fp = fopen(argv[1], "r")) == NULL) {
        perror("fopen");
        exit(1);
    }
    else if (getc(fp) == EOF) perror("getc");
```

← 빈 파일일 때

buffer.c (2/2)

```
    printf("스트림 = %s, ", argv[1]);

    if (fp->_flags & _IO_UNBUFFERED)
        printf("버퍼 미사용");
    else if (fp->_flags & _IO_LINE_BUF)
        printf("줄 버퍼 사용");
    else
        printf("완전 버퍼 사용");

    printf(", 버퍼 크기 = %ld₩n", fp->_IO_buf_end - fp->_IO_buf_base);
    exit(0);
}
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o buffer buffer.c
cju@cju-VirtualBox:~/chap4$ ./buffer stdin
한 글자 입력:a
스트림 = stdin, 줄 버퍼 사용, 버퍼 크기 = 1024
cju@cju-VirtualBox:~/chap4$ ./buffer stdout
스트림 = stdout, 줄 버퍼 사용, 버퍼 크기 = 1024
cju@cju-VirtualBox:~/chap4$ ./buffer stderr
스트림 = stderr, 버퍼 미사용, 버퍼 크기 = 0
cju@cju-VirtualBox:~/chap4$ ./buffer /etc/passwd
스트림 = /etc/passwd, 완전 버퍼 사용, 버퍼 크기 = 4096
cju@cju-VirtualBox:~/chap4$
```

버퍼 사용의 on/off

- setbuf() 함수를 이용하여 버퍼 사용을 on/off 할 수 있음.
- 함수 프로토타입
  - void setbuf(FILE *fp, char *buf);
    - fp는 열린 파일을 나타냄.
    - buf는 길이가 BUFSIZ(8192)인 배열의 주소(배열의 이름) 또는 NULL을 지정함.
- 사용 예

```
setbuf(fp, NULL);          ← fp의 버퍼 사용 off

char buf[BUFSIZ];
setbuf(stdout, buf);       ← stdout의 버퍼를 buf로 지정
```

setbuf.c

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

int main()
{
    printf("안녕하세요, "); sleep(1);
    printf("리눅스입니다!"); sleep(1);
    printf(" ₩n"); sleep(1);

    setbuf(stdout, NULL);          ← 버퍼 미사용으로 전환
    printf("여러분, "); sleep(1);
    printf("반갑습니다"); sleep(1);
    printf(" ^^"); sleep(1);
    printf(" ₩n"); sleep(1);

    return 0;
}
```

```
cju@cju-VirtualBox:~/chap4$ gcc -o setbuf setbuf.c
cju@cju-VirtualBox:~/chap4$ ./setbuf
안녕하세요, 리눅스입니다!
여러분, 반갑습니다 ^^
cju@cju-VirtualBox:~/chap4$
```

버퍼의 다양한 설정

- setvbuf() 함수를 이용하여 버퍼의 사용 방식과 버퍼의 크기를 다양하게 설정할 수 있음.
- 함수 프로토타입
  - int setvbuf(FILE *fp, char *buf, int mode, size_t size);
    - fp는 열린 파일을 나타냄.
    - buf는 버퍼로 사용할 배열의 주소(배열의 이름)를 나타냄.
    - mode는 버퍼 사용 방식을 나타냄.
      - _IOFBF: 완전 버퍼, _IOLBF: 줄 버퍼, _IONBF: 버퍼 미사용
    - size는 버퍼의 크기(buf 배열의 길이)를 나타냄.
    - 버퍼 변경이 성공하면 0을 반환함.
- 사용 예

```
setvbuf(fp, NULL, _IONBF, 0);
```

← fp의 버퍼 사용 off, 즉, 버퍼 미사용

```
char buf[512];
setvbuf(fp, buf, _IOLBF, 512);
```

← fp의 버퍼로 buf 배열을 이용하고 줄 버퍼 방식을 사용함.

setvbuf.c

```
#include <stdio.h>

int main()
{
    FILE *fp1 = fopen("data1", "a");
    FILE *fp2 = fopen("data2", "w");
    char buf[1024];

    if(setvbuf(fp1, buf, _IOFBF, sizeof(buf)) != 0)
        printf("첫 번째 스트림: 잘못된 버퍼₩n" );
    else
        printf("첫 번째 스트림: %ld 바이트 크기 버퍼 사용₩n",
                fp1->_IO_buf_end - fp1->_IO_buf_base);

    if(setvbuf(fp2, NULL, _IONBF, 0) != 0)
        printf("두 번째 스트림: 잘못된 버퍼₩n" );
    else
        printf("두 번째 스트림: 버퍼 미사용₩n" );

    return 0;
}
```

← setvbuf() 함수는 성공하면 0을 반환함.

← setvbuf() 함수는 성공하면 0을 반환함.

```
cju@cju-VirtualBox:~/chap4$ gcc -o setvbuf setvbuf.c
cju@cju-VirtualBox:~/chap4$ ./setvbuf
첫 번째 스트림: 1024 바이트 크기 버퍼 사용
두 번째 스트림: 버퍼 미사용
```

기타 버퍼 관련 함수

- fflush() 함수는 열린 파일의 버퍼를 비움.
- 함수 프로토타입
  - int fflush(FILE *fp);
    - fp의 버퍼가 출력 버퍼이면, 버퍼에 남아 있는 데이터를 모두 파일에 저장함.
    - fp의 버퍼가 입력 버퍼(stdin)이면, 버퍼를 비움.
    - fp가 NULL이면 모든 열린 파일들에 대하여 작업을 수행함.

기타 함수

- C 표준 라이브러리는 문자 및 문자열 처리와 관련된 다양한 함수들을 제공함.
- 교재 195쪽~197쪽의 주요 문자 및 문자열 처리 함수 목록을 읽고 다양하게 응용해 보시길 바랍니다.

질문

Q&A