입력(Input)

2020년 12월 10일 목요일 오후 9:32

입력

- 출력의 반대
- 외부의 데이터를 읽어와서 프로그램에서 사용
- 어떤 데이터가 들어올지 몰라서 괴상한 데이터가 종종 들어옴
  - 사용자가 잘못된 데이터를 키보드에서 입력
  - 예전에 저장해 놓은 파일을 누가 잘못 바꿨거나 일부 데이터가 유실

입력은 출력보다 까다롭다

- 출력에 비해 조심해야 할 일이 많음
- 데이터 읽기에 실패했는지 제대로 처리 안하면 팡팡 터짐
  - 정말 많이들 실수하는 부분
- 그래서 모든 입력 함수에는 반환값이 있음
  - 데이터 읽기에 실패했는지 성공했는지 반환
- 따라서 어떤 함수가 어떤 값을 반환하는지 문서에서 확실히 읽고 코드에서 검사할 것
  - 대부분의 입력처리 코드의 문제는 반환값이 뭔지 문서를 제대로 안 읽어봐서 발생

입력의 출처는 어디?

- 입력은 어디에서부터 읽어올까?
- 어딘가에 출력을 했다면 거기에서 읽어올 수도 있다고 생각하면 편함
  - 그것은 바로 스트림
- 스트림
  - 콘솔 창에 출력했으니 콘솔(키보드)로부터 입력받아 옴
  - 파일에 출력(저장)했으니 파일로부터 입력받아(읽어) 옴
  - 등등
- 문자열
  - 문자열에 출력(저장)했으니 문자열로부터 입력받아(읽어)옴

입력처리 전략

- 크게 4가지의 전략이 있음

1. 한 글자씩 읽기
   - 여러 개의 문자를 입력하고 한 글자씩 읽는 방법
2. 한 줄씩 읽기
   - 데이터가 여러 줄이 있을 때 한 줄씩 읽는 방법
3. 한 데이터씩 읽기
   - 어떤 긴 데이터가 있으면 거기서 문자열도 하나 읽어오고, 문자도 하나 읽어오고, 부동 소수점도 하나씩 읽어오고 이런식으로 각 데이터 형에 맞게 읽는 방법
4. 한 블록씩 읽기 (이진 데이터)
   - 바이너리 파일에서 int 정수를 한 번에 읽어오거나, 배열을 한꺼번에 저장하고 한꺼번에 읽어올 수도 있음

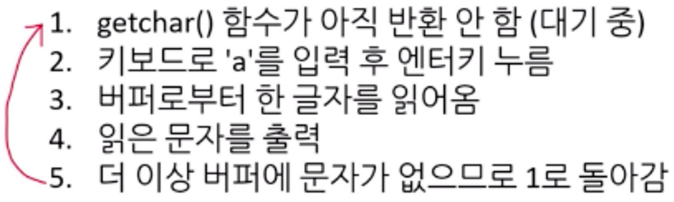
한 글자씩 읽기

한 글자씩 읽는 알고리듬1

1. 한 글자(char)를 읽어온다
2. 그 글자를 필요한 곳에 사용한다
3. 1번 단계로 되돌아간다

(매우 단순화시킨 예)

```
int c;

while (TRUE) {
    c = getchar();
    putchar(c);
}
```



- a입력 후 엔터를 치면 버퍼에는 a와 ₩n(개행문자)가 들어있다.
  a와 개행문자까지 출력하기 위해 3,4번을 두번 반복함
  - Putchar(c)를 실행하고 다시 한번 더 c = getchar():로 돌아간다

    - 이 알고리듬의 문제: 반복문에서 나갈 방법이 없음
- getchar( ) 반쪽자리 설명
    - `int getchar(void);`
    - `int fgetc(FILE* stream);`
        - 키보드(stdin)으로부터 문자를 하나 읽어서 int 형으로 반환
            - int형으로 반환하는 이유는 -1(EOF)을 반환하기 위해
                - EOF임을 알리는 것임
            - 많은 입출력 함수들이 문자를 읽고 쓸 때 char대신 int를 쓴다
        - fgetc(stdin)은 getchar( )와 같다
- getchar( )의 반환값

> **Return value**
> The obtained character on success or EOF on failure.
> If the failure has been caused by end-of-file condition, additionally sets the *eof* indicator (see feof()) on stdin. If the failure has been caused by some other error, sets the *error* indicator (see ferror()) on stdin.
>
> (Source: https://en.cppreference.com/w/c/io/getchar)

    - 성공하면 문자를, 실패하는 EOF(end-of-file)를 반환
- 입력의 끝을 나타내는 값 EOF
    - C 표준에 의하면 EOF는 음수라고 함
    - 그런데 char는 표준에 따르면 부호가 있을 수도 있고 없을 수도 있음
    - 그래서 char에 EOF를 담는것은 부적절함
    - 이것이 getchar( )가 int를 반환하는 이유
- 완전한 getchar( ) 설명
    - `int getchar(void);`
    - `int fgetc(FILE* stream);`
        - 키보드(stdin)으로부터 문자를 하나를 읽음
        - 반환값
            - 성공 시, 읽은 문자(의 아스키코드)를 반환
            - 실패시, EOF를 반환
        - fgetc(stdin)과 getchar( )는 같음
- 한 글자씩 읽는 알고리듬2

1. 한 글자(char)을 읽어온다
2. 글자를 읽어오는데 실패했다면(EOF) 프로그램을 종료
3. 아니라면 그 글자를 필요한 곳에 사용한다
4. 1번 단계로 되돌아 간다

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

```c
int c;

c = getchar();

while (c != EOF) {
    putchar(c);
    c = getchar();
}
```

- EOF키는 뭘까?
    - ctrl 키와 다른 키를 조합해서 넣음
        - 윈도우: ctrl + z
        - 리눅스, 맥: ctrl + d
- 알고리듬2 에서 getchar()중복이 거슬릴 경우
    - getchar( )가 두번 있음
    - do while로 해결 가능할까?
        - 안됨. while 검사 전에 c를 사용하기 때문
    - 한번으로 줄이는 방법

```c
int c;

while ((c = getchar()) != EOF) {
    putchar(c);
}
```

        - != 연산자가 = 연산자보다 결합 우선순위가 높아서 괄호를 침
            - 괄호를 안 치면 c = (getchar() != EOF) 이렇게 되버림
                - 이 경우 c는 0또는 1이 됨
                - 이런식으로 한 줄로 줄이는 건 실수하기가 쉽다
                  그래서 요즘 다른 언어에서는 잘 하지 않는 방법
- 한 글자씩 읽는 알고리듬의 장점
    1. 가장 간단한 입력 방법
    2. 입력이 문자/문자열일 때 매우 좋음
    3. 쓸데없이 메모리에 입력값을 저장해 두지 않아도 됨
        - 글자 하나 읽와서 글자 하나 처리하고 바로 버림
            - 글자를 읽어와서 배열에 쌓아두고 쓰는 것이 아님
            - 배열을 쓰면 배열 범위 신경쓰고 실수할 가능성 높음
        - 용량을 절약하고 실수도 줄임

    4. 복잡도가 O(N)이다!
        - for문 딱 한 번만 도는 알고리듬에 적합한 경우가 많음
            - 키보드로부터 한 글자씩 읽어서 곧바로 처리
    5. 다른 데이터형으로 쓰기는 좀 어려움
        - 예: 정수형 숫자 1004를 읽기
            - '1', '0', '0', '4' 이렇게 4번 읽어서 그걸 정수로 변환하기가 어려움

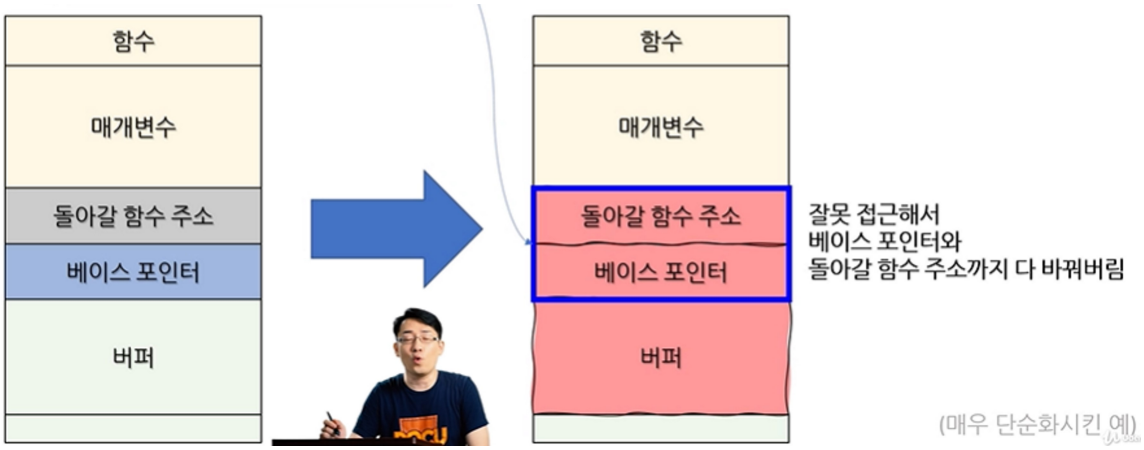
한 줄씩 읽기

- 한 줄씩 읽는 알고리듬
    1. 한 줄을 읽어온다
    2. 한 줄을 읽어오는데 실패하면 프로그램을 종료
    3. 성공했다면 한 줄 읽어온 데이터를 필요에 따라 사용
    4. 1번 단계로 되돌아간다
- 한 줄을 읽는 방법
    - 한 줄을 어떻게 읽어야 할까?
        - 한 줄을 읽어오면 어디에다 저장해야 할까?
        - 함수가 새로운 문자열을 반환해줄까?
        - 아니다. 프로그래머가 미리 만든 배열을 함수에 전달해야 함
          함수는 그 배열에 한줄을 읽어 온다.
- gets( )
    - `char* gets(char* str);`
    - stdin에서 새 줄 문자(₩n)또는 EOF를 만날 때까지 계속 문자들을 읽어서 str 배열에 저장
        - 그래서 str이 const가 아님
    - 마지막 문자 바로 다음에 널 문자(₩0)도 넣어 줌
    - stdin에서 새 줄 문자(₩n)를 제거함
        - 버퍼에 저장하지는 않음
    - 반환되는 것은?
        - 성공 시, str
        - 실패 시, NULL(널 포인터)
- gets( )의 위험성
    - gets( )는 매우매우 위험한 함수
        - C11는 아예 gets( )를 제거해버림
        - 그래서 최신 헤더파일에는 gets( )가 더 이상 존재하지 않음
            - 굳이 쓰려면 직접 함수 원형을 전방 선언하면 됨
    - 왜 위험할까?
        - 함수에 전달한 배열의 길이 이상 입력하면 버퍼 오버플로 발생
    - 왜 버퍼 오버플로가 발생?

```c
#define LINE_LENGTH (10)

void print_my_input(void)
{
    char line[LINE_LENGTH];

    gets(line);
    puts(line);
}

int main(void)
{
    print_my_input();
    printf("Well done\n");

    return 0;
}
```

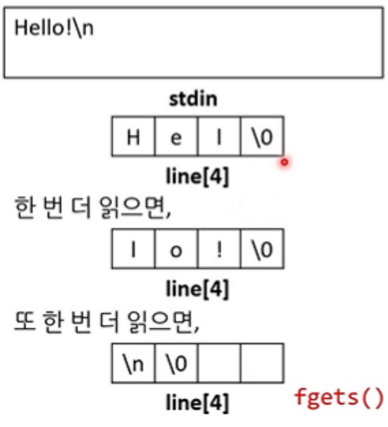
```
D:\COMP2200\input_test\input_test>a.exe
okay I have to enter more than 10 characters for testing
okay I have to ente
D:\COMP2200\input_test\input_test>
```

- 올바르지 않은 메모리 주소에 키보드로 입력받은 값을 써버림



- 이를 이용한 보안적인 공격도 있음
  - 버퍼 오버플로우 어택
    - 버퍼를 오버플로우 해서 함수의 중요한 변수에 접근을 하는 공격
      - 사용자가 올바른 권한이 있는지 확인하는 코드
      - int auth; /* 패스워드가 인증되면 1, 아니면 0*/
        char pw[Length];
        gets(pw);
        if (auth) {
        /* 올바른 패스워드면 실행 */
        }
      - 패스워드를 받으려고 gets(pw)를 사용했는데 버퍼 오버플로를 일으켜서 스택메모리에 auth가 저장된 메모리를 1로 덮어쓸 수 있다.

- fgets( )로 안전하게 한 줄 읽기
  - `char* fgets(char* str, int count, FILE* stream);`
    - <stdio.h> 안에 있음
    - 최대 count -1개의 문자열을 읽어서 str에 저장
      - 남은 한 개는 널 문자(₩0)를 저장
    - 즉, 새 줄을 만나지 만나지 않아도 이 함수가 반환될 수 있음
    - str에 새 줄 문자(₩n)까지 넣어 줌
      - 새 줄을 만나서 끝났을 때랑 아닐 때를 구분해주기 위해서
    - fgets( )의 매개변수
      - str: 입력받은 한 줄을 저장할 char 배열
      - count: 한 번에 str에 쓰는 최대 문자 수
        - 널 문자를 포함하기 때문에 실제로 읽어오는 문자 수는 count - 1개

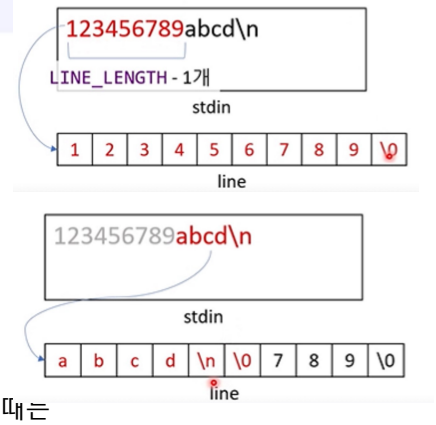


        - stream: 데이터를 읽어올 스트림. 키보드 입력을 읽어오고 싶다면 stdin을 넣어주면 됨
    - fgets( )의 반환값
        - 성공 시, str을 반환
        - 실패 시, NULL을 반환
- fgets()의 매개변수: FILE 자료형
    - 스트림을 제어하기 위해 필요한 정보를 담고 있는 자료형
        - 파일 위치 표시자
        - 스트림이 사용하는 버퍼의 포인터
        - 읽기/쓰기 중에 발생한 오류를 기록하는 오류 표시자
        - 파일의 끝에 도달했음을 기록하는 EOF 지시자
    - 이 정보들을 담기 위해 구조체로 구현되어있음
    - 플랫폼마다 이 자료형을 구현하는 방식은 다를 수 있음
    - 입력 및 출력 스트림은 오직 FILE 포인터로만 접근 및 조작가능
        - 다른 스트림도 모두 표현 가능
- 한줄 읽기 예

```c
#include <stdio.h>

#define LINE_LENGTH (10)

char line[LINE_LENGTH];

while (fgets(line, LINE_LENGTH, stdin) != NULL) {
  printf("%s", line);
}
```

    - fgets( )에 쓸 버퍼는 초기화 필요 없음

1. 입력 스트림에서 LINE_LENGTH - 1개 만큼 읽음
   line의 마지막 요소에는 ₩0을 넣어 줌
2. 아직 입력 스트림에 남아있는 문자들을 읽음
   읽는 중간에 새 줄 문자(₩n)을 만남
   새 줄 문자까지의 문자열을 line에 넣음



- 한 줄씩 읽는 방법이 유용한 경우
    - 단어 하나씩 읽는 것보단 한 줄씩 읽는 게 빠름
    - CPU를 벗어나 외부 구성요소로부터 뭔가를 읽어올 때는
      한 번에 많이 읽어오는 게 빠름
    - 따라서 버퍼 크기는 충분히 큰 게 좋다
        - 버퍼 오버플로우는 없어야 함

한 데이터씩 읽기

- 내가 원하는 데이터 형식으로 읽는 것
  한 int 씩, 한 부동소수점형 식, 한 문자씩 등 내 맘대로 정할 수 있음
- 세 가지 버전이 있음

1. scanf( ): stdin으로부터 읽음

```c
int scanf(const char* format, ...);
```

2. fscanf( ): 파일 스트림으로부터 읽음

```c
int fscanf(FILE* stream, const char* format, ...);
```

3. sscanf( ): C 스타일 문자열로부터 읽음

```c
int sscanf(const char* buffer, const char* format, ...);
```

    - scanf()와 sprintf()는 파일스트림이나 콘솔에 입출력하는게 조금 느리니까
      빠르게 스택 내부 문자열에 입출력하려고 쓰이는 경우가 많다

- scanf( )

```
int scanf(const char* format, ...);
```

  - <stdio.h>에 있음
  - 키보드(stdin)로부터 입력을 받아 변수에 저장
  - 반환값
    - 몇 개의 데이터를 읽었는지 반환
    - 첫 데이터를 읽기 전에 실패했다면 EOF를 반환
- 저장할 변수의 주소를 전달하는 이유

```
scanf("%d", &num);
```

  - 그냥 num을 넣으면 복사된 매개변수를 전달(값에 의한 전달)임
    함수 속에서 바꿔봐야 반환시 사라짐
- 그냥 scanf( )에서 값을 반환하면 안될까?
  - 불가능
    - 읽는 타입에 따라 함수이름이 달라져야 함
      - int scanf_int( );
      - char scanf_char( );
      - float scanf_float( );
      - char* scanf_string( );
  - 이름을 다르게 짓는다 해도 반환값이 여러 개 있으면 문제가 됨
    - scanf("%d %d", &num1, &num2);

scanf( )의 서식 문자열

- 일반적인 서식 문자열 형식

```
%[*][너비][길이]서식 지정자
```

  - 일반적으로 % 뒤에 최대 4개의 지정자를 가질 수 있음
    1. *(선택)
    2. 너비 (선택)
    3. 길이 수정자(선택)
    4. 서식 지정자(**필수**)
       - 반드시 순서를 지켜 작성
- scanf( )의 서식 지정자

%	%를 순수하게 문자로 인식	scanf("%%d", &num); /* 컴파일 경고 */
c	문자(char)	scanf("%c", &ch);
s	한 단어	scanf("%s", str);
d	부호 있는 10진수 수	scanf("%d", &num);
x	부호 없는 16진수 수	scanf("%x", &num);
f	부동 소수점(float, double)	scanf("%f", &num);

  - 반드시 넣어야 함
  - 모든 데이터는 한 단어씩(공백 문자로 구분) 또는 가능할 때까지 읽음
  - 공백 문자는 버림
    - %c는 공백문자도 입력
    - 공백문자는 단어를 구분하기위해 있는 것

② 대입 생략 문자 *

  - assignment-suppressing character라고 함
  - 이 문자를 쓸 경우 키보드로부터 받은 입력을 변수에 저장하지 않음

```c
int num;

printf("Enter a number: ");
scanf("%*d%d",&num);
printf("num = %d\n", num);
```

```
Enter a number: 10 20
num = 20
```

- 거의 쓸 일이 없다고 함

② 너비

정수	읽을 최대 문자 수	scanf("%5s", str); scanf("%10d\n", &num);

- %s의 경우 너비를 지정 안 하면 버퍼 오버플로가 날 수 있음
- 이를 방지하기 위해 문자열의 너비를 서식문자를 통해 정해 줌
  - 하지만 이렇게 쓰는 경우는 드물고 다른 방식을 사용한다
- 너비 지정시 주의할 점
  - 너비를 지정 후에 여러 데이터를 한 번에 읽을 경우 이런 문제 발생

```c
scanf("%3d %3d", &num1, &num2);
printf("%d, %d\n", num1, num2);
```

```
123456789 987654321
123, 456
```

123456789 987654321 일부 사용

  - 한 정수를 3자리 읽고 그 뒤에 정수 3자리를 읽고 싶었는데
    너비 지정자를 사용하는 경우 한 정수를 3자리 읽고 같은 정수의 뒤 3자리를 읽는다
    - 원하는 결과가 아닌 경우이다.

② 길이 수정자(length modifier)

길이 수정자	서식 지정자		
	d	int*	scanf("%d\n", &number);
l	d	long int*	scanf("%ld\n", &number);
	f	double*	scanf("%f\n", &number);
L	f	long double*	scanf("%lf\n", &number);

- 인자의 바이트 크기를 지정해 준다
  - 출력함수때와 마찬가지로 별 의미는 없다
    - 요즘 운영체제에서 int == long int인 경우와 double == long double인 경우가 많다

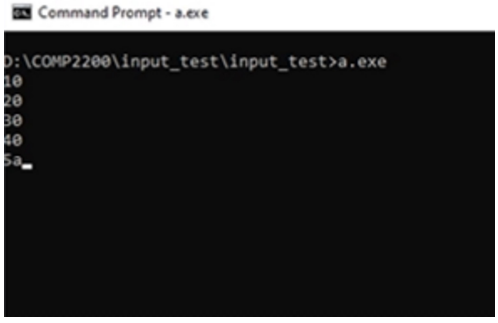
- scanf( ) 사용 예
  - https://www.udemy.com/course/c-unmanaged-programming-by-pocu/learn/lecture/16254382#content

문자를 읽을 때 scanf( )의 문제점과 해결책, clearerr( )

- %s 쓸 때 배열 크기보다 큰 문자열이 들어오면 버퍼 오버플로

```c
char str[16];

scanf("%s", str); /* 콘솔창에서 "123456789abcdefghij" 입력 */
```

- 다른 자료형 읽을 때 무한 루프에 빠지는 위험




```c
int num;
int sum = 0;

while (TRUE) {
    if (scanf("%d", &num) == 0) {
        printf("Error!\n");
        continue;
    }

    if (num == 0) {
        break;
    }

    sum += num;
}

printf("Sum: %d\n", sum);
```

  - 5a 입력 후 계속 'Error!'만 무한반복 출력됨
  - 정수(int)만 읽으려 했는데 문자(char)가 있어서 읽기 실패
  - 문자a가 계속 입력 스트림에 남아있으니 무한루프에 빠짐
  - 해결법?

- 앞에서 배웠던 fgets( )와 sscanf( )함수를 같이 쓰는게 좋음

- 무한 루프 문제 없이 숫자 읽기 예

```
#define LINE_LENGTH (1024)
```

```
int sum = 0;
int num;
char line[LINE_LENGTH];

while (TRUE) {
    if (fgets(line, LINE_LENGTH, stdin) == NULL) {
        clearerr(stdin);
        break;
    }

    if (sscanf(line, "%d", &num) == 1) {
        sum += num;
    }
} /* 출력 코드는 생략 */
```

- 버퍼 오버플로 문제 없이 문자열 읽기

```
#define LENGTH (4096)
```

```
char line[LENGTH];
char word[LENGTH];

while (TRUE) {
    if (fgets(line, LENGTH, stdin) == NULL) {
        clearerr(stdin);
        break;
    }

    if (sscanf(line, "%s", word) == 1) {
        printf("%s\n", word);
    }
}
```

  - 최대 4096만큼의 line을 읽어서,
    최대 4096 만큼의 word에 넣으면 오버플로가 발생하지 않을 것임
  - 단, 4096보다 긴 문자열이 들어오면 그냥 짤리는게 전부

② clearerr( )는 뭘까?

×

  - clearerr -> clear error라는 뜻
  - 스트림을 읽거나 쓸 때 EOF를 만나면 그 스트림의 EOF 표시자(indicator)가 세팅 됨
  - 그 외의 이유로 실패하면 오류 표시자(error indicator)를 세팅
    - 두 표시자의 세팅 여부를 확인하고싶으면
      feof( )나 ferror( )함수를 사용하면 됨
  - 그게 잘 안 지워져서 다음에 읽거나 쓸 때 계속 실패할 수 있음
  - 그래서 그 오류를 지워주는 것

- 한 데이터씩 읽는 방법이 유용한 경우
  - 텍스트를 다른 자료형으로 곧바로 읽어오는 가장 간단한 방법
    - 문자열로 적혀있는 정수를 읽어서 따로 반환하는건 번거롭다
  - 사용자 입력 받을 때(그리고 여러 데이터가 혼용된 텍스트파일을 읽어올 때) 가장 많이 쓰는 방법

■ 한 블록씩 읽기

- 여태까지 본 입력 방법은 텍스트로 저장된 데이터를 읽는 것
- C#과 마찬가지로 C에서도 이진 데이터를 읽을 수 있음
  - 그게 바로 한 블록씩 읽는 것

한 블록 읽기

```
```

- size 바이트짜리 데이터를 총 count개수만큼 읽음
- 그래서 buffer에 저장
- EOF 만나면 당연히 멈춤
  - 그렇다면 count보다 적은 수를 읽을 수도 있다는 것
  - 그래서 실제로 읽는 개수를 반환
- 한 블록씩 쓰는 함수도 있음

```
```

- 두 함수의 버퍼가 다른 이유
  - fwrite의 버퍼는 const void*
    - 이 뜻은 버퍼에 있는 데이터를 바꾸지 않겠다는것,
      버퍼에 있는 데이터를 읽어다 스트림에 쓰는 것
  - fread의 버퍼는 void*
    - const가 아니므로 버퍼에 있는 데이터를 수정하겠다는 것
      스트림에서 받은 데이터를 버퍼에 쓰는 것
- 두 함수를 어디에 쓸까?
  - stdin, stdout은 텍스트가 들어오므로 아니다
  - 파일에 쓰고 읽어야 함
    - 그럴려면 '파일 스트림'을 알아야 함
    - 파일 스트림이 있다면 작동하는 코드

      - int 블록을 저장
        - 총 몇 바이트? 64 * sizeof(int)

- 한 블록씩 읽는 방법이 유효한 경우
  - 이진 데이터를 읽기 위해
  - 이진 데이터를 하나씩 읽는 것 보다 한꺼번에 읽는것이 빠르다
- 한 블록씩 읽을 때 주의할 점
  - 기본 데이터형의 크기는 시스템마다 다름
    - int의 크기가 4바이트가 아닐 수도 있다
      - A 시스템 용으로 빌드한 실행파일을 실행해서 파일을 저장
      - B 시스템 용으로 빌드한 실행파일을 실행해서 그 파일을 읽음
      - 그러나 두 시스템 간 바이트 크기가 틀려서 엉뚱한 데이터가 읽힘
        - 4바이트로 저장한 int가 2바이트씩 읽힌다거나
  - 따라서 이런 일을 하려면 정확히 파일에 저장할 데이터 크기를 고정해 두는게 좋음
    - 파일에 저장할 데이터 크기를 'int 바이트 몇 개 들어있다', 'char 바이트 몇 개 들어있다'
      이런 식으로 정해두는게 아니라 '몇 바이트가 몇 개 있다' 라고 확실히 정해두는게 좋음