CHAPTER 4

문자 처리: 문자 분류 및 변환

학습 목표

- 스트림과 데이터의 이동
- 문자열 단위 입출력 함수

스트림과 데이터의 이동

4.1 문자 입출력 함수

```
#include <stdio.h>
int putchar(int c);
int fputc(int c, FILE * stream);
```

putchar 함수는 인자로 전달된 문자를 모니터에 출력한다.

fputc 함수의 두 번째 인자를 통해서 출력의 대상을 지정한다.

➜ 함수호출 성공 시 쓰여진 문자정보가, 실패 시 EOF 반환

fputc의 두 번째 인자로 stdout이 전달되면 이 putchar 함수와 동일한 결과를 보인다.

√ 하나의 문자를 입력 받는 두 함수

```
#include <stdio.h>
int getchar(void);
int fgetc(FILE * stream);
```

키보드로 입력된 문자의 정보를 반환한다.

문자를 입력 받을 대상정보를 인자로 전달한다.

➜ 파일의 끝에 도달하거나 함수호출 실패 시 EOF 반환

getchar 함수와 fgetc 함수의 관계는 putchar 함수와 fputc 함수의 관계와 같다.

4.2 문자 입출력 관련 예제

```
int main(void)
{
    int ch1, ch2;

    ch1=getchar();   // 문자 입력
    ch2=fgetc(stdin);   // 엔터 키 입력

    putchar(ch1);   // 문자 출력
    fputc(ch2, stdout);   // 엔터 키 출력
    return 0;
}
```

문자의 입력을 완성하는 엔터 키의 입력도 하나의 문자로 인식이 된다.
따라서 이 역시도 입출력이 가능하다.

```
p
p
```

실행결과

첫 번째 P는 입력이 된 P, 두 번째 P는 출력된 P

문자를 int형 변수에 저장하는 이유는 EOF를 설명하면서 함께 설명한다.

4.3 문자 입출력에서의 EOF

√ EOF의 의미

- ▶ EOF는 End Of File의 약자로서, 파일의 끝을 표현하기 위해서 정의해 놓은 상수이다.
- ▶ 파일을 대상으로 fgetc 함수가 호출되었을 때 파일에 끝에 도달을 하면 EOF가 반환된다.

√ 콘솔 대상의 fgetc, getchar 함수호출로 EOF를 반환하는 경우

- ▶ 함수호출의 실패
- ▶ Windows에서 Ctrl+Z 키, Linux에서 Ctrl+D 키가 입력이 되는 경우

```
int main(void)
{
    int ch;

    while(1)
    {
        ch=getchar();
        if(ch==EOF)
            break;
        putchar(ch);
    }
    return 0;
}
```

키보드에는 EOF가 존재하지 않는다.

따라서 EOF를 Ctrl+Z 키와 Ctrl+D 키로 약속해 놓은 것이다.

예제에서 보이듯이. 하나의 문장이 입력되어도

문장을 이루는 모든 문자들이 반복된 getchar 함수의 호출을 통해서 입력될 수 있다.

```
Hi~
Hi~
I like C lang.
I like C lang.
^Z
```

실행결과

4.4 반환형이 int이고, int형 변수에 문자를 담는 이유는?

```
int getchar(void);
int fgetc(FILE * stream);
```

√ 반환형이 char형이 아닌 int형인 이유는?

- char형은 예외적으로 signed char가 아닌 unsiged char로 표현하는 컴파일러가 존재한다.
- 파일의 끝에 도달했을 때 반환하는 EOF는 -1로 정의되어 있다.
- char를 unsigend char로 표현하는 컴파일러는 EOF에 해당하는 -1을 반환하지 못한다.
- int는 모든 컴파일러가 signed int로 처리한다. 따라서 -1의 반환에 무리가 없다.

문자열 단위 입출력 함수

4.5 문자열 출력 함수: puts, fputs

```
#include <stdio.h>
int puts(const char * s);
int fputs(const char * s, FILE * stream);
```

➡ 성공 시 0이 아닌 값을, 실패 시 EOF 반환

인자로 전달되는 문자열을 출력한다. 단 fputs 함수는 두 번째 인자를 통해서 출력의 대상을 지정할 수 있다.

```
int main(void)
{
    char * str="Simple String";

    printf("1. puts test ------ \n");
    puts(str);
    puts("So Simple String");

    printf("2. fputs test ----- \n");
    fputs(str, stdout); printf("\n");
    fputs("So Simple String", stdout); printf("\n");

    printf("3. end of main ----\n");
    return 0;
}
```

puts 함수가 호출되면 문자열 출력 후 자동으로 개행이 이뤄지지만, fputs 함수가 호출되면 문자열 출력 후 자동으로 개행이 이뤄지지 않는다는 사실에 주목!

```
1. puts test ------
Simple String
So Simple String
2. fputs test -----
Simple String
So Simple String
3. end of main ----
```

실행결과

4.6 문자열 입력 함수: gets, fgets

```
#include <stdio.h>
char * gets(char * s);
char * fgets(char * s, int n, FILE * stream);
```

➜ 파일의 끝에 도달하거나 함수호출 실패 시 NULL 포인터 반환

```
int main(void)
{
    char str[7];   // 7바이트의 메모리 공간 할당
    gets(str);   // 입력 받은 문자열을 배열 str에 저장
    . . . .
}
```

이 경우 입력되는 문자열의 길이가 배열을 넘어설 경우 할당 받지 않은 메모리를 참조하는 오류가 발생한다.

```
int main(void)
{
    char str[7];
    fgets(str, sizeof(str), stdin);
    . . . .  // stdin으로부터 문자열 입력 받아서 str에 저장
}
```

stdin으로부터 문자열을 입력 받아서
str에 저장을 하되,
널 문자를 포함하여 sizeof(str)의 크기만큼 저장을 해라.

4.7 fgets 함수의 호출의 예

```
int main(void)
{
    char str[7];
    int i;

    for(i=0; i<3; i++)
    {
        fgets(str, sizeof(str), stdin);
        printf("Read %d: %s \n", i+1, str);
    }
    return 0;
}
```

```
12345678901234567890
Read 1: 123456
Read 2: 789012
Read 3: 345678
```

실행결과1

6개의 문자씩 끊어서 읽히고 있다.

즉, 한번의 fgets 함수호출당 최대 6개의 문자만 읽혀진다.

```
We 엔터
Read 1: We

like 엔터
Read 2: like

you 엔터
Read 3: you
```

실행결과2

엔터 키의 입력도 문자열의 일부로 받아 들임을 보임

```
Y & I 엔터
Read 1: Y & I

ha ha 엔터
Read 2: ha ha

^^ -- 엔터
Read 3: ^^ --
```

실행결과3

공백을 포함하는 문자열을 읽어 들임을 보임

표준 입출력 버퍼

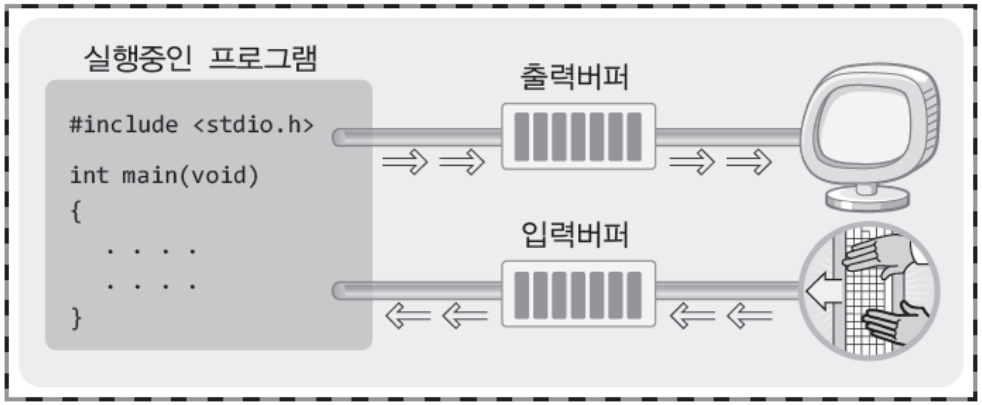
4.8 표준 입출력 기반의 버퍼와 버퍼링의 이유

√ 입출력 버퍼

- 버퍼는 특정 크기의 메모리 공간을 의미한다.
- 운영체제는 입력과 출력을 돕는 입출력 버퍼를 생성하여 제공한다.
- 표준 입출력 함수를 기반으로 데이터 입출력 시 입출력 버퍼를 거친다.

√ 입출력 버퍼에 데이터가 전송되는 시점

- 호출된 출력함수가 반환이 되는 시점이 출력버퍼로 데이터가 완전히 전송된 시점이다.
- 엔터를 입력하는 시점이 키보드로 입력된 데이터가 입력버퍼로 전달되는 시점이다.



버퍼링을 하는 이유는 데이터 이동의 효율과 관련이 있다. 데이터를 모아서 전송하면, 하나씩 전송하는 것보다 효율적이

4.9 출력버퍼를 비우는 fflush 함수

```
#include <stdio.h>
int fflush(FILE * stream);
```

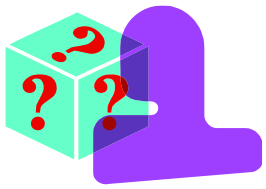
➜ 함수호출 성공 시 0, 실패 시 EOF 반환

- 인자에 해당하는 출력버퍼를 비운다.

  출력버퍼를 비운다는 것은 출력버퍼에 저장된 데이터를 지우는 것이 아니라,

  출력버퍼에 저장된 데이터를 목적지로 최종 전송함을 뜻한다.

- fflush(stdout) → 출력버퍼를 지워라!



출력버퍼의 경우와 달리 입력버퍼의 비움은 입력버퍼에 저장된 데이터의 소멸을 뜻한다.

그리고 fflush 함수는 출력버퍼를 대상으로 정의된 함수이다. 따라서 fflush(stdin) 과 같은 형태의 함수호출은 그 결과를 보장받지 못한다.

그렇다면 입력버퍼는 어떻게 비워야 할까?

4.10 입력버퍼는 어떻게 비워야 하나요?

```
int main(void)
{
    char perID[7];
    char name[10];
    fputs("주민번호 앞 6자리 입력: ", stdout);
    fgets(perID, sizeof(perID), stdin);
    fputs("이름 입력: ", stdout);
    fgets(name, sizeof(name), stdin);
    printf("주민번호: %s \n", perID);
    printf("이름: %s \n", name);
    return 0;
}
```

주민번호 앞 6자리만 입력 받기 위해서 배열의 길이가 널 문자 포함 7이다.

```
주민번호 앞 6자리 입력: 950915
이름 입력: 주민번호: 950915
이름:
```

실행결과1

엔터 키가 남아서 문제가 되는 상황

```
주민번호 앞 6자리 입력: 950709-1122345
이름 입력: 주민번호: 950709
이름: -1122345
```

실행결과2

말 안 듣는 사람들 때문에 문제되는 상황

입력버퍼를 비우는 함수

```
void ClearLineFromReadBuffer(void)
{
    while(getchar()!='\n');
}
int main(void)
{
    char perID[7];
    char name[10];
    fputs("주민번호 앞 6자리 입력: ", stdout);
    fgets(perID, sizeof(perID), stdin);
    ClearLineFromReadBuffer();   // 입력버퍼 비우기
    fputs("이름 입력: ", stdout);
    fgets(name, sizeof(name), stdin);
    printf("주민번호: %s\n", perID);
    printf("이름: %s\n", name);
    return 0;
}
```

어떠한 경우에도 주민번호 6자리만 입력 받도록 재 구현된 예제

입출력 이외의 문자열 관련 함수

4.11 문자열의 길이를 반환하는 함수 : strlen

```
#include <string.h>
size_t strlen(const char * s);
```

➜ 전달된 문자열의 길이를 반환하되, 널 문자는 길이에 포함하지 않는다.

size_t의 일반적인 선언

typedef unsigned int size_t;

typedef에 관해서는 후에 설명

```
int main(void)
{
    char str[]="1234567";
    printf("%u \n", strlen(str));
    . . . .     // 문자열의 길이 7이 출력
}
```

실행결과

```
문자열 입력: Good morning
길이: 13, 내용: Good morning

길이: 12, 내용: Good morning
```

```
void RemoveBSN(char str[])
{
    int len=strlen(str);
    str[len-1]=0;
}

int main(void)
{
    char str[100];
    printf("문자열 입력: ");
    fgets(str, sizeof(str), stdin);
    printf("길이: %d, 내용: %s \n", strlen(str), str);

    RemoveBSN(str);
    printf("길이: %d, 내용: %s \n", strlen(str), str);
    return 0;
}
```

마지막에 삽입되는 개행 문자를 없애는 예제

4.12 문자열을 복사하는 함수들: strcpy, strncpy

```
#include <string.h>
char * strcpy(char * dest, const char * src);
char * strncpy(char * dest, const char * src, size_t n);
```

➜ 복사된 문자열의 주소 값 반환

대표적인 문자열 복사 함수

```
int main(void)
{
    char str1[30]="Simple String";
    char str2[30];
    strcpy(str2, str1);
    . . . .    // str1의 문자열을 str2에 복사
}
```

str1에 저장된 문자열을 str2에 단순히 복사!

strcpy 함수를 호출하는 경우 배열의 범위를 넘어서 복사가 진행될 위험이 있다.

```
int main(void)
{
    char str1[30]="Simple String";
    char str2[30];
    strncpy(str2, str1, sizeof(str2));
    . . . .
}
```

str1에 저장된 문자열을 str2에 복사하되 최대 sizeof(str2)의 반환 값 크기만큼 복사한다.

4.13 strncpy 함수를 잘못 사용한 예

```
int main(void)
{
    char str1[20]="1234567890";
    char str2[20];
    char str3[5];

    /**** case 1 ****/
    strcpy(str2, str1);
    puts(str2);

    /**** case 2 ****/
    strncpy(str3, str1, sizeof(str3));
    puts(str3);

    /**** case 3 ****/
    strncpy(str3, str1, sizeof(str3)-1);
    str3[sizeof(str3)-1]=0;
    puts(str3);
    return 0;
}
```

배열 길이 str1에 딱 맞는 길이만큼만 복사를 하겠다는 의도의 문장

실행결과

```
1234567890
12345徼徼徼徼徼?234567890
1234
```

두 번째 strncpy 함수호출 후의 결과에 이상이 보이는 이유는 복사하는 과정에서 문자열의 끝을 의미하는 널 문자가 복사되지 않았기 때문이다. 문자열을 복사할 때에는 항상 널 문자의 복사까지 고려해야 한다.

4.14 문자열을 덧붙이는 함수들 : strcat, strncat

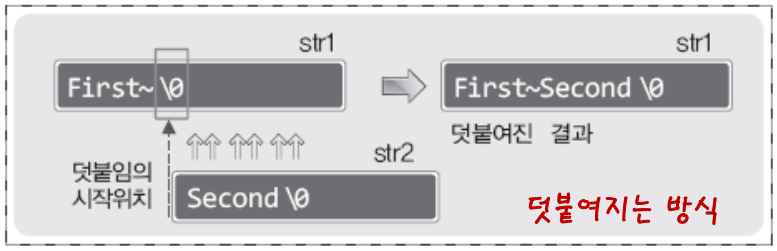
```
#include <string.h>
char * strcat(char * dest, const char * src);
char * strncat(char * dest, const char * src, size_t n);
```

➡ 덧붙여진 문자열의 주소 값 반환

strncat 함수는 덧붙일 문자열의 최대 길이를 제한한다.

최대 n개의 문자를 덧붙이되 널 문자 포함하여 n+1개의 문자를 덧붙인다.

```
int main(void)
{
    char str1[30]="First~";
    char str2[30]="Second";
    strcat(str1, str2);
    . . . .  // str1의 문자열 뒤에 str2를 복사
}
```




```
int main(void)
{
    char str1[20]="First~";
    char str2[20]="Second";
    char str3[20]="Simple num: ";
    char str4[20]="1234567890";
    /**** case 1 ****/
    strcat(str1, str2);
    puts(str1);
    /**** case 2 ****/
    strncat(str3, str4, 7);
    puts(str3);
    return 0;
}
```

실행결과

```
First~Second
Simple num: 1234567
```

4.15 문자열을 비교하는 함수들 : strcmp, strncmp

```
#include <string.h>
int strcmp(const char * s1, const char * s2);
int strncmp(const char * s1, const char * s2, size_t n);
```

➜ 두 문자열의 내용이 같으면 0, 같지 않으면 0이 아닌 값 반환

- s1이 더 크면 0보다 큰 값 반환
- s2가 더 크면 0보다 작은 값 반환
- s1과 s2의 내용이 모두 같으면 0 반환

strncmp는 최대 n개의 문자를 비교

- 크고 작음은 **아스키코드 값**을 근거로 한다.
- A보다 B가, B보다 C가 아스키 코드 값이 더 크고 A보다 a가 , B보다 b가 아스키 코드 값이 더 크니, 사전편찬순서를 기준으로 뒤에 위치할 수록 더 큰 문자열로 인식해도 된다.

```
printf("%d", strcmp("ABCD", "ABCC"));
```
0보다 큰 값이 출력

```
printf("%d", strcmp("ABCD", "ABCDE"));
```
0보다 작은 값이 출력

두 문자열이 같으면 0, 다르면 0이 아닌 값을 반환한다고 인식하고 있어도 충분하다!

4.16 문자열 비교의 예

```
int main(void)
{
    char str1[20];
    char str2[20];
    printf("문자열 입력 1: ");
    scanf("%s", str1);
    printf("문자열 입력 2: ");
    scanf("%s", str2);

    if(!strcmp(str1, str2))
    {
        puts("두 문자열은 완벽히 동일합니다.");
    }
    else
    {
        puts("두 문자열은 동일하지 않습니다.");

        if(!strncmp(str1, str2, 3))
            puts("그러나 앞 세 글자는 동일합니다.");
    }
    return 0;
}
```

실행결과

```
문자열 입력 1: Simple
문자열 입력 2: Simon
두 문자열은 동일하지 않습니다.
그러나 앞 세 글자는 동일합니다.
```

4.17 그 외의 변환함수들

`int atoi(const char * str);`	문자열의 내용을 int형으로 변환
`long atol(const char * str);`	문자열의 내용을 long형으로 변환
`double atof(const char * str);`	문자열의 내용을 double형으로 변환

헤더파일 stdlib.h에 선언

```
int main(void)
{
    char str[20];
    printf("정수 입력: ");
    scanf("%s", str);
    printf("%d \n", atoi(str));
    printf("실수 입력: ");
    scanf("%s", str);
    printf("%g \n", atof(str));
    return 0;
}
```

위의 함수들을 모른다면 문자열에 저장된 숫자 정보를 int형 또는 double형으로 변환하는 일은 번거로운 일이 될 수 있다.

실행결과

```
정수 입력: 15
15
실수 입력: 12.456
12.456
```

질문 및 정리