일일 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	20221114~20221118

세부 사항

1. 업무 내역 요약 정리

목표 내역	Done & Plan
	8 주차 I. 입출력 함수 - 스트림과 데이터 이동 (데이터 변환) - 문자 단위 입출력 함수 - 문자열 단위 입출력 함수 - 표준 입출력과 버퍼 II. 문자열 관련 함수 - 문자열 길이 계산 - 문자열 복사 - 문자열 연결 - 문자열 변환 - 문자열 검색 III. 메모리 관리 - 동적 메모리 할당/해제 - 메모리 크기 변경 - 메모리 초기화 - 메모리 영역 복사 - 메모리 영역 검색 - 메모리 영역 비교

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

I. 입출력 함수

1. 스트림과 데이터의 이동

- 데이터의 이동수단이 되는 스트림

printf 함수호출 시 어떠한 경로 및 과정을 거쳐서 문자열이 출력될까?

실행중인 프로그램과 모니터를 연결해주는 출력 스트림이라는 다리가 놓여있고, 실행중인 프로그램과 키보드를 연결해주는 입력 스트림이라는 다리가 놓여있다. printf 함수와 scanf 함수를 통해서 데이터를 입출력 할 수 있는 근본적인 이유는 바로 이 다리에 있다.

그렇다면 다리의 역할을 하는 스트림의 정체는 무엇일까?

이는 운영체제에서 제공하는 소프트웨어 적인 가상의 다리이다. 다시 말해서, 운영체제는 외부장치와 프로그램과의 데이터 송수신의 도구가 되는 스트림을 제공하고 있다.

- 스트림의 생성과 소멸

콘솔(일반적으로 키보드와 모니터를 의미함) 입출력과 파일 입출력 사이에는 차이점이 하나 있다. 그것은 파일과의 연결을 위한 스트림의 생성은 우리가 직접 요구해야 하지만, 콘솔과의 연결을 위한 스트림의 생성은 요구할 필요가 없다는 것이다. 정리하자면, 콘솔 입출력을 위한 입력 스트림과 출력 스트림은 프로그램이 실행되면 자동으로 생성되고, 프로그램이 종료되면 자동으로 소멸되는 스트림이다. 즉, 이 둘은 기본적으로 제공되는 표준 스트림이다. 그리고 표준 스트림에는 에러 스트림도 존재하며 이들 각각에는 다음과 같이 stdin, stdout, stderr라는 이름이 붙어있다.

* stdin 표준 입력 스트림 키보드 대상으로 입력
* stdout 표준 출력 스트림 모니터 대상으로 출력
* stderr 표준 에러 스트림 모니터 대상으로 출력

2. 문자 단위 입출력 함수

- 문자 출력 함수: putchar, fputc

모니터로 하나의 문자를 출력할 때 일반적으로 사용하는 두 함수는 다음과 같다.

```
#include <stdio.h>
int putchar(int c);
int fputc(int c, FILE * stream);
→ 함수호출 성공 시 쓰여진 문자정보가, 실패시 EOF 반환
```

putchar 함수는 인자로 전달된 문자정보를 stdout으로 표현되는 표준 출력 스트림으로 전송하는 함수이다. 따라서 인자로 전달된 문자를 모니터로 출력하는 함수라 할 수 있다. 그리고 문자를 전송한다는 측면에서는 fputc 함수도 putchar 함수와 동일하다. 단, fput 함수는 문자를 전송할 스트림을 지정할 수 있다. 즉 fputc 함수를 이용하면 stdout뿐만 아니라, 파일을 대상으로도 데이터를 전송할 수 있다. 그리고 fputc 함수에 대해서 부연설명을 하자면, fputc 함수의 두 번째 매개변수 stream은 문자를 출력할 스트림의 지정에 사용된다. 따라서 이 인자에 표준 출력 스트림을 의미하는 stdout을 전달하면, putchar 함수와 동일한 함수가 된다.

- 문자 입력 함수: getchar, fgetc
키보드로부터 하나의 문자를 입력 받을 때 일반적으로 사용하는 두 함수는 다음과 같다.

```
#include <stdio.h>
int getchar(void);
int fgetc(FILE * stream);
→ 파일의 끝에 도달하거나 함수호출 실패 시 EOF 반환
```

getchar 함수는 stdin으로 표현되는 표준 입력 스트림으로부터 하나의 문자를 입력 받아서 반환하는 함수이다. 따라서 키보드로부터 하나의 문자를 입력 받는 함수라 할 수 있다. 그리고 fgetc 함수도 하나의 문자를 입력 받는 함수이다. 다만 getchar 함수와 달리 문자를 입력 받을 스트림을 지정할 수 있다. 즉, 위의 두 함수의 관계는 앞서 설명한 putchar, fputc 함수의 관계와 동일하다. 그럼 지금까지 설명한 문자 입출력 함수의 동작을 확인하기 위해서 다음 예제를 제시하겠다. 이 예제에서는 앞서 설명한 4 개의 함수를 모두 호출하고 있다.

```c
#include <stdio.h>

int main(void)
{
        int ch1, ch2;

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

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

7, 8 행: 키보드로부터 각각 하나의 문자를 입력 받고 있다. 이 두 문장이 하는 일은 완전히 동일하다.
10, 11 행: 모니터로 각각 하나의 문자를 출력하고 있다. 이 두 문장이 하는 일도 완전히 동일하다.

```
a
a
```

소스코드 상에서는 분명 두 개의 문자를 입출력하고 있다. 그런데 실행결과만 놓고 보면, 하나의 문자가 입력되고 출력된 것으로 보인다. 그러나 실제로는 두 개의 문자가 입력되고 출력되었다. 다만 두 번째 문자가 엔터키이다 보니 눈에 띄질 않았을 뿐이다. 사실 엔터키도 아스키 코드 값이 10 인 ₩n으로 표현되는 문자이다. 따라서 입출력 대상이 되는 것은 당연하다.
그런데 왜 예제에서 문자를 int형 변수에 저장할까? 언뜻 생각하기에 위 예제 5 행에 선언된 두 변수 ch1 과 ch2 는 char형으로 선언도어야 할 것 같다. 하지만 int형으로 선언해야 한다. 앞서 보인 함수의 원형에서도 알 수 있듯이 getchar 함수와 fgetc 함수의 반환형이 int이기 때문이다. 이와 관련해서 EOF를 얘기한 후 자세히 얘기하겠다.

- 문자 입출력에서의 EOF
EOF는 End Of File의 약자로서, 파일의 끝을 표현하기 위해서 정의해 놓은 상수이다. 따라서 파일을 대상으로 fgetc 함수가 호출되면, 그리고 그 결과로 EOF가 반환되면, 이는 파일의 끝에 도달해서 더 이상 읽을 내용이 없다는 뜻이 된다. 그렇다면 키보드를 대상으로 하는 fgetc 함수와 getchar 함수는 언제 EOF를 반환할까? 이는 다음 두 가지 경우 중 하나가 만족되었을 때이다.
* 함수 호출의 실패
* Windows에서 ctrl+z키, linux에서 ctrl+D키가 입력되는 경우

키보드의 입력에 파일의 끝이라는 것이 존재할 수 있겠는가? 따라서 EOF의 반환시기를 ctrl+z 또는 ctrl+d키의 입력으로 별도로 약속해 놓은 것이다. 그럼 다음 예제를 통해서 키보드를 통한 EOF의 입력을 확인해보자.

```c
#include <stdio.h>

int main(void)
{
	int ch;

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

```
hi
hi
hello
hello
```

위의 실행결과는 리눅스에서의 실행결과이다. 따라서 프로그램의 종료를 위해서 ctrl+d를 입력하였다. 참고로 위의 예제에서는 getchar 함수가 호출된다고 해서 하나의 문자만 입력하려고 노력하지 않아도 된다. 문자가 아닌 공백을 포함하는 문장을 입력해도 된다. 문장이 입력되면 문장을 구성하는 문자의 수만큼 getchar 함수가 호출되면서 모든 문자를 읽어 들이니 말이다.

- 반환형이 int이고, int형 변수에 문자를 담는 이유는?

앞서 소개한 getchar 함수와 fgetc 함수를 다시 한번 관찰하자.

int getchar(void);

int fgetc(FILE * stream);

반환되는 것은 1 바이트 크기의 문자인데, 반환형이 int이다. 이유가 무엇일까?

위의 두 함수가 반환하는 값 중 하나인 eof는 -1 로 정의된 상수이다. 따라서 반환형이 char형이라면, 그리고 char를 unsigned char로 처리하는 컴파일러에 의해서 컴파일이 되었다면, eof는 반환의 과정에서 엉뚱한 양의 정수로 형 변환이 되어버리고 만다. 그래서 어떠한 상황에서도 -1 을 인식할 수 있는 int형으로 반환형을 정의해 놓은 것이다. 물론 반환되는 값을 그대로 유지하기 위해서 우리도 int형 변수에 반환 값을 저장해야 한다.

3. 문자열 단위 입출력 함수

printf 함수와 scanf 함수를 이용해도 문자열의 입출력이 가능하다. 그러나 이번에 소개하는 문자열 입출력 함수는 그 성격이 제법 다르다. scanf 함수는 공백이 포함된 형태의 문자열을 입력 받는데 제한이 있지 않았는가? 이번에 소개하는 문자열 입력 함수는 공백을 포함하는 문자열도 입력 받을 수 있다.

- 문자열 출력 함수: puts, fputs

모니터로 하나의 문자열을 출력할 때 일반적으로 사용하는 두 함수는 다음과 같다.

```
#include <stdio.h>
int puts(const char * s);
int fputs(const char * s, FILE * stream);
→ 성공 시 음수가 아닌 값을, 실패 시 eof 반환
```

puts 함수는 출력의 대상이 stdout으로 결정되어 있지만, fputs 함수는 두 번째 인자를 통해서 출력의 대상을 결정할 수 있다. 그리고 둘 다 첫 번째 인자로 전달되는 주소 값의 문자열을 출력하지만, 출력의 형태에 있어 한가지 차이점이 있다. 어떠한 차이점이 있는지 다음 예제를 통해서 확인하기로 하자.

```
#include <stdio.h>

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;
}
```

8 ,9 행: puts 함수의 호출을 보이고 있다. 문자열이 선언된 위치에는 문자열의 주소 값이 반환되므로, 두 경우 모두 문자열의 주소 값이 전달된다.

12, 13 행: fputs 함수의 호출을 보이고 있다. 두 번째 인자로 stdout이 전달되었으니, 모니터로 출력이 된다. 참고로 fputs 함수호출 후에 ₩n을 출력해서 별도의 개행 작업을 거치고 있다.

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

puts 함수가 호출되면 문자열 출력 후 자동으로 개행이 이뤄지지만, fputs 함수가 호출되면 문자열 출력 후 자동으로 개행이 이뤄지지 않는다.

- 문자열 입력 함수: gets, fgets

이번에 소개하는 두 개의 문자열 입력 함수는 다음과 같다.

```
#include <stdio.h>
char * gets(char * s);
char * fgets(char * s, int n, FILE * stream);
→ 파일의 끝에 도달하거나 함수호출 실패 시 null 포인터 변환
```

위의 gets 함수는 다음의 유형으로 호출한다.

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

위의 문장구성만으로도 키보드로부터 문자열을 입력 받게 되니, 확실히 문장구성은 간단하다. 하지만 미리 마련해 놓은 배열을 넘어서는 길이의 문자열이 입력되면, 할당 받지 않은 메모리 공간을 침범하여 실행 중 오류가 발생한다는 단점이 있다. 그래서 가급적이면 다음의 형태로 fgets 함수를 호출하는 것이 좋다.

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

위의 fgets 함수 호출은 stdin으로부터 문자열을 입력 받아서 배열 str에 저장하되, sizeof(str)의 길이만큼만 저장하라는 의미를 지닌다.

예를 들어서 위의 형태로 fgets 함수가 호출되었는데, 프로그램 사용자가 다음의 문자열을 입력하면, "123456789" sizeof(str)의 반환 값인 7 보다 하나가 작은 6 에 해당하는 길이의 문자열만 읽어서 str에 저장하게 된다. 즉, str에 저장되는 문자열은"123456"이다. 문자열이기 때문에 널 문자의 저장을 위해서 하나가 작은 길이의 문자열이 저장된다.

```c
#include <stdio.h>

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;
}
```

위 예제에서는 길이가 7 인 char 형 배열을 선언한 다음에 이를 대상으로 fgets 함수를 총 3 회 호출하고 있다. 그럼 먼저 다음의 방식으로 예제를 실행해보자

```
12345678901234567890
Read 1: 123456
Read 2: 789012
Read 3: 345678
[lms@localhost c]$
```

이는 앞서 설명한 바를 증명하는 결과이다. 입력된 문자열의 길이가 배열의 길이를 넘어서다 보니, fgets 함수는 7 보다 하나 작은 6 의 길이만큼만 문자열을 읽어 들이고 있다. 따라서 프로그램 사용자는 딱 한 번 입력하였지만. fgets 함수는 3 회 모두 호출되었다. 그러나 아직도 프로그램 사용자가 입력한 문자열을 모두 읽어 들이지 못한 상황이다. 만약에 fgets 함수를 총 4 회 호출하게 하였거나 배열의 길이를 늘린다면 입력된 문자열을 모두 읽어 들일 수 있다. 그럼 이번에는 문자열의 길이를 5 로 제한해서, 다음과 같이 키보드를 통한 입력이 총 3 회 이뤄지도록 하자.

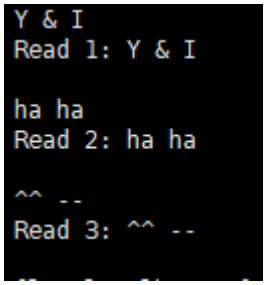
```
we
Read 1: we

like
Read 2: like

you
Read 3: you
```

위 예제 11 행의 printf 함수호출 문에는 개행을 의미하는 ₩n이 하나 삽입되어 있다. 그런데 위의 실행 결과를 보면 문장이 출력될 때마다 개행이 두 번 이뤄졌음을 알 수 있다. 이유가 무엇일까?

fgets 함수는 ₩n을 만날 때까지 문자열을 읽어 들이는데, ₩n을 제외시키거나 버리지 않고 문자열의 일부로 받아들인다. 쉽게 말해서 여러분이 입력 한 엔터 키의 정보까지도 문자열의 일부로 저장되는 것이다. 그래서 11 행의 printf 함수호출 시 문자열의 일부로 저장된 ₩n에 의해서 한번, 그리고 printf 함수호출 문에 삽입된 ₩n에 의해서 또 한 번 개행이 이뤄지는 것이다. 이제 마지막으로 공백을 포함하는 형태의 문자열을 입력하는 방식으로 예제를 실행해보자.



fgets 함수는 ₩n을 만날 때까지 문자열을 읽어 들이기 때문에, 위의 실행결과에서 보이듯이 중간에 삽입된 공백문자도 문자열의 일부로 읽어 들인다.

4. 표준 입출력과 버퍼

- 표준 입출력 기반의 버퍼

표준 입출력 함수를 통해서 데이터를 입출력 하는 경우, 해당 데이터들은 운영체제가 제공하는 메모리 버퍼를 중간에 통과하게 된다. 여기서 말하는 메모리 버퍼는 데이터를 임시로 모아두는 메모리 공간이다.

키보드를 통해 입력되는 데이터는 입력 버퍼에 저장된 다음에 (버퍼링 된 다음에) 프로그램에서 읽혀진다. 즉, fgets 함수가 읽어 들이는 문자열은 입력 버퍼에 저장된 문자열이다. 그럼 키보드로부터 입력된 데이터가 입력 스트림을 거쳐서 입력 버퍼로 들어가는 시점은 언제일까? 이는 엔터 키가 눌리는 시점이다. 그래서 키보드로 아무리 문자열을 입력해도 엔터 키가 눌리기 전에는 fgets 함수가 문자열을 읽어 들이지 못하는 것이다. 엔터 키가 눌리기 전에는 입력 버퍼가 비워져 있으니 말이다.

- 버퍼링을 하는 이유는 무엇일까?

데이터를 목적지로 바로 전송하지 않고 중간에 출력버퍼와 입력버퍼를 둬서 전송하고자 하는 데이터를 임시 저장하는 이유는 무엇일까? 이러한 데이터 버퍼링의 가장 큰 이유는 데이터 전송의 효율성과 관련이 있다. 키보드나 모니터와 같은 외부 장치와의 데이터 입출력은 생각보다 시간이 걸리는 작업이다. 따라서 버퍼링 없이 키보드가 눌릴 때마다 눌린 문자의 정보를 목적지로 바로 이동시키는 것보다 중간에 메모리 버퍼를 둬서 데이터를 한데 묶어서 이동시키는 것이 보다 효율적이고 빠르다.

창고에 물건을 나르는 경우 손으로 하나씩 나르는 것보다 손수레에 가득 채워서 나르는 것이 보다 빠르고 효율적이라는 것을 알고 있을 것이다. 입출력의 과정에서 버퍼링을 했을 때 보다 빠르고 효율적인 이유도 이와 같은 이치이다.

- 출력버퍼를 비우는 fflush 함수

출력버퍼가 비워진다는 것은 출력버퍼에 저장된 데이터가 버퍼를 떠나서 목적지로 이동됨을 뜻한다. 그런데 출력버퍼가 비워지는 시점은 시스템에 따라 그리고 버퍼의 성격에 따라 달라진다. 예를 들어서 버퍼가 꽉 찼을 때 비워지는 버퍼도 있고, 하나의 문장이 완전히 입력되었을 때마다 비워지는 버퍼도 있다. 이렇듯 버퍼가 비워지는 시점을 동일하지 않기 때문에 다음 함수를 알아 둘 필요가 있다.

```
#include <stdio.h>
int fflush(FILE * stream);
→ 함수호출 성공 시 0, 실패 시 EOF 반환
```

위 함수는 인자로 전달된 스트림의 버퍼를 비우는 기능을 제공한다. 따라서 다음과 같이 함수를 호출하면,

fflush(stdout); // 표준 출력버퍼를 비워라

어떠한 시스템의 어떠한 표준 출력버퍼라 할지라도 버퍼에 저장된 내용이 비워지면서 데이터가 목적지로 이동한다. 참고로 위의 함수는 파일을 대상으로도 호출이 가능하다. 인자로 파일의 스트림정보가 전달되면, 해당 버퍼에 저장되어 있던 데이터들이 버퍼를 떠나서 파일에 기록이 된다. 그런데 여러분이 콘솔 입출력을 하는 상황이라면, 그리고 windows나 linux와 같은 범용 os를 사용하고 있다면 stdout을 대상으로 위의 함수를 호출할 일은 사실상 많지 않다.

- 입력버퍼는 어떻게 비워야 하나?

입력버퍼의 비워짐은 출력버퍼의 비워짐과 개념적으로 차이가 있다. 출력버퍼의 비워짐이 저장된 데이터가 목적지로 전송됨을 의미한다면, 입력버퍼의 비워짐은 데이터의 소멸을 의미하기 때문이다. 그런데 잠시 후에 예제를 통해서 보게 되겠지만, 가끔은 입력버퍼에 남아있는 불필요한 데이터의 소멸을 위해서 입력버퍼를 비워야 하는 경우가 종종 있다. 그렇다면 입력버퍼는 어떻게 비워야 할까?

fflush(stdin);

위에서 언급했듯이 fflush 함수는 출력버퍼를 대상으로 호출하는 함수이다. 조금 더 정확히 설명하면, c언어의 표준에서는 위의결과에 대해 정의하고 있지 않다. 따라서 위의 함수호출 결과는 예측이 불가능하다. 물론 일부 컴파일러는 위의 형태로 함수가 호출되었을 때 입력버퍼를 비워주기도 한다. 대표적으로 windows 계열의 컴파일러가 그러하다. 하지만 그 이외의 컴파일러들은 전혀 다른 결과를 보인다. 그럼 입력버퍼에 저장된 불필요한 데이터는 어떻게 소멸해야 할까?

```c
#include <stdio.h>

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;
}
~
```

5 행에 선언된 배열의 길이가 7 임에 주목하자. 이는 주민번호 앞 6 자리만 저장해야 하는 상황을 고려하여 널 문자까지 저장할 수 있도록 선언된 배열이다.

```
주민번호 앞 6자리 입력 : 030211
이름 입력 : 주민번호 : 030211
이름 :
```

분명히 6 자리만 입력을 했는데도 문제가 생겼다. 이름을 입력할 기회를 얻지 못한 것이다. 이러한 문제가 발생한 이유를 알겠는가? 위의 실행결과에서 입력한 데이터는 030211 ↵ 이다. 이렇듯 엔터 키를 포함하여 총 7 문자가 입력되었다. 그런데 9 행의 fgets 함수의 인자로 7 이 전달되었으니, 널 문자를 제외하고 최대 6 문자를 읽어 들인다. 따라서 ₩n을 제외한 나머지 여섯 문자만 읽혀지고 ₩n은 입력버퍼에 남아있게 된다. 그리고 이어서 12 행의 fgets 함수가 호출된다. 그런데 fgets 함수는 ₩n을 만날 때까지 읽어 들이는 함수이니, 버퍼에 남아있는 ₩n만 읽어버리고 만다. 때문에 위와 같은 실행결과를 보이는 것이다. 이러한 문제의 상황을 해결하기 위해서는 예제 실행 중간에, 입력버퍼 남아있는 ₩n 문자 하나만 지워버리면 된다. 하지만 다음의 실행결과까지 고려하면 상황은 달라진다.

```
주민번호 앞 6자리 입력 : 030211-3170612
이름 입력 : 주민번호 : 030211
이름 : -3170612
```

분명히 주민번호 앞 6 자리만 입력하라고 했는데, -를 포함하여 총 14 자리의 주민번호를 전부 입력하였다. 그래서 9 행의 fgets 함수호출을 통해서 여섯 개의 문자가 읽혀지고, 12 행의 fgets 함수호출을 통해서 나머지 문자들이 읽혀진 것이다. 이렇듯 명시한대로 행동하지 않는 프로그램 사용자를 고려한다면, 주민번호 앞 6 자리를 제외한 나머지 문자들을 입력버퍼에서 지워줘야한다.

이제 필요한 것이 무엇인지 알았을 것이다. 위의 예제가 정상적으로 동작하는데 필요한 다음 함수를 정의하고자 한다.

```
void ClearLineFromReadBuffer(void)
{
        while(getchar()!='₩n');
}
```

입력버퍼에 저장된 문자들은 읽어 들이면 지워진다. 그래서 ₩n을 만날 때까지 문자를 읽어 들이는 함수를 정의하였다. 물론 읽어 들인 문자를 저장하거나 하지는 않는다. 버리는 것이 목적이니 말이다. 그럼 이 함수를 추가하여 예제를 다시 작성하겠다.

```c
#include <stdio.h>

void clear(void)
{
        while(getchar()!='\n');
}

int main(void)
{
        char perID[7];
        char name[10];

        fputs("주민번호 앞 6자리 입력 : ", stdout);
        fgets(perID, sizeof(perID), stdin);
        clear(); // 입력버퍼 비우기

        fputs("이름 입력 : ", stdout);
        fgets(name, sizeof(name), stdin);

        printf("주민번호 : %s \n", perID);
        printf("이름 : %s \n", name);
        return 0;
}
```

위 예제에서 정의한 clear 함수는 입력버퍼를 통째로 비우는 함수가 아니라, ₩n이 읽혀질 때까지 입력버퍼에 저장된 문자들을 지우는 함수이다.

```
주민번호 앞 6자리 입력 : 030211
이름 입력 : 이민성
주민번호 : 030211
이름 : 이민성
```

입력버퍼에 남아있는 ₩n을 지워버리기 때문에 위와 같이 정상적으로 입출력이 이뤄진다. 이어서 다음 실행결과를 보자.

```
주민번호 앞 6자리 입력 : 030211-1111234
이름 입력 : dlalstjd
주민번호 : 030211
이름 : dlalstjd
```

프로그램 사용자가 잘못 입력을 해도, 필요한 만큼만 읽어 들이고 나머지는 지워버리기 때문에 위에서 보이는 바와 같이 정상적으로 동작한다.

II. 문자열 관련 함수

여기서는 헤더파일 string.h에 선언된 문자열 관련 함수들 중 사용 빈도수가 높은 몇몇 함수를 소개하고자 한다.

- 문자열의 길이를 반환하는 함수: strlen

다음 함수는 인자로 전달된 문자열의 길이를 반환하는 함수로서 문자열과 관련해서 많이 사용되는 대표적인 함수이다.

```
#include <string.h>
size_t strlen(const shar * s);
→ 전달된 문자열의 길이를 반환하되, 널 문자의 길이에 포함하지 않는다.
```

위 함수의 반환형 size_t는 일반적으로 다음과 같이 선언되어 있다.

typedef unsigned int size_t

typedef 선언으로 인해서 size_t가 unsigned int를 대신할 수 있게 되었다. 즉, 위의 typedef 선언으로 인해서 size_t가 unsigned int를 대신할 수 있게 된 것이다. 따라서 다음 두 선언은 완전히 동일하다.

size_t len;

unsigned int len;

그럼 이어서 strlen 함수의 호출방법을 보이겠다.

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

참고로 strlen 함수의 반환형은 size_t이니, 이 함수의 반환 값을 unsigned int형 변수에 저장하고 서식문자 %u로 출력하는 것이 정확하다. 그러나 문자열이 아무리 길어도 문자열의 길이정보는 int형 변수에 저장이 가능하기 때문에, strlen 함수의 반환 값을 int형 변수에 저장하고 서식문자 %d로 출력하는 것도 가능할 뿐만 아니라 이것이 더 흔한 일이다. 그래서 이후부터는 strlen 함수의 반환 값을 int형 변수에 저장하고 %d로 출력하겠다.

그럼 strlen 함수와 관련해서 다음 예제를 살펴보자. 이 예제에서는 다음 요구사항에 대한 해결책을 제시한다.
"fgets 함수호출을 통해서 문자열을 입력 받고 싶은데, 같이 딸려서 들어오는 ₩n 문자는 문자열에서 제외시키고 싶다."

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

void remove(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);

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

6, 7 행: 문자열의 길이를 계산해서 ₩n이 저장된 위치에 널 문자의 아스키 코드 값 0 을 저장하고 있다. 이로써 ₩n은 문자열에서 사라진 셈이다.
14 행: fgets 함수호출을 통해서 문자열을 입력 받고 있다. 따라서 ₩n 문자가 문자열의 일부로 포함된다.

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

길이 : 12, 내용 : good morning
```

15 행을 통한 출력에서는 개행이 두 번 이뤄졌다. 그런데 17 행의 remove 함수호출 이후에 18 행의 출력에서는 개행이 한 번 이뤄졌다. 이는 remove 함수호출을 통해서 ₩n문자가 소멸되었기 때문이다.

- 문자열을 복사하는 함수들: strcpy, strncpy

이번에는 문자열의 복사에 사용되는 함수 둘을 소개하겠다.

```
#include <string.h>
char * strcpy(char * dest, const char * src);
char * strncpy(char * dest, const char * src, size_t n);
→ 복사된 문자열의 주소 값 변환
```

위의 strcpy 함수를 호출하는 형태는 다음과 같다.

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

위의 코드가 실행되면, str2 에는 str1 이 저장하고 있는 문자열이 복사된다. 물론 문자열이 복사될 배열의 길이가

문자열의 길이보다 작지 않도록 주의해야 한다.

그럼 이어서 strncpy 함수의 호출형태를 보이겠다.

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

위 코드의 strncpy 함수 호출문은 "str1 에 저장된 문자열을 str2 에 복사하되, str1 의 길이가 매우 길다면, sizeof(str2)가 반환한 값에 해당하는 문자의 수만큼만 복사를 진행해라"라는 의미를 지닌다.

이렇듯 strncpy 함수는 복사될 배열의 길이를 넘어서지 않는 범위 내에서 복사를 진행하고자 하는 경우에 유용하다. 하지만 이 역시 주의해야 할 사실이 하나 있는데, 이는 다음 예제를 통해서 설명하겠다.

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

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;
}
```

```
1234567890
12345
1234
```

위의 예제 15 행에는 다음 문장이 삽입되어 있다.

strncpy(str3, str1, sizeof(str3));

그리고 이 문장을 보면서 "복사하는 최대 문자의 수로 sizeof(str3)의 반환 값이 전달되었으니, 할당된 배열을 넘어서서 복사가 이뤄지지는 않겠구나"라고 생각할 수 있다.

실제로 배열을 넘어서서 복사가 이뤄지지는 않으니 이것이 잘못된 판단은 아니다. 즉, 배열 str3 의 길이가 5 이니 총 5 개의 문자가 복사된다. 단, 이 5 개의 문자 안에 널 문자가 포함되지 않는다는 문제가 있다. strncpy 함수는 문자열을 단순하게 복사한다. 5 개의 문자를 복사하라고 하면 앞에서부터 딱 5 개의 문자만 복사한다. 마지막 문자가 널 문자인지 아닌지는 상관하지 않는다. 따라서 위의 문장 실행 후 str3 에 저장되는 다섯 개의 문자는 다음과 같다.

1, 2, 3, 4, 5

그래서 16 행의 출력결과가 이상한 것이다. 널 문자가 존재해야 널 문자 이전까지 출력을 할 텐데, 널 문자가 존재하지 않으니 엉뚱한 영역까지 출력을 하는 것이다. 그럼 strncpy 함수는 어떻게 호출해야 할까? 위 예제 19, 20 행에서 보이듯이 다음과 같이 호출해야 한다.

```
strncpy(str3, str1, sizeof(str3)-1);
        str3[sizeof(str3)-1]=0;
```

strncpy 함수의 세 번째 인자로 배열의 실제길이보다 하나 작은 값을 전달해서 널 문자가 삽입될 공간을 남겨두고 복사를 진행해야 한다. 그리고 이어서 배열의 끝에 널 문자를 삽입해야 한다.

- 문자열을 덧붙이는 함수들: strcat, strncat

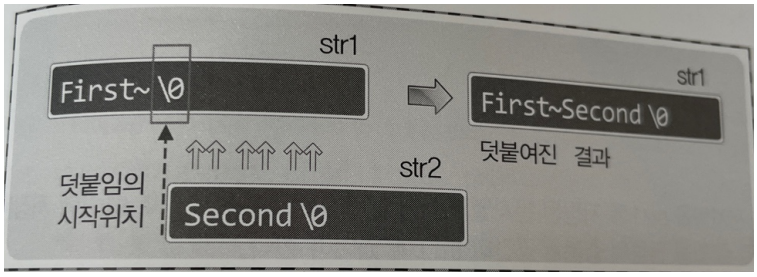
두 함수는 문자열의 뒤에 다른 문자열을 복사하는 기능을 제공한다. 간단히 str1 에 저장된 문자열의 뒤에 str2 에 저장된 문자열을 좀 복사해라를 만족시키는 함수로 이해하면 된다.

```
#include <string.h>
char * stract(char * dest, const char * src);
char* strncat(char * dest, const char * src, size_t n);
→ 덧붙여진 문자열의 주소 값 변환
```

stract 함수를 호출하는 형태는 다음과 같다.

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

위의 형태로 stract 함수가 호출되면 str2 의 문자열이 str1 의 문자열 뒤에 덧붙여지는데, 덧붙여지는 형태는 다음과 같다(널 문자가 어떻게 처리되는지 주의 깊게 보자)



위 그림에서 보이듯이 덧붙임이 시작되는 위치는 널 문자 다음이 아닌, 널 문자가 저장된 위치에서부터 이다. 이렇듯 널 문자가 저장된 위치에서부터 복사가 진행되어야 덧붙임 이후에도 문자열의 끝에 하나의 널문자만 존재하는 정상적인 문자열이 되지 않겠는가. 이어서 다음 문장을 보자. 이는 strncat 함수의 호출문이다.

```
strncat(str1, str2, 8);
```

이 문장이 의미하는 바는 다음과 같다.

"str2 의 문자열 중 최대 8 개를 str1 의 뒤에 덧붙여라!"

즉 str2 의 길이가 8 을 넘어선다면 8 개의 문자까지만 str1 에 덧붙이라는 의미인데, 이 8 개의 문자에는 널 문자가 포함되지 않는다는 사실에 주목하자. 따라서 널 문자를 포함하여 실제로는 총 9 개의 문자가 str1 에 덧붙여진다. 이렇듯 strncpy 함수와 달리 strncat 함수는 문자열의 끝에 널 문자를 자동으로 삽입해준다. 그럼 지금까지 공부한 내용의 확인을 위한 예제를 제시하겠다.

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

int main(void)
{
        char str1[20]="First~";
        char str2[20]="Second";

        char str3[20]="Simple num: ";
        char str4[20]="1234567890";

        /**** case1 ****/
        strcat(str1, str2);
        puts(str1);

        /**** case2 ****/
        strncat(str3, str4, 7);
        puts(str3);
        return 0;
}
```

13 행: str2 에 저장된 문자열을 str1 에 저장된 문자열의 뒤에 덧붙이고 있다. 이 때 str2 에 저장된 문자열을 덧붙일 수 있을 만큼을 충분한 공간이 str1 에 있어야 한다.

17 행: str4 에 저장된 문자열을 str3 에 저장된 문자열의 뒤에 덧붙이되 7 개의 문자만 덧붙이고 있다. 따라서 널 문자를 포함해서 8 개의 문자가 덧붙여진다.

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

- 문자열을 비교하는 함수들: strcmp, strncmp

```
int main(void)
{
        char str1[]="My String";
        char str2[]="My String";
        if(str1==str2)
                puts("equal");
        else
                puts("not equal");
        return 0;
}
```

위의 코드에서는 str1 과 str2 를 비교하고 있다. 그런데 이는 문자열의 내용을 비교하는 것이 아니라 배열 str1 과 배열 str2 의 주소 값을 비교하는 것이다. 배열의 이름은 배열의 주소 값을 의미하므로 이는 배열의 주소 값 비교로 이어지고, 때문에 위 예제의 실행결과로는 결코 문자열 equal 가 출력되지 않는다. 따라서 문자열의 내용을 비교하고자 한다면, 다음의 함수를 별도로 호출해야 한다.

```
#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 0 이 아닌 값 반환
```

위의 두 함수 모두 인자로 전달된 두 문자열의 내용을 비교하여 다음의 결과를 반환한다. 단, strncmp 함수는 세 번째 인자로 전달된 수의 크기만큼만 문자를 비교한다. 즉 strncmp 함수를 호출하면 앞에서부터 시작에서 중간부분까지 부분적으로만 문자열을 비교할 수 있다.

* s1 이 더 크면 0 보다 큰 값 반환
* s2 가 더 크면 0 보다 작은 값 반환
* s1 과 s2 의 내용이 모두 같으면 0 반환

여기서 말하는 문자열의 크고 작음은 아스키 코드 값을 기준으로 결정된다. 예를 들어서 다음 두 문자열을 비교한다고 가정해보자.

"ABCD"

"ABCC"

첫 번째 문자부터 비교가 시작된다. 그런데 세 번째 문자까지 동일하다. 따라서 네 번째 문자를 비교하게 된다. 그런데 D의 아스키 코드 값이 C의 아스키 코드 값보다 크다. 따라서 "ABCD"가 더 큰 문자열이 되어 다음의 실행결과로 양수가 출력된다.

printf("%d", strcmp("ABCD", "ABCC"));

참고로 양수가 반환되어야 한다는 사실만 표준으로 정의되어 있을 뿐, 그 값이 구체적으로 얼마가 되어야 한다는 사실까지 정의되어 있지는 않다. 따라서 여러분도 양수가 반환된다는 사실만을 근거로 코드를 작성해야 한다. 컴파일러에 따라서 반환되는 양의 정수 값은 달라질 수 있으니 말이다. 그럼 이번에는 다음 두 문자열을 비교해보자.

"ABCD"

"ABCDE"

첫 번째 문자부터 비교가 시작되어 네 번째 문자까지 비교가 진행된다. 그런데 네 번째 문자도 동일하다. 그래서 마지막으로 다섯 번째 문자를 비교한다. "ABCD"의 다섯 번째 문자는 널 문자이고, "ABCDE"의 다섯 번째 문자는 E이다. 그런데 E의 아스키 코드 값은 널의 아스키 코드 값인 0 보다 크므로 다음의 실행결과로는 음수가 반환된다.

printf("%d", strcmp("ABCD", "ABCDE"));

일반적으로 strcmp 함수를 호출할 때에는 다음 두 가지 사실에만 근거하여 코드를 작성한다.

"0 이 반환되면 동일한 문자열, 0 이 아닌 값이 반환되면 동일하지 않은 문자열"

때문에 언제 음수가 반환되고, 언제 양수가 반환되는지는 그리 중요하게 인식되지 않는다.

그럼 다음 예제를 통해서 strcmp 함수와 strncmp 함수의 호출 예를 모두 보이겠다.

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

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;
}
```

13 행: str1 과 str2 가 동일하면 거짓을 의미하는 0 이 반환된다. 그런데 이 반환 값을 대상으로 ! 연산을 하였으니 거짓은 참으로 바뀐다. 즉 이 if문은 str1 과 str2 의 문자열이 완벽히 동일할 때 참이 된다.
21 행: 이 문장은 두 문자열이 일치하지 않는 경우에 한해서 실행된다. 그리고 strncmp 함수의 세 번째 인자로 3 이 전달되었으니, 앞의 세 문자가 동일한 경우에 한해서 if문이 참이되어 22 행을 실행하게 된다.

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

- 문자열 변환

다음은 헤더파일 <stdlib.h>에 선언된 함수들이다.

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

문자열로 표현된 정수나 실수의 값을 해당 정수나 실수의 데이터로 변환해야 하는 경우가 간혹 있다. 예를 들어서 문자열 "123"을 정수 123 으로 변환하거나 문자열 "7.15"를 실수 7.15 로 변환해야 하는 경우가 종종 있다는 뜻이다. 이러한 경우에는 위 함수의 존재를 알면 쉽게 해결이 가능하다. 그럼 다음 예제를 통해서 위 함수들의 사용방법을 보이겠다.

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

int main(void)
{
        char str[20];
        printf("정 수  입 력 : ");
        scanf("%s", str);
        printf("%d \n", atoi(str));

        printf("실 수  입 력 : ");
        scanf("%s", str);
        printf("%g \n", atof(str));
        return 0;
}
```

8, 9 행: 프로그램 사용자로부터 문자열을 입력 받고 있다. 그리고 그 문자열에 담긴 내용을 정수로 변환해서 서식문자 %d로 출력하고 있다.

12, 13 행: 프로그램 사용자로부터 입력 받은 문자열을 실수로 변환해서 서식문자 %g로 출력하고 있다.

```
정 수  입 력 : 4
4
실 수  입 력 : 3.14
3.14
```

- 문자열 검색: strstr()함수

strstr 함수를 사용하기 위해서는 해당 함수가 포함되어있는 <string.h> 헤더를 포함해야 한다.

strstr 함수는 찾고자하는 문자열이 있다면 해당 문자열로 시작하는 문자열 포인터를 반환하고 찾고자 하는 문자열이 없다면 NULL값을 반환한다.

```c
#include <stdio.h>
#include <string.h> // strstr 함 수 가  선 언 된  헤 더  파

int main()
{
        char str[] = "Hello World Welcome to the World"
        char* ptr = strstr(str, "World"); // world로  시 작 하 는  문 자 열  검 색
        printf("주 소 값 : %p\n", ptr);
        printf("문 자 열 : %s\n", ptr);

        return 0;
}
```

```
주 소 값 : 0x7ff7b4e8fa26
문 자 열 : World Welcome to the World
```

strstr 함수를 사용하면 대상 문자열에서 찾고자 하는 문자열을 발견하면 해당 위치의 주소 값을 반환하고

종료된다. 문자열에 찾고 자하는 문자열이 여러 개가 있어도 하나밖에 찾지 못한다. 이것을 방지하기 위해서 먼저 찾은 포인터에 +1 을 더하여 계속해서 검색해주어야 한다.

```
#include <stdio.h>
#include <string.h> //strstr 함수가 선언된 헤더 파일

int main()
{

    char str[] = "Hello World Wecome to the World My World";
    char* ptr = strstr(str, "World"); // World으로 시작하는 문자열 검색
    int count = 0;

    while (ptr != NULL) // 검색된 문자열이 없을 때까지 반복
    {
        printf("%s\n", ptr); // 검색된 문자열 출력
        ptr = strstr(ptr + 1, "World"); // 리턴된 포인터 +1 계속 검색
        count++;
    }

    printf("찾은 문자열 %d개 ", count);

    return 0;
}
```

```
World Wecome to the World My World
World My World
World
찾은 문자열 3개
```

strstr함수는 대상 문자열에 찾을 문자열이 없다면 NULL값을 반환한다. NULL값이 나올때까지 리턴된 포인터에 1 을 더하여 계속해서 검색하면 찾고자 하는 문자열이 여러개가 있어도 전부 찾을 수 있다.

III. 메모리 관리

- 동적 메모리 할당/해제

동적 할당: 프로그램 작성 단계가 아닌 프로그램 실행 단계에서 결정되는 크기에 따라 메모리를 할당해주는 방식으로 상황에 따라 필요한 만큼의 메모리를 할당 받을 수 있어 유연한 프로그램을 작성하는 것이 가능하다.

동적 메모리 할당: 프로그램 작성할 때 동적 메모리가 몇 개가 필요할지 알 수 없다. 그래서 포인터 변수를 이용하여 간접적으로 참조하게 된다. malloc 함수가 할당된 메모리의 시작 주소를 반환하므로 이를 포인터 변수에 대입하고, 이후에는 이 포인터 변수를 이용하여 동적으로 할당 받은 메모리에 접근한다. 여기서 malloc 함수에 할당된 메모리를 어떤 자료형으로 사용할지 모르기 때문에 사용하고자 하는 자료형의 포인터로 형 변환을 해주어야 한다. 아래는 동적 할당 받은 메모리를 포인터 변수에 연결하는 기본적인 예시이다.

```
int *p = NULL;                                  // 포인터 변수 선언 및 NULL로 초기화
p = (int*) malloc( 4*sizeof(int) );     // 앞의 괄호는 사용할 자료형, 뒤에 괄호는 메모리 크기
```

이렇게 포인터 변수와 메모리를 연결 후엔 포인터 또는 배열 형태로 사용하면 된다.

동적 메모리 헤제: 정적으로 할당된 변수는 메모리가 자동으로 해제되지만, 이와 반대로 동적 메모리는 직접 해제를 해줘야 한다. 동적 메모리를 해제할 땐 free( ) 함수를 이용한다. 동적 메모리를 해제할 땐 free( ) 함수를 이용한다.

함수 원형	void free(void *ptr);
함수 인자	헤제할 메모리의 시작 주소

메모리 해제를 제대로 하지 않으면 메모리 누수가 발생한다. 메모리 누수란 사용하지 않는 메모리가 계속 점유하는 현상을 말한다. 이러한 상황은 간단한 프로그램에서는 큰 문제가 되진 않지만 서버 프로그램과 같이 오래 실행되는 프로그램에서는 메모리 부족이 되어 발생할 수 있다. 그러므로 불필요한 메모리는 꼭 해제하도록 해야 한다.

동적 메모리 사용 중 주의 사항:

1. 동적 메모리를 할당받은 포인터 변수를 할당받기 전과 해제한 후에 NULL을 대입하여 메모리 접근 오류를 방지한다.

```
int *p = NULL;          // 포인터 변수 선언시 NULL 로 초기화
p = (int *) malloc( 10*sizeof(int) );

...

free(p);
p = NULL;               // 메모리 해제 후 NULL 대입 (대글링 포인터 방지)
```

2. malloc ( ) 함수를 호출한 후에는 그 반환 값을 검사하여 메모리 할당의 성공 여부를 확인한다.

```
int *p = NULL;
p =(int*) malloc( 10*sizeof(int) );

if( p == NULL ){                          // 메모리 할당 실패
        printf("Not enough memory!");      // 오류 상황 알림
    return -1;                            // 함수 종료
}
```

3. 메모리를 해제하기 전에, 해제하려는 메모리 주소가 NULL 인지 점검한다.

```
int *p = NULL;
p = (int*) malloc( 10*sizeof(int) );

if(p! = NULL )
        free(p);      //p가 NULL이 아닌 경우만 free( ) 함수 호출
```

- 메모리 크기 변경

헤더	Stdlib.h
형태	Void *realloc(void *ptr, size_t size)
인수	Void *ptr 메모리의 크기를 변경할 포인터 Size_t size 새로 지정할 메모리의 크기
반환	Void * 메모리의 크기가 변경된 메모리의 주소

주의: 새로 변경할 메모리의 크기가 변경하기 어려운 경우에는 아래와 같이 새로 메모리가 할당되므로, 이전 포인터 값을 계속 사용하는 것은 위험하다.
새로 지정한 메모리 양으로 메모리를 할당 받고,
이전 메모리의 내용을 복사한 후,
이전에 할당 받은 메모리를 반환한다.
그리고 새로 할당받은 메모리를 반환한다.

```
char *ptr;
char *trmp;

tmp = malloc( 10);
ptr = malloc( 10);

printf( "realloc() 호출 전 ptr의 값: %x\n", ptr);
ptr = realloc( ptr, 50);
printf( "realloc() 호출 후 ptr의 값: %x\n", ptr);
```

tmp가 할당받은 메모리가 걸리기 때문에 바로 크기 변경을 할 수 없어 메모리를 새로 할당받게 된다. realloc() 함수 호출 전과 호출 후의 주소 값을 확인해 보면 바뀐 것을 볼 수 있다.

예제:

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

int main( void)
{
   char   *ptr;
   char   *tmp;

   tmp = malloc( 10);
   ptr = malloc( 10);

   strcpy( ptr, "badayak");

   printf( "realloc() 호출 전 ptr의 값: %p\n", ptr);
   ptr   = realloc( ptr, 50);
   printf( "realloc() 호출 후 ptr의 값: %p\n", ptr);

   strcpy( ptr+strlen( ptr), ".com");  // 이전 문자열 뒤에 .com 추가
   printf( "%s\n", ptr);

   free( ptr);

   return 0;
}
```

```
]$ ./a.out
realloc() 호출 전 ptr의 값: 0x1993030
realloc() 호출 후 ptr의 값: 0x1993460
badayak.com
]$
```

- 메모리 초기화

memset 함수는 메모리의 내용을 원하는 크기만큼 특정 값으로 세팅할 수 있는 함수이다.

함수 원형: void* memset(void* ptr, int value, size_t num);

첫번째 인자 void* ptr은 세팅하고자 하는 메모리의 시작 주소로 즉, 그 주소를 가리키고 있는 포인터가 위치하는 자리 입니다.

두번째 인자 value는 메모리에 세팅하고자 하는 값을 집어 넣으면 된다. int 타입으로 받지만 내부에서는 unsigned char 로 변환되어서 저장된다. 즉 'a' 이런것을 넣어도 무방하다는 뜻이다.

세번째 인자 size_t num은 길이를 뜻한다. 이 길이는 바이트 단위로써 메모리의 크기 한 조각 단위의 길이를 말한다. 이는 보통 "길이 * sizeof(데이터타입)" 의 형태로 작성하면 된다.

반환값은 성공하면 첫번째 인자로 들어간 ptr을 반환하고, 실패한다면 NULL을 반환한다.

헤더파일은 memory.h 혹은 string.h 둘중 하나를 사용해도 무방하다.

char arr[] = "blockdmask"; // 목적지의 첫번째 주소(포인터)
memset(arr, 'c', 5 * sizeof(char)); // 변경하고자 하는 값 c, 변경하고자 하는 길이 * char 타입의 바이트 크기
print(arr);

예제 1: 문자 배열 변경

```c
#include<string.h> // #include<memory.h> 도 괜찮습니다.
#include<stdio.h>

int main(void)
{
    char arr1[] = "blockdmask blog";
    memset(arr1, 'c', 5 * sizeof(char));
    printf(arr1);

    return 0;
}
```

```
ccccdmask blog
```

주의사항: 결과에서 보셨듯이 0 이 아닌 int 타입의 숫자를 넣게되면 예상할 수 없는 값으로 세팅 된다. 그 이유는 memset 함수는 1 바이트 단위로 값을 초기화 하기 때문이다. 1 바이트 단위로 1 를 만들었기 때문에 4 바이트로 표현된 int 숫자 1 은 제대로된 숫자로 표현될 수 없다.

0 은 4 바이트든 1 바이트든 0 이기 때문에 동일한 결과값을 기대할 수 있던 것 이다.

결론적으로
0 이아닌 그리고 char 타입이 아닌 값을 넣어서 세팅 하려고 할때는 우리가 원하는 값으로 세팅할수 없다.
즉, 0 이랑 char 타입만 사용해야한다.
그렇기 때문에 memset 함수는 보통 문자열(char 배열)에서 값을 변경하거나, 숫자배열을 0 으로 세팅할때 사용하곤 한다.

- 메모리 영역 복사

헤더	string.h
형태	Void *memmove(void *s1, const void *s2, size_t n)
인수	Void *s1 복사될 메모리의 포인터 Void *s2 복사할 메모리의 포인터 Size_t size 복사할 바이트 갯 수
반환	Void * s1 포인터를 반환하며 실패하면 NULL을 반환함

하나의 포인터에 대해서 동일한 영역 내에서 복사가 가능하다.

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

int main( void)
{
   char  ptr[] = "badayak.com";
   char  tmp;
   int   ndx;

   for ( ndx = 0; ndx <= strlen(ptr); ndx++)
   {
      printf( "%s\n", ptr);                 // 현재의 ptr 출력

      tmp    = ptr[0];                      // ptr의 첫번째 문자
      memmove( ptr, ptr+1, strlen(ptr)); // 2번째문자부터 끝문자까지 복사
      ptr[strlen(ptr)] =  tmp;              // ptr 끝에 보관해둔 첫번째 문자
   }
   return 0;
}
```

```
]$ ./a.out
badayak.com
adayak.comb
dayak.comba
ayak.combad
yak.combada
ak.combaday
k.combadaya
.combadayak
combadayak.
ombadayak.c
mbadayak.co
badayak.com
]$
```

- 메모리 영역 검색

헤더	string.h
형태	Void *memchr(const void *s, int c, size_t n)
인수	Void *s 검사할 메모리의 포인터 Int c 검색 문자 Size_t n 검사할 영역의 크기
반환	Void * 처음 발견된 위치의 포인터. 발견하지 못하면 null

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

int main( void)
{
   char  *ptr = "badayak.com";

   printf( "found=%s\n", ( char *)memchr( ptr, 'c',  7)); // 7자 안에는 'c'가 없음
   printf( "found=%s\n", ( char *)memchr( ptr, 'c', strlen( ptr))); // 'c' 위치의 포인터로 문자열 출력
   return 0;
}
```

```
]$ ./a.out
found=(null)
found=com
]$
```

- 메모리 영역 비교

```
#include <string.h>


int memcmp(const void *s1, const void *s2, size_t n);
```

memcmp(3)은 s1 과 s2 의 메모리 영역을 첫번째 바이트부터 n바이트 만큼 비교하여 최초로 다른 값을 만났을 때에 크고 작음을 return한다. n바이트를 다 비교했는데, 모든 데이터가 같으면 0 을 return 한다.
strcmp(3)함수와 달리 0x00(null ternminate값)을 만나도 계속 비교를 한다.

파라미터:

```
s1

    – 비교할 메모리 영역 1

s2

    – 비교할 메모리 영역 2

n

    – 비교할 데이터 바이트 수
```

return:

```
0 보다 작음

    – 다른 데이터를 처음 만났을 때(idx 번째)에 s1[idx]이 s2[idx]보다 작은 값입니다.

0

    – n 바이트 모두 같은 값입니다.

0 보다 큼

    – 다른 데이터를 처음 만났을 때(idx 번째)에 s1[idx]이 s2[idx]보다 큰 값입니다.
```