C PORT FOLIO 2020

20190715
신현정

컴퓨터정보공학
E반

들어가며

본 포트폴리오는 필자의 C언어에 대한 이해도를 높이는 것이 주 목적입니다. 해당 목적을 달성하기 위해 필자의 눈높이에 맞추어 재해석하였습니다. 그 과정에서 그림과 예제를 필자의 방식대로 더 쉽게 제작하였고 본인만 보는 것이 아닌 다른 이들에게 가르쳐 준다는 생각으로 설명에 임하였습니다.

내용은 도서 'Perfect C'의 11장부터 15장과 해당 학기의 강의 내용을 기반으로 두고 있으며, 개발환경으로는 'Visual Studio 2019'를 사용했습니다. 자세한 강의 일정에 따른 강의 내용은 뒷 페이지를 참고해주세요.

필자는 지난 학기의 C언어 강의에서 Perfect C 1장부터 10장의 내용을 공부하였습니다. 공부 과정에서 앞서 말한 개발 환경에 대한 전반적인 사용법과 자료형, 배열의 기초내용, 포인터에 대한 기초 내용을 충분히 습득하였고, 그렇기 때문에 해당 습득 내용이 존재한다는 전제 하에 내용이 기술되었습니다.

해당 내용을 참고하시며 읽어주시길 바라겠습니다.

동양미래대학 컴퓨터정보공학과
신현정

강의 일정

주차	내용
1	강의 OT 11-1장 ~ 11-2장
2	11-2 ~ 12-2장 Visual Studio의 설치 및 사용법
3	12-3장 Visual Studio 실습 방법
4	13-1장 ~ 13-2장
5	13-3장 ~ 14-2장
6	14-2장 ~ 15-1장
7	15-1장 ~ 15-3장

Perfect C를 기반으로 작성된 강의 일정입니다.

CONTENTS

11 문자와 문자열

1. 문자와 문자열
2. 문자열 관련 함수
3. 여러 문자열 처리

12 변수 유효범위

1. 전역변수와 지역변수
2. 정적변수와 레지스터 변수
3. 메모리 영역과 변수 이용

13 구조체와 공용체

1. 구조체와 공용체
2. 자료형 재정의
3. 구조체와 공용체의 포인터 배열

14 함수와 포인터 활용

1. 함수의 인자 전달 방식
2. 포인터 전달과 반환
3. 힘수 포인터와 void포인터

15 파일 처리

1. 파일 기초
2. 텍스트 파일 입출력
3. 이진 파일 입출력

CHAPTER

11

문자와 문자열

11.1 문자와 문자열
11.2 문자열 관련 함수
11.3 여러 문자열 처리

1 CHAPTER 11 문자와 문자열

01 문자와 문자열의 개념

02 문자와 문자열의 선언

03 문자와 문자열의 입력

04 문자와 문자열의 출력

01 문자와 문자열의 개념

문자란 간단히 말하자면, 작은 따옴표에 의해 기재된 문자 상수를 말한다.
즉, 영어의 알파벳이나 한글의 한 글자를 작은 따옴표(' ')로 둘러싸 ' A ' 와 같이 표기되는 것을 문자라 일컫는다. C에서는 기본적으로 1바이트의 크기를 차지하는 char 자료형을 사용한다.

문자열이란, 앞서 설명한 문자의 모임인 일련의 문자를 일컫는다. 문자와 다르게 작은 따옴표가 아닌 큰따옴표(" ")로 표기한다. 만일 작은 따옴표로 둘러쌀 경우 1byte의 문자가 아니므로 오류가 발생한다.

02 문자와 문자열의 선언

문자는 앞서 언급했던 것 처럼 char 자료형을 사용해 변수를 선언한다.

```
char 변수명 = '값';

char ch = 'A'; //작은 따옴표로 감쌈
```

문자열은 문자와 다르게 따로 자료형이 존재하지 않는다.
하지만 문자열은 문자들의 모임의 개념이므로 char형(문자) 배열로 변수를 선언해 각각의 원소에 문자를 저장한다.

다음과 같은 방법이 가장 기본적인 방법이다.

```
char 변수명[] = "값";

char ch[] = "DearD"; //큰 따옴표로 감쌈
```

아래와 같은 방법도 사용이 가능하다.

```
char 변수명[] = {'넣을문자열의 n번째 문자 값',...,'\0'};

char ch[] = {'D','e','a','r','D'};
```

```
char 변수명[넣을문자수+1];
ch[n] = '넣을문자열의 n번째 문자 값';
ch[마지막원소위치]='\0';

char ch[6];
ch[0] = 'D'; ch[1] = 'e'; ch[2] = 'a'; ch[3] = 'r'; ch[4] = 'D';
ch[5]='\0';
```

코드를 살펴보면 맨 처음 방식보다 복잡해지고, 가장 마지막 원소로 '₩0' 을 넣는 것을 알 수 있다. '₩0' 을 넣는 것은 문자열의 마지막을 알려주기 위해 삽입한다.

첫번째 방식처럼 큰 따옴표로 문자열을 선언하면 자동으로 뒤에 '₩0'이 삽입되기 때문에 명시적으로 정의할 필요가 없었지만 두번째, 세번째 방식처럼 문자열을 선언할 경우엔 자동으로 삽입이 되지 않기 때문에 명시적으로 정의해 주어야 한다.

또한, 첫번째 방식을 사용하더라도 배열의 크기를 지정하는 경우가 있는데 이때 배열의 크기는 넣을 문자열의 문자수+1 로 지정해 주어야한다. '₩0'이 삽입될 공간이 존재해야하기 때문이다.

TIP

C는 아스키 코드를 기반이기 때문에 한글과 영어의 byte 차이가 난다.
한 글자당 영어는 1byte, 한글은 2byte
그러므로 한글을 저장하는 배열의 크기를 지정할 땐
문자 수의 2배의 크기를 지정해 주도록 하자!

03 문자와 문자열의 입력

getchar()

getchar()는 라인 버퍼링(line buffering)방식을 사용하는 입력 함수이다.
문자를 입력하면 임시저장소인 버퍼에 저장되었다가 enter를 누르면 버퍼에서 문자를 읽어와 저장된다. 즉, enter가 눌려져야 문자가 저장되므로 즉각적인 입력을 요하는 작업에서 사용하면 안된다.

사용 방법은 다음과 같다.

```
char 변수명='\0'; //문자를 null 값으로 초기화 함
변수명 = getchar(변수명); //enter를 누르면 입력이 처리 됨
```

_getche()

_getche()는 버퍼를 사용하지 않고 문자를 입력하는 함수이다.
getchar()와 달리 버퍼를 사용하지 않기 때문에 임시 저장소의 개념이 없어 즉각적인 입력이 가능하다. 또한, 한 글자가 입력될 때마다 바로 입력처리가 되므로
한 번 문자를 입력하게 되면 입력 창에서 수정될 수 없다

사용 방법은 다음과 같다.

```
char 변수명='\0'; //문자를 null 값으로 초기화 함
변수명 = _getche(변수명); //문자 입력 시, 처리 됨
```

단 , 사용 시 헤더파일 conio.h 을 정의해주어야 한다.

_getch()

_getch()는 버퍼를 사용하지 않는 입력을 위한 함수이다.
_getche()와 비슷하지만, 특이한 점은 다른 입력 함수와 달리 입력 창에서 사용자가 문자를 입력한 내용이 보이지 않는다는 것이다. 그 이유는 echo기능에 있다.
다른 함수들은 echo기능을 자동으로 해주기 때문에 입력한 내용이 가시적으로 보이지만, _getch()는 자동으로 이루어지지않기때문에 보이지 않는 것이다.

사용 방법은 다음과 같다.

```
char 변수명='\0'; //문자를 null 값으로 초기화 함
변수명 = _getch(변수명); //문자 입력 시 처리, 입력 내용이 보이지 않음
```

단 , 사용 시 헤더파일 conio.h 을 정의해주어야 한다.

TIP

헤더파일 conio.h를 이용하는 입력 함수의 경우
버퍼를 사용하지 않고 입력한 문자의 수정이 불가하다!

scanf() / scanf_s()

scanf()는 공백으로 구분되는 문자열을 입력받기 위해 사용하는 표준입력함수이다.
첫번째인자로는 값을 형식제어문자 %s를 이용해 값을 얼마난 입력받을 것인지를 전달하고, 두번째 인자로는 입력한 값을 넣을 변수의 주소를 전달한다.

사용 방법은 다음과 같다.

```
char 변수명[101]; //문자 배열의 공간이 충분이 주어져야 함
//변수명 자체가 포인터의 개념이기 때문에 %변수명의 경우 오류
scanf("%s",변수명);
```

단, scanf()를 사용할 경우 문자열이나 파일에 관련된 버퍼나 스택등 메모리에 문제가 생길 수도 있기 때문에

코드의 맨 위에 다음과 같이 전처리기를 정의해주어야한다.

```
#define _CRT_SECURE_NO_WARNINGS
```

이것을 사용하는 것을 원치 않을 경우, scanf_s()를 사용하면 된다.
scanf_s()는 scanf()와 사용 방법은 같으나 세번째 인자로 메모리의 크기(버퍼 크기)를 정의해 사용해야 한다.

gets() / get_s()

gets()는 개행으로 구분되는 문자열을 입력받기 위해 사용하는 함수이다.
사용에 있어서 scanf()와의 차이점은 처리 속도가 빠르다는 것에 있으며, 인자로 버퍼의 주소 값을 전달한다.

이 함수를 이용해 문자열을 입력할 때 수행 과정은 다음과 같다.
enter를 누르게 되면 해당 문자열을 버퍼에 저장하게 된다.
여기서 enter를 입력한 자리엔 개행을 뜻하는 ₩n이 삽입 되는데
이것이 ₩0 으로 바뀌어 배열에 저장하는 작업이 수행된다.

사용 방법은 다음과 같다.

```
char 변수명[101]; //문자 배열의 공간이 충분이 주어져야 함
//변수명 자체가 포인터의 개념이기 때문에 %변수명의 경우 오류
gets(변수명);
```

gets()는 앞서 설명했던 scanf()와 동일한 문제가 발생하므로 위와 동일한 전처리기를 정의해 주어야한다. 마찬가지로 전처리기를 생략하고 싶다면 gets_s()를 사용하며, 두번째 인자로 메모리의 크기(버퍼 크기)를 전달한다.

04 문자와 문자열의 출력

putchar()

putchar()는 함께 문자를 출력할 때 주로 사용되는 함수이다.

사용 방법은 다음과 같다.

```
putchar(변수명);
```

_putch()

putch()는 _getch()와 함께 문자를 출력할 때 주로 사용되는 함수이다.

사용 방법은 다음과 같다.

```
_putch(변수명);
```

단 , 사용 시 헤더파일 conio.h 을 정의해주어야 한다.

printf()

printf()는 다양한 출력을 가능하게 하는 함수이다.
형식 제어 문자를 이용해 출력을 하며, 형식제어문자 %s는 null문자까지 하나의 문자열로 인식한다. 문자열이 아니더라도 다양한 타입의 출력이 가능하다.

문자 출력 시, 사용 방법은 다음과 같다.

```
printf("%c",변수명);
```

문자열 출력 시, 사용 방법은 다음과 같다.

```
printf("%s",변수명);
```

다음은 문자 포인터를 이용해 변수를 선언하고, ₩0을 이용해 문자를 분리해 printf()함수로 출력하는 예제이다.

```c
#include <stdio.h>

int main(void)
{
    char* ch = "DearD\0Blog"; //\0으로 분리
    int i=0;

    // 해당 원소 값이 null이 아닐 경우, 해당 원소 출력
    while (ch[i]) printf("%c", ch[i++]); //DearD 출력
    printf("\n");

    i++;
    while (ch[i]) printf("%c", ch[i++]); //Blog 출력
    printf("\n");

    return 0;
}
```

```
DearD
Blog
```

printf()함수는 출력 방법에 있어 가장 기본적이고 가장 많이 쓰이는 함수이다.
위 결과 값을 확인하고 printf()의 개념에 대해 확립해보자.

puts()

puts()은 한 행의 문자열을 출력하는데 사용되는 함수이다.
인자로 출력할 문자열을 전달한다.

인자로 주어진 문자열의 NULL이 있는 곳까지 출력이 이루어지는데, gets()와는 반대로 ₩0을 ₩n으로 교체하여 버퍼에 전송한 뒤, 한 행을 출력한다.

사용은 다음과 같이 한다.

```
puts(변수명);
```

오류가 발생하면 EOF(End Of File)을 반환하며, 그렇지 않을 경우 0을 반환한다.

2 CHAPTER 11 문자열 관련 함수

01 길이 관련 함수
02 문자 찾기 관련 함수
03 비교 관련 함수
04 복사 관련 함수
05 연결 관련 함수
06 추출 관련 함수
07 변환 관련 함수

STOP

문자열 관련 함수는
헤더파일 string.h 를 정의해야 사용 가능!

01 길이 관련 함수

strlen()

strlen()은 인자로 주어진 문자 포인터의 위치에서부터 null까지 null문자는 제외한 문자열의 길이를 반환하는 함수이다.

사용 방법은 다음과 같다.

```
strlen(문자열);
//문자열에 DearD가 저장돼 있다면, 5가 반환될 것임
```

02 문자 찾기 관련 함수

memchr()

memchr() 은 첫번째 인자로 주어진 문자포인터에서 두번째 인자의 문자를 찾아 두번째 인자에 해당하는 문자부터 세번째 인자로 주어진 바이트 크기에 해당하는 문자 위치까지의 문자열(두번째 인자 포함)을 반환하는 함수이다.

사용 방법은 다음과 같다.

```
memchr(검사할문자열, 찾을문자, 바이트크기);
```

```
char* ch="DearD is a blogger";
char* mem = memchr(ch, 'b',strlen(ch)); //반환 값을 저장
//mem을 출력한다면 blogger라는 값이 출력될 것임
```

03 비교 관련 함수

memcmp()

memcmp()는 문자열의 처음부터 세번째 인자의 바이트 크기의 위치까지 첫번째 인자와 두번째 인자의 문자들을 사전식으로 비교해 같으면 0, 두번째 인자가 크면 양수, 첫번째 인자가 크면 음수를 반환하는 함수이다.

사용 방법은 다음과 같다.

```
memcmp(문자열1, 문자열2, 바이트크기);
```

strcmp()

strcmp()는 memcmp()와의 동작 방식은 비슷하지만, 어디까지 비교할 것인지를 정하지 못하고 둘 중 더 큰 메모리를 차지하는 문자열의 바이트크기의 위치까지 비교한다.

사용 방법은 다음과 같다.

```
strcmp(문자열1, 문자열2);
```

strncmp()

strncmp()와 memcmp()는 같은 기능을 수행한다.

사용 방법은 다음과 같다.

```
strncmp(문자열1, 문자열2, 바이트크기);
```

04 복사 관련 함수

memcpy()

memcpy()함수는 두번째 인자 문자 포인터의 위치에서부터 세번째인자의 바이트 크기만큼 복사한 후 첫번째 인자 문자 포인터의 위치에서부터 복사 내용을 삽입하고, 저장한 문자열을 반환한다.

사용 방법은 다음과 같다.

```
memcpy(저장할문자열, 추출할문자열, 바이트크기);
```

```
char ch1[]="abcd";
char* ch2 = "eeee";
memcpy(ch1, ch2, 3);
//ch2의 3바이트까지(abc) 복사해 ch1의 3바이트까지 붙여넣기가 수행됨
//c1 출력 시, eeed가 출력될 것
```

memmove()

memcpy()와 같은 기능을 수행한다.

사용 방법은 다음과 같다.

```
memmove(저장할문자열, 추출할문자열, 바이트크기);
```

strcpy() / strcpy_s()

strcpy()는 memcpy()와 비슷하지만 두번째 인자에서 얼마나 추출할지를 정할 수 없는 함수이다. 자동으로 두번째 인자의 null까지 추출되어 복사된다.

사용 방법은 다음과 같다.

```
strcpy(저장할문자열, 추출할문자열);
```

단, strcpy() 사용 시 코드의 맨 위에 다음과 같은 전처리기를 정의해주어야한다.

```
#define _CRT_SECURE_NO_WARNINGS
```

그렇지 않으면, strcpy_s()의 두번째 인자로 첫번째 인자의 바이트 크기를 전달해 사용한다.

strcpy()와의 차이점은 수정 되고 반환되는 것이 수정된 첫번째 인자가 아니라 숫자(성공 시 0)이다.

strncpy() / strncpy_s()

memcpy()와 같은 기능을 수행한다.

사용 방법은 다음과 같다.

```
strncpy(저장할문자열, 추출할문자열, 바이트크기);
```

단, strncpy() 사용 시 코드의 맨 위에 다음과 같은 전처리기를 정의해주어야한다.

```
#define _CRT_SECURE_NO_WARNINGS
```

그렇지 않으면, strncpy_s()의 두번째 인자로 첫번째 인자의 바이트 크기를,
네번째 인자로 두번째 인자의 바이트 크기를 전달해 사용한다.
strncpy()와의 차이점은 수정 되고 반환되는 것이 수정된 첫번째 인자가 아니라
숫자(성공 시 0)이다.

05 연결 관련 함수

strcat() / strcat_s()

strcat()은 두번째 인자 문자 포인터의 위치부터 null문자까지를 첫번째 인자의
null문자부터 연결하는 함수이다.

사용 방법은 다음과 같다.

```
strcat(문자열1, 문자열2);
//문자열1에 문자열2를 연결
```

마찬가지로 전처리기를 사용해야하며 그렇지 않으면, strcat_s()의 두번째 인자로 첫번째 인자의 바이트 크기를 전달해 사용한다.

strcat()와의 차이점은 연결되고 반환되는 것이 수정된 첫번째 인자가 아니라 숫자(성공 시 0)이다.

strcat() / strcat_s()

strncat()은 두번째 인자 문자 포인터의 위치부터 세번째 인자의 바이트 크기만큼을 첫번째 인자의 null문자부터 연결하는 함수이다.

사용 방법은 다음과 같다

```
strcat(문자열1, 문자열2, 바이트크기);
```

마찬가지로 전처리기를 사용해야하며 그렇지 않으면, strncat_s()의 두번째 인자로 첫번째 인자의 바이트 크기를, 네번째 인자로 두번째 인자의 바이트 크기를 전달해 사용한다.

strncat()와의 차이점은 수정 되고 반환되는 것이 수정된 첫번째 인자가 아니라 숫자(성공 시 0)이다.

06 추출 관련 함수

strtok() / strtok_s()

strtok()는 두번째 인자 구분자를 기준으로 첫번째 인자로 지정된 문자열을 분리해 문자 포인터로 반환하는 함수이다.

첫번째 인자로는 문자열 상수(문자 포인터)의 사용 불가하다.
두번째 인자는 문자열, 상수 모두 가능하며, 구분자를 여러개 지정할 수도 있다.

사용 방법은 다음과 같다.

```
char* 토큰을저장할변수명 = strtok(추출대상문자열, 구분자);
```

마찬가지로 전처리기를 사용해야하며 그렇지 않으면, 다음 토큰의 위치값의 저장을 목적으로 strtok_s()의 세번째 인자로 문자포인터를 전달해 사용한다.

07 변환 관련 함수

memset()

memset() 은 첫번째 인자로 주어진 문자포인터의 위치에서 세번째 인자로 주어진 바이트 크기에 해당하는 문자 위치까지 두번째 인자의 문자로 설정한뒤 설정된 문자열을 반환하는 함수이다. 원래 문자가 수정되므로 따로 반환 값을 저장해 주지 않아도 된다.

사용 방법은 다음과 같다.

```
memset(문자포인터, 문자, 수정할문자열의마지막위치);
```

```
char ch[]="DearD is a blogger"; //수정을 위해 배열로 선언함
memset(ch, 'D',strlen(ch));
//ch를 출력한다면 모두 D로 바뀌어 있을 것임
```

_strlwr() / _strlwr_s()

_strlwr()은 인자로 전달된 문자 포인터의 내용을 모두 소문자로 수정하고 수정된 문자열을 반환하는 함수이다.

인자로 문자열상수(문자 포인터)의 사용이 불가하다.

사용 방법은 다음과 같다.

```
_strlwr(변환할문자);
```

마찬가지로 전처리기를 사용해야하며 그렇지 않으면, _strlwr_s()의 두번째인자로 첫번째인자의 바이트크기를 전달해 사용한다.

_strlwr()과의 차이점은 수정된 문자열을 반환하지 않고 숫자(성공 시 0)이다.

_strupr() / _strupr_s()

_strupr()은 인자로 전달된 문자 포인터의 내용을 모두 대문자로 수정하고 수정된 내용을 반환하는 함수이다.

인자로 문자열상수(문자 포인터)의 사용이 불가하다.

사용 방법은 다음과 같다.

```
_strupr(변환할문자);
```

마찬가지로 전처리기를 사용해야하며 그렇지 않으면, _strupr_s()의 두번째인자로 첫번째인자의 바이트크기를 전달해 사용한다.

_strupr()과의 차이점은 수정된 문자열을 반환하지 않고 숫자(성공 시 0)이다.

3 CHAPTER 11 여러 문자열 처리

01 문자 포인터 배열

02 이차원 문자 배열

03 명령행 인자

지난 11-1장과 11-2장에서는 한 개의 문자열만을 가지고 다루어 왔다.
그렇다면 여러 개의 문자를 한꺼번에 저장하고 다루는 방법은 없을까?
이번 11-3장에서는 그에 대한 내용을 다루어 본다.

01 문자 포인터 배열

문자열 각각에 대한 주소 값을 문자포인터에 저장하게 되므로 여러개의 문자열 저장을 위한 최적의 공간을 사용하게 되며, 여러 개의 문자열 처리가 가능해 진다.

사용 방법은 다음과 같다.

```
char* 변수명[] = {"문자열1", "문자열2", "문자열3"...};
```

하지만 이러한 방식은 문자열 상수(문자 포인터)로 저장되므로 각각의 원소 즉, 각각의 문자열에 대해 수정이 불가능하다. 수정이 가능하게 하려면 어떻게 해야할지 계속 진행해보자.

02 이차원 문자 배열

수정이 가능하게 하려면 여러개의 문자열의 모임에서 각각의 문자열들로 나누고, 나눈 문자열들을 각각의 문자로 나누는 과정을 거쳐 문자열의 각 문자에 메모리를 할당해 주어야한다.

위 과정을 가능하게 하는 것이 이차원 배열이다.
이차원배열은 열의 크기를 꼭 지정해주어야 하므로 여러 문자열들 중 가장 긴 문자열을 기준으로 1이 큰 크기를 열의 크기로 지정해 이차원 배열을 선언해 사용한다.

사용 방법은 다음과 같다.

```
char 변수명[][가장 긴 문자열의 크기+1] = {"문자열1", "문자열2"...};
```

이러한 방식의 단점은 가장 긴 문자열을 기준으로 열의 크기를 지정하므로 문자열이 상대적으로 짧은 문자열은 남는 열의 자리에 전부 NULL차지 된다. 이러한 경우, 메모리 공간을 낭비하게 된다.

하지만 요즘은 메모리 문제가 크지 않으므로 그다지 문제가 되진 않는다.

03 명령행 인자

명령행 인자를 이용해 여러개의 문자열을 프로그램으로 전달해 처리하는 방식이다.
여기서 명령한 인자란, 프로그램에서 main()함수의 인자로 기술되는 것들인데
이 명령행인자를 이용해 처리를 하려면 main함수의 인자로 다음처럼 작성해야한다.

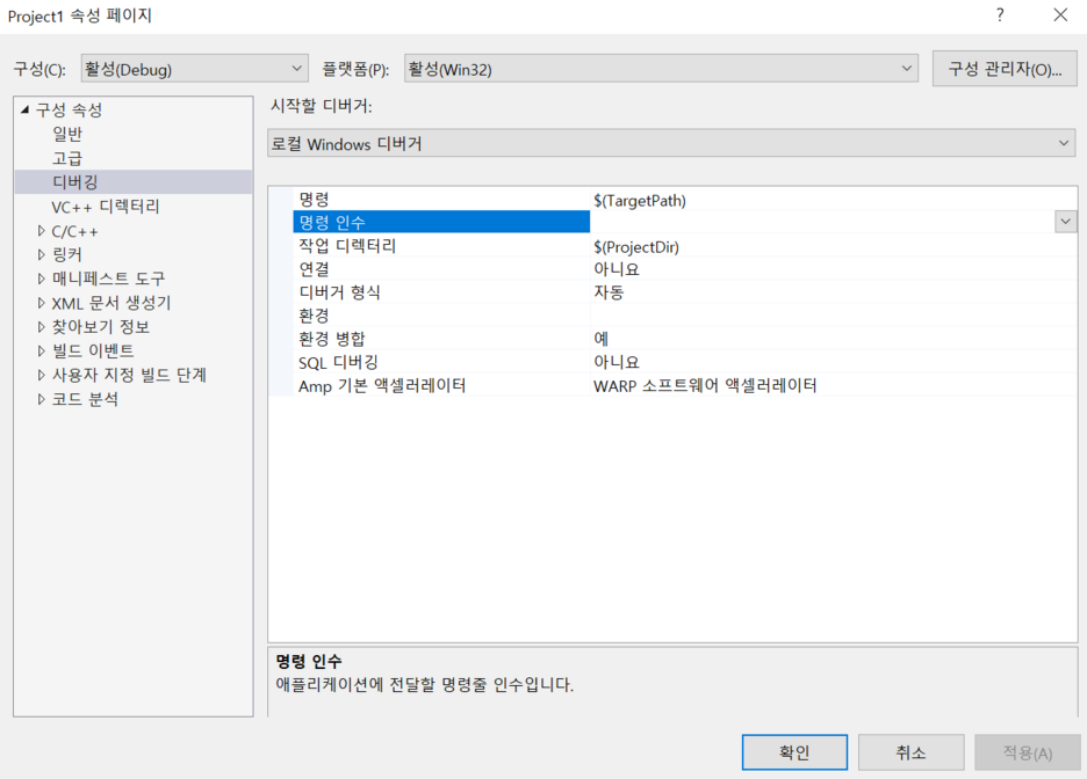
```
 int main(int argc, char* argv[]){
//Code
return 0;
}
```

첫번째 인자인 argc는 명령행에서 입력한 문자열의 수이며, 두번째 인자인 argv는 명령행에서 입력한 문자열의 주소값과, 실행파일명, 옵션들이 저장된 문자열 상수이다. argv의 0번째 원소로 실행파일명이 저장되고, 그 이후부터 입력한 문자열이 차례대로 저장된다.

명령행 인자를 설정하는 방식은 2가지 방식이 존재하며, 다음과 같다

프로그램 속성 창에서 설정하기

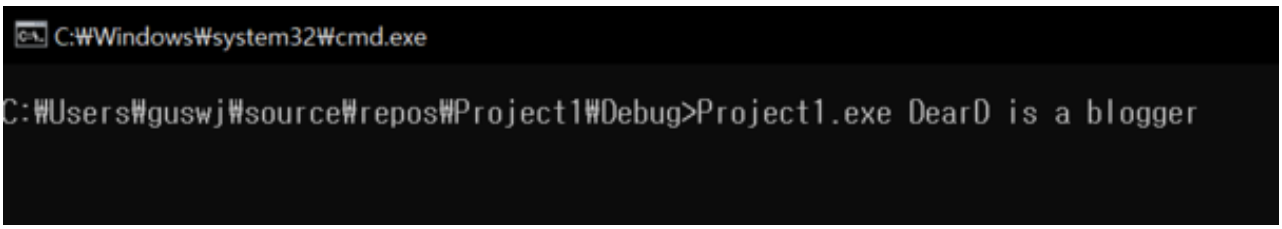
실행 프로젝트 우클릭 > 속성 > 디버깅으로 이동해 명령인수의 입력상자에 argv에 넣을 문자열을 입력한다.



▲ 속성창

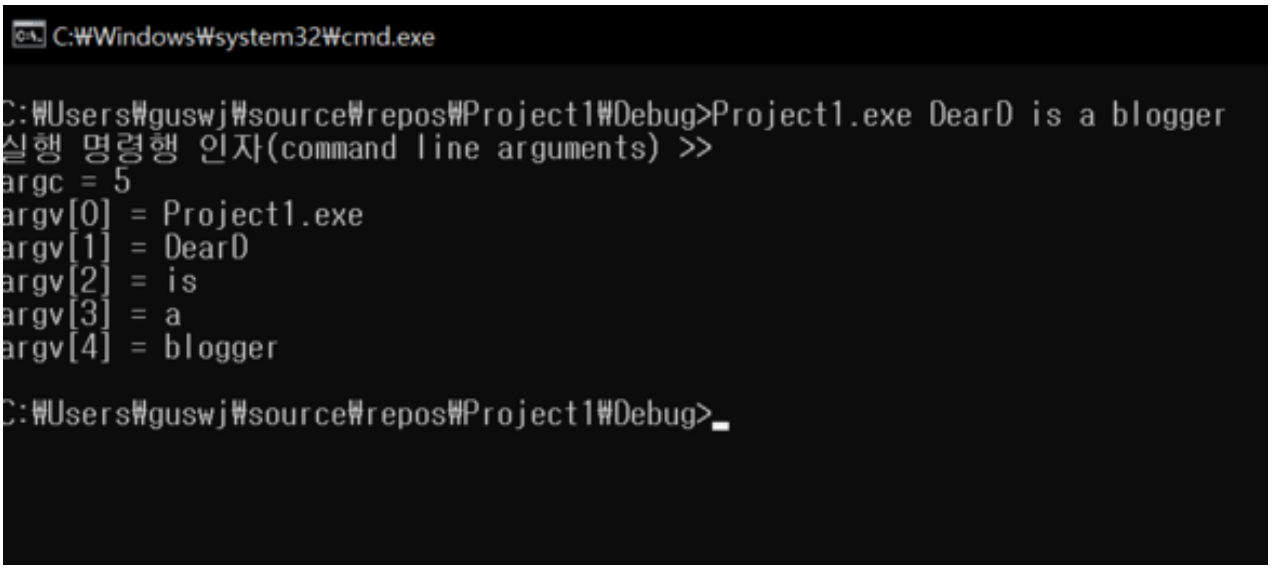
명령 프롬프트에서 설정하기

앞서 언급했던 것과 같이 main의 인자를 설정한 뒤 도스 창으로 해당 프로젝트의 실행파일(exe)을 전달할 문자열과 함께 다음과 같이 실행한다.



▲ 실행 (DearD is blogger 입력)

위와 같이 실행파일명과 넣을 문자열을 입력한 뒤 enter 키를 누르면 해당 문자열이 들어 가도록 실행파일이 실행될 것이다.



▲ 결과

결과를 보면 위와 같이 배열에 성공적으로 저장이 된 것을 확인할 수 있다.

사용한 코드는 다음과 같다.

```
#include <stdio.h>

int main(int argc, char* argv[]) {

    int i = 0;

    printf("실행 명령행 인자(command line arguments) >> \n");
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);

    return 0;
}
```

CHAPTER

12

변수 유효범위

12.1 전역변수와 지역변수

12.2 정적 변수와 레지스터 변수

12.3 메모리 영역과 변수 이용

1 CHAPTER 12 전역변수와 지역변수

01 변수의 유효 범위

02 전역 유효 범위와 전역 변수

03 지역 유효 범위와 지역 변수

01 변수의 유효 범위 (Scope)

변수의 유효 범위(scope)란 변수의 참조가 가능한 범위를 말한다.

유효 범위는 전역 유효 범위(global scope)와 지역 유효 범위(local scope)로 나누어지며, 이 유효 범위에 따라 변수를 크게 2가지로 구분 할 수 있다.

바로 전역 변수(global variable)와 지역 변수(local variable)이다.

이번 12-1장을 학습하며, 해당 개념에 대한 내용들을 함께 알아보도록 하자.

02 전역 유효 범위와 전역 변수

전역 유효 범위

전역 유효 범위는 2가지로 나누어 설명할 수 있다.
하나의 파일 내부에서 참조가 가능한 범위와 프로젝트를 구성하는 모든 범위에서 참조가 가능한 범위 모두 전역 유효 범위라 일컫는다.

전역 변수의 개념

전역 변수는 위 전역 유효 범위에서 선언된 변수를 말한다.

전역 유효 범위 내에서 활동하는 변수이기 때문에 일반적으로 프로젝트 내 모든 함수에서 참조가 가능하다.

함수 외부에서 선언되는 변수이기 때문에 외부 변수라고도 말한다.

전역 변수의 선언

선언 시, 초기화를 하지 않아도 다음과 같은 기본 값으로 초기화 된다.

정수	문자	실수	포인터
0	₩0	0.0	NULL

함수 블록에서 전역 변수와 같은 이름으로 지역변수의 선언이 가능하지만 함수 블록에서 참조 시, 지역변수를 가리킨다고 인식하므로 사실상 전역 변수의 사용이 불가하게 된다. 또한, 프로그램은 순차적으로 코드 인식을 하기 때문에 선언 시, 전역변수를 쓰려고하는 함수 뒤에 선언이 된다면 인식을 하지 못해 오류가 발생한다. 이러한 점을 주의하도록 하자.

다른 파일에서의 전역 변수 참조

다른 파일에서 선언된 전역변수를 참조하려면 참조할 전역변수의 이름과 동일한 전역 변수를 새로 생성하는데 이때 키워드 extern를 붙여준다.
키워드 extern을 붙여 줌으로 이미 다른 파일에서 선언된 전역 변수이고 이건 새로 생성하는 변수가 아니라 다른 파일에 선언된 전역 변수를 가져다 쓰겠다는 것을 명시해 주는 작업을 거친다. 이미 존재하는 전역변수의 범위를 확장하는 개념이므로 데이터 타입의 생략이 가능하다.
이 extern 키워드에 대한 내용은 12-2장에서 더 자세히 다루도록 한다.

전역 변수의 부작용

마지막으로, 전역 변수는 참조 범위가 넓기 때문에 어디서든 수정할 수 있다는 장점이 있지만, 남발을 하게 되면 어디서 수정이 됐는지 알 수 없게 되므로 적당히 사용하는 것이 올바르다.

03 지역 유효 범위와 지역 변수

지역 유효 범위

지역 유효 범위란 함수 또는 블록 내부에 선언되어 해당 블록 영역에서 참조가 가능한 범위를 일컫는다.

지역 변수의 개념

지역 변수는 위 지역 유효 범위에서 선언된 변수를 말한다.

지역 유효 범위에서만 참조가 가능하기 때문에 선언된 함수나 블록 내부에서만 참조가 가능하며, 그 외에서는 사용될 수 없다.

지역 변수는 전역 변수와는 반대로 보통 함수 내부에서 선언되는 변수이기 때문에
내부 변수라고도 말한다.
또한, 선언과 동시에 특정 메모리영역으로부터 메모리가 할당되고,
함수나 블록이 종료된 순간 할당된 메모리가 자동으로 제거되기 때문에
자동 변수라고도 말한다. 이는 auto라는 키워드 와 관련이 있는데, 생략이 가능하여 안붙이는 것이 일반적이다. auto키워드에 대한 내용은 12-2장에서 다루도록 한다.

지역 변수의 선언

선언 시, 초기화를 하지 않으면 알 수 없는 쓰레기 값이 저장돼 참조하지 못하므로 주의해야한다.

2 CHAPTER 12 정적 변수와 레지스터 변수

01 기억부류

02 키워드 auto

03 키워드 register

04 키워드 extern

05 키워드 static

01 기억 부류 (Storage Class)

기억 부류의 개념

기억 부류란 메모리가 할당되는 장소와 제거 시기를 결정하는 요소의 집합이다.

기억 부류의 종류

기억부류의 종류는 auto, register, static, extern 이렇게 4가지이며
12-1장에 나왔던 전역 변수와 지역 변수는 이 기억부류의 상위 개체로서 존재한다.

기억 부류에 따른 전역와 지역의 유효 범위는 다음과 같이 auto와 register는
지역변수로, extern은 전역변수로, static은 둘 다 속해 있다.

기억 부류 종류	전역	지역
auto	X	○
register	X	○
extern	○	X
static	○	○

기억 부류의 사용 방법

```
기억부류 자료형 변수명;
기억부류 자료형 변수명 = 초기값; //extern 제외
```

이제 위 내용을 기억하며 각 기억 부류의 대한 설명으로 넘어가 자세히 알아보도록 하자.

02 키워드 auto

auto는 12-1장에서 언급했던 것처럼 일반 지역변수의 의미이며, 지역변수가 자동 변수라고 불리우는 특징에 대한 기능을 하게 해주는 키워드이다.

지역 변수 선언 시, 따로 기억부류를 정의하지 않는다면 자동으로 지역 변수에 설정되는 키워드이기 때문에 생략 가능하므로 굳이 명시적으로 정의하지 않는다.

03 키워드 register

레지스터는 CPU내부의 기억 영역을 의미한다.

보통 변수는 일반 메모리로부터 메모리가 할당되지만, 레지스터 변수는 레지스터 영역으로부터 메모리가 할당된다.
레지스터는 CPU내부의 기억영역이기 때문에 이렇게 일반 메모리가 아닌 레지스터 영역의 메모리를 이용할 경우 처리 속도가 상대적으로 빠르다는 장점이 존재한다.

지역 변수에만 이용 가능한 키워드이기 때문에 함수나 블록의 종료로 인한 참조가 불가능한 상태가 되면 레지스터 영역에서 할당된 메모리가 소멸되는 특징과 초기값의 자동 지정이 되지않는다는 특징을 가진다.

일반 메모리에 할당되는 변수가 아니므로 주소연산자 &을 사용할 수 없으며 레지스터 영역에서 할당될 영역이 존재할 경우 일반 메모리로 할당이 되는 단점이 있다.
그렇기 때문에 레지스터 변수는 주로 반복문의 횟수를 제어하는 제어변수로 이용이된다.

04 키워드 extern

12-1장에서 언급했던 것과 같이 다른 파일에서 선언된 전역변수를 참조하기 위해 사용하는 키워드이다.

이미 다른 파일에서 선언된 전역 변수이고 이건 새로 생성하는 변수가 아니라 다른 파일에 선언된 전역 변수를 가져다 쓰겠다는 것을 명시해 주는 기능을 한다.

이미 존재하는 전역변수의 범위를 확장하는 개념이므로 선언 시 데이터 타입의 생략이 가능하지만, 다른 곳에 이미 선언되고 초기화된 변수를 이용하는 것이므로 초기값을 지정할 수 없다.

05 키워드 static

static의 개념

지역과 전역 모든 변수에 이용 가능한 키워드이다.

한 번 생성되면 프로그램 자체가 종료될 때까지 메모리에서 제거되지않아 지속적으로 저장 값을 유지하거나 수정이 가능하다. 즉, 해당 키워드가 전역이든 지역이든 함수가 종료되더라도 메모리가 여전히 할당되어 있음을 말한다.

또한, 전역과 지역 상관없이 초기값을 지정하지 않으면 전역 변수처럼 자동으로 초기화되는 특징을 가진다. 하지만, 초기화가 최초 1회만 실행이 돼 실행 중간에 더이상 초기화되지 않으므로 초기화 시 변수를 대입하게 되면 해당 변수가 변해 오류가 발생한다. 그러므로 초기값을 상수로 지정해야한다는 주의점이 존재한다.

선언 위치에 따라 변수의 사용범위를 전역(함수 외부)과 지역(함수 내부)로 한정할 수 있으며 그에 대한 내용은 다음 내용에 나온다.

정적 전역 변수

함수 외부에서 선언되는 전역 변수를 static으로 선언할 경우, 정적 전역 변수가 된다.

보통의 전역 변수는 다른 파일에 존재하는 전역변수를 참조할 때 extren키워드를 사용했었지만, static의 최초1회만 초기화되는 특성으로 인해 다른 파일에선 static으로 선언된 전역변수를 참조하지 못한다. 즉, 정적 지역 변수는 선언된 파일에서만 사용할수있는 유효범위를 갖는다.

정적 지역 변수

함수나 블록에서 선언되는 지역 변수를 static으로 선언할 경우, 정적 지역 변수가 된다. 보통 지역 변수와는 다르게 선언된 블록이 끝나도 메모리가 남아있으며, 초기값을 지정하지 않아도 자동으로 초기값이 지정된다.

3 CHAPTER 12 메모리 영역과 변수 이용

01 메모리 영역

02 유효 범위와 기억부류 별 변수의 특징

01 메모리 영역

메모리 영역

메모리 영역이란 변수 선언 시, 변수에게 할당되는 메모리 영역이라는 뜻을 포함한다. 12-2장에서 설명한 기억 부류의 결정에 따라 할당될 메모리 영역이 부여되므로 변수의 유효 범위(scope)와 생존 기간(life-time)을 결정하는 중요한 역할을 한다.

메인 메모리 영역은 낮은 메모리 주소를 가지고 있는 순대로 데이터 영역, 힙 영역, 스택 영역 이렇게 크게 3가지로 나뉜다.

낮은 주소		높은주소
데이터 영역	힙 영역	스택 영역

데이터 영역

데이터 영역에는 전역변수와 정적 변수가 할당된다. 즉, 프로그램 시작부터 종료 때까지 메모리 영역을 고정적으로 차지하고 있는다는 뜻이다. 그렇기 때문에 프로그램이 시작되는 시점에 정해진 크기대로 고정된 메모리 영역 확보되어 가장 낮은 주소의 데이터 영역을 소유한다.

힙 영역

메모리 주소가 낮은 값에서 높은 값으로 사용하지 않는 공간이 차례대로 동적할당(dynamic allocation)되는 변수가 할당된다.

스택 영역

함수 호출에 의한 형식 매개변수 와 함수 내부의 지역 변수(정적 지역 변수 제외)가 할당된다.

함수 호출과 종료에 의해 높은 주소에서 낮은 주소로 메모리가 할당되었다 소멸되며 그에 따라 영역의 크기가 변하고, 만약 함수가 없다면 스택영역이 없을 수도 있다.

02 유효 범위와 기억부류 별 변수의 특징

이번 12장은 전체 내용이 유기적으로 연결되기 때문에 표를 통해 마지막 정리를 한다.

<table>
<tr><th>기억
부류</th><th>유효
범위</th><th>상세
유효범위</th><th>상세 종류</th><th>초기 값</th><th>초기 값
할당시기</th><th>메모리
영역</th><th>메모리
할당시기</th><th>메모리
생존시간</th><th>메모리
제거시기</th><th>동일
파일
외부 함수
에서의
이용</th><th>다른
파일
외부 함수
에서의
이용</th></tr>
<tr><td>Extern</td><td>프로그램
전역</td><td rowspan="2">전역</td><td>전역
변수</td><td rowspan="3">자동
부여</td><td rowspan="3">프로그램
시작</td><td rowspan="3">데이터</td><td rowspan="3">프로그램
시작</td><td rowspan="3">프로그램
실행
시간</td><td rowspan="3">프로그램
종료</td><td rowspan="2">O</td><td>O</td></tr>
<tr><td rowspan="2">Static</td><td>파일
내부</td><td>정적
전역변수</td><td rowspan="3">X</td></tr>
<tr><td rowspan="3">함수
(블록)
내부</td><td rowspan="3">지역</td><td>정적
지역변수</td><td rowspan="2">X</td></tr>
<tr><td>Register</td><td>레지스터
변수</td><td rowspan="2">쓰레기 값</td><td rowspan="2">함수
(블록)
실행 시
마다</td><td>레지스터</td><td rowspan="2">함수
(블록)
시작</td><td rowspan="2">함수
(블록)
실행
시간</td><td rowspan="2">함수
(블록)
종료</td></tr>
<tr><td>auto</td><td>자동
지역변수</td><td>스택</td></tr>
</table>

CHAPTER

13

구조체와 공용체

13.1 구조체와 공용체

13.2 자료형 재정의

13.3 구조체와 공용체의 포인터와 배열

1 CHAPTER 13 구조체와 공용체

01 구조체

02 공용체

01 구조체

개념

구조체란 일반적으로 사용하는 개별적인 변수들(정수, 문자, 실수, 포인터, 배열 등)을 하나의 단위로 묶은 자료형이다.

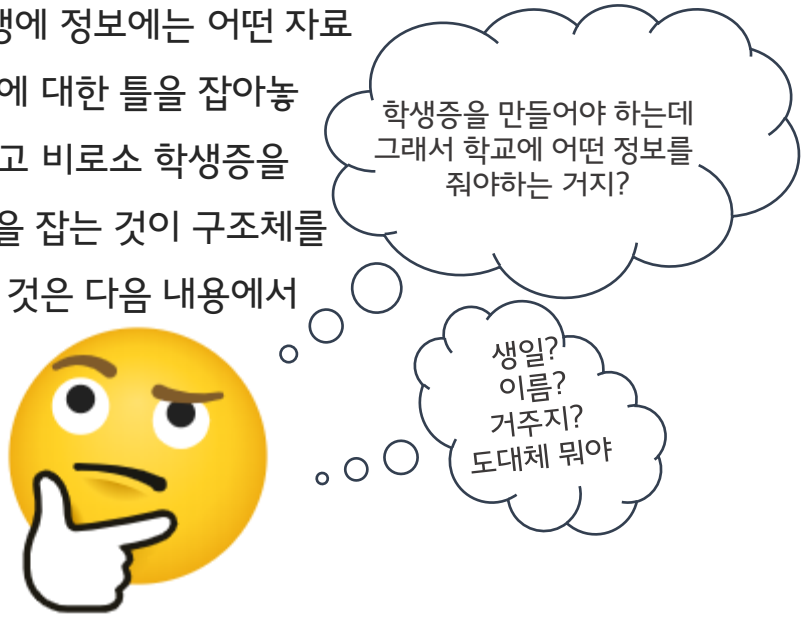
예를 들어, 학생의 정보 안에는 이름, 학과, 학번의 정보가 들어간다. 여기서 학생이 구조체가 되고 이름, 학과, 학번이 구조체 속 일반적인 변수들을 의미한다.

구조체는 기존 자료형의 집합을 만들며 새로 만들어진 자료형이므로 유도 자료형이라고도 불리운다.

TIP 유도 자료형 : 기존 자료형 구성의 집합체로 새로 만들어진 자료형

구조체는 어디까지나 자료형 중에 하나이지만, 기존 자료형과 같이 바로 사용할 수 없다. 바로 구조체를 먼저 정의해 주어야하기 때문이다.

앞서 언급한 학생의 정보로 계속 이야기를 하자면 학생A가 존재하고 학생의 정보가 들어간 학생증을 발급한다고 가정했을 때 학생증 안에는 학생의 정보에 해당하는 특정한 자료들이 들어간다. 이때, 학생에 정보에는 어떤 자료들이 있는지를 먼저 생각해 학생정보에 대한 틀을 잡아놓아야 특정한 자료가 무엇인지 파악하고 비로소 학생증을 만들 수 있게 된다. 여기서 말하는 틀을 잡는 것이 구조체를 정의하는 작업이 되시겠다. 정의하는 것은 다음 내용에서 알아보도록 하자.



정의

키워드 struct를 이용해 구조체 영역을 생성하고, 영역 안에 다양한 변수를 생성한다.

이때 다양한 변수를 멤버 혹은 필드라 칭하며 자료형은 일반 변수부터 포인터, 배열, 다른 구조체 변수, 구조체 포인터 등 다양하게 생성이 가능하다. 하지만 초기값의 지정이 불가하다.구조체를 정의하는 것 또한 하나의 문장이므로 마지막(중괄호 끝)에 세미콜론(;)을 붙여여 한다.

정의 위치는 함수 내부, 외부 정의 둘다 가능하다. 내부에 정의된다면 지역 유효 범위에 정의될 것이고, 외부에 정의된다면 전역 유효 범위에 정의될 것이다.

사용 방법은 다음과 같다.

```
struct 구조체태그명{
자료형 변수명; // 멤버(필드)를 선언
}; //세미콜론 필수
```

```
struct stc{
int a;
char b,c,d // 같은 자료형이 연속되는 경우 콤마(,)로 정의 가능
};
```

이렇게 틀을 만드는 작업, 구조체 정의를 성공적으로 끝마친 경우,
struct 구조체태그명이 하나의 자료형으로 새로이 생성된 것이다.
이젠 만든 구조체를 어떻게 변수로 선언해 사용해야 할지 살펴보도록 하자.

변수 선언

struct 구조체태그명 을 하나의 자료형으로 생각하고 일반 변수를 선언 하는 것과 같이 다음처럼 선언하면 된다.

```
struct 구조체태그명 구조체변수명;
// struct 구조체태그명 이 하나의 자료형으로 인식
```

위가 가장 일반적인 방법이지만 앞서 설명한 구조체를 정의할 때 함께 변수로서의 선언이 가능하다.

그 방법은 다음과 같다.

```
struct 구조체태그명{
자료형 변수명;
} 구조체변수명; //구조체의 끝에서 변수를 선언
```

다음과 같이 구조체태그명을 쓰지 않는 경우도 있다.

```
struct{ //구조체태그명X
자료형 변수명;
} 구조체변수명; //구조체의 끝에서 변수명 명시
```

이 경우엔 구조체에 해당하는 변수를 여러 개 선언할 수 없고 맨 끝에 정의된 변수만 해당 구조체의 변수로서 선언되어 사용한다.

만약 구조체태그명도 안쓰고 구조체변수명도 명시하지 않는 경우, 해당 구조체를 이용할 방법이 없기 때문에 주의하길 바란다.

변수 초기화

구조체변수에 값을 넣는 방법은 두가지이다.
첫번째는 선언과 동시에 초기화하는 것이고, 두번째는 선언 후 초기화하는 방법이다.

선언과 동시에 초기화하는 것은 앞서 설명한 변수 선언에서 나아가 중괄호로 초기화를 하면 된다. 단, 구조체필드에 정의된 순서대로 써야한다.

방법은 다음과 같다.

```
struct 구조체태그명 구조체변수명 = {값1, 값2, 값3...};
```

선언 후 초기화하는 방식은 다음과 같다.

```
구조체변수명.구조체필드명 = 값1;
```

접근연산자(.)를 이용한 방식으로 선언한 구조체변수명을 참조한 뒤, 구조체 내부 필드로 접근해 값을 넣는다.

구조체 크기

생성한 구조체의 크기를 알아야할 경우 sizeof()함수를 이용한다.

sizeof()는 구조체에만 국한되는 것이 아닌 전달인자에 대한 바이트크기값을 알고 싶을 때 사용하는 함수이다.

이렇게 sizeof()함수를 사용할 경우 산술적으로 계산한 바이트 크기보다 즉, 필드로 선언된 자료형의 크기의 합보다 크거나 같게 나온다.

크게 나오는 이유는 시스템은 정보를 전송할 때 4바이트 혹은 8바이트 단위로 전송을 처리하기 때문에 컴파일러가 이러한 시스템에 맞추어 구조체에게 메모리를 할당하다 보면 중간에 사용하지 않는 바이트를 할당하는 경우가 존재하기 때문이다.

그에 대한 예는 다음과 같다.

```c
#include <stdio.h>

struct MyStruct //12byte
{
    int  a, b, c; //12byte
};
struct MyStruct2 //36byte
{
    struct MyStruct a;  //12byte
    char b[12];  //12byte
    int c;  //4byte
    double d;  //8byte
};

int main(void) {
    struct MyStruct stc;
    struct MyStruct2 stc2;
    printf("%d\n", sizeof(stc));
    printf("%d",sizeof(stc2));

    return 0;
}
```

```
12
40
```

MyStruct와 MyStruct2가 정의되어 있으며, 각각의 구조체의 바이트 할당 값을 계산해보면 12바이트와 36바이트가 나온다.

그리고 sizeof()함수를 사용해 구조체의 크기를 출력해보면 MyStruct의 구조체 변수인 stc는 예상한 계산대로 12바이트가 출력되지만 MyStruct2는 40바이트가 출력되는 것을 알 수 있다.

하지만, double d를 지울 경우 36byte에서 8byte를 뺀 값인 28byte가 제대로 출력되므로 이를 통해 구조체의 크기 할당이 매우 유동적인 것을 알 수 있다.

구조체 변수의 내용 비교

구조체의 내용, 즉 필드의 값에 대해 동등연산자를 이용해 비교를 할 경우 구조체 자체를 비교하는 것이아니라 접근연산자(.)를 이용해 각각의 필드값에 대해 비교를 해야한다.

방법은 다음과 같다.

```
구조체변수명1.구조체필드1 == 구조체변수명2.구조체필드1
```

오류가 나는 경우는 다음과 같이 접근 연산자를 사용하지 않고 구조체만을 참조하려 했을때 이다.

```
구조체변수명1 == 구조체변수명2 //구조체만을 참조하려 함
```

02 공용체

개념

공용체란 말 그대로 공용으로 저장공간을 사용하는 객체이다.
이번엔 사진과 함께 이해해 보자.

사진과 같이 공용으로 사용하는 메모리 공간을 생성하는데 여기서 공용으로 사용하는 메모리 공간이 공용체이다.

공용체의 사용 가능 대상은 구조체에 멤버(필드)를 정의 했을 때와 같이 자료형의 구분없이 지정할 수 있다. 사진에 1, 2, 3 이 공용체의 멤버라고 생각하면 된다. 이 여러 공용체 멤버들이 메모리 공간에 할당되는 즉, 사용 중인 상태로 변하는 기준은 값의 저장이다. 가장 마지막에 값을 저장한 멤버만이 공용 메모리 공간을 할당받고 값을 참조할 수 있다.

예를 들어 1번 멤버가 값을 마지막으로 저장한 멤버라면 1번 멤버의 값만 유효하고 나머지 2,3번 멤버는 메모리 공간을 할당받지 않았기 때문에 그 이전에 2, 3번 멤버에 값을 저장했더라도 의미없는 값이 저장된다. (한마디로 쓰레기 값 저장. 사용 가치가 없다.)

정의

전체적인 구조는 구조체를 정의했을 때와 같으며, struct 키워드를 union키워드로 바꾸어 정의하면 된다.

방법은 다음과 같다.

```
union 공용체태그명{
자료형 변수명;//멤버
};
```

변수 선언

구조체와 마찬가지로 공용체태그명 을 하나의 자료형으로 생각하고 일반 변수를 선언하는 것과 같이 다음처럼 선언하면 된다.

```
union 공용체태그명 공용체변수명;
// union 공용체태그명 이 하나의 자료형으로 인식
```

위가 가장 일반적인 방법이지만 앞서 설명한 공용체를 정의할 때와 함께 변수로서의 선언이 가능하다.

그 방법은 다음과 같다.

```
union 공용체태그명{
자료형 변수명;
} 공용체변수명; //공용체의 끝에서 변수를 선언
```

공용체태그명을 쓰지 않는 경우도 있다. 그 예는 다음과 같다. 이 경우엔 구조체 때의 설명과 마찬가지로 다른 곳에서 참조할 수 없으니 다른 곳에서 생성하지 못해 정의와 동시에 공용체변수명을 꼭 명시해주어야한다.

```
union { //공용체태그명이 없음
자료형 변수명;
} 공용체변수명; //꼭 필요
```

변수 초기화

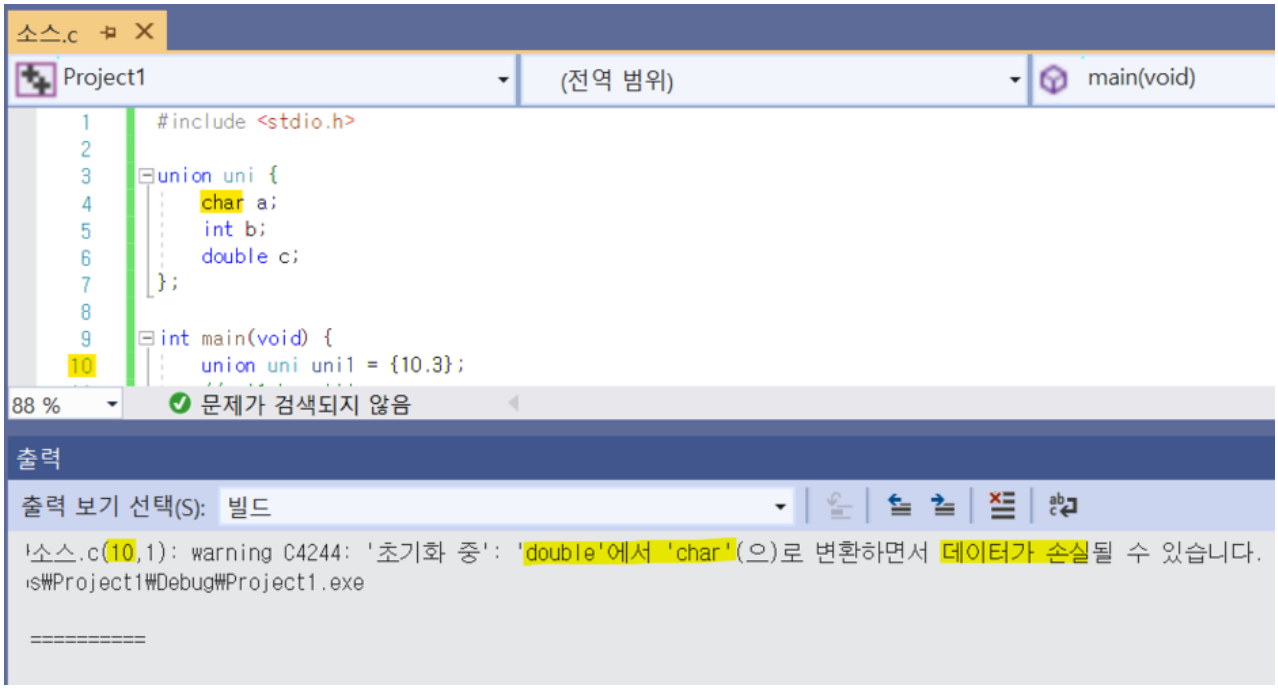
공용체변수에 값을 넣어 초기화하는 방법은 두가지이다.
첫번째는 선언과 동시에 초기화하는 것이고, 두번째는 선언 후 초기화하는 방법이다.

선언과 동시에 초기화하는 것은 앞서 설명한 변수 선언에서 나아가 중괄호로 초기화를 하면 된다. 단, 공용체의 특성에 맞게 하나의 값만을 초기화할 수 있으나 이처럼 동시에 선언할 경우엔 가장 첫번째 멤버에 값의 초기화가 된다.

방법은 다음과 같다.

```
union 공용체태그명 공용체변수명 = {첫번째멤버의값};
```

만약 첫번째 멤버에 해당하지 않는 값을 초기화할 경우엔 다음과 같은 문구가 뜬다.



▲ c에 해당하는 값을 초기화 했더니 a의 자료형으로 강제 변환을 한다.
첫번째 멤버에 값을 저장하려함을 알 수 있다.

다음 선언 후 초기화하는 방식은 다음과 같다.

```
공용체변수명.공용체필드명 = 값;
```

접근연산자(.)를 이용한 방식으로 선언한 공용체변수명을 참조한 뒤, 공용체 내부 필드로 접근해 값을 넣는다.

공용체의 크기

공용체의 크기는 멤버로 선언된 것 중 가장 큰 자료형을 기준으로 정해진다고 한다. 교재와 인터넷을 통해 찾아본 결과 구조체의 크기는 달라진다는 말을 많이 보았지만, 공용체의 크기는 큰 자료형을 기준으로 한다는 말 밖에 보지 못했다.

하지만 여러 코드를 실행해본 결과, 구조체와 똑같이 유동적으로 가장 큰 자료형의 크기와 같거나 더 크게(4byte, 8byte단위) 메모리가 할당이 된다는 결론을 내었다.

실험 코드는 다음과 같다.

```
union uni
{
    int  a; //4byte
    char b; //1byte
    char e[13]; //13byte
};
```

가장 큰 자료형을 기준으로 정해진다는 이론적인 입장에서 봤을 땐 sizeof()사용 시 공용체 uni의 크기는 13byte가 뜨는 것이 이상적이다.

하지만 16byte가 할당되었고, int a가 사라져야 13byte의 결과를 확인할 수 있었다.

그 다음으로는 위에서 구조체의 크기를 알아본 코드를 공용체로 바꾸어 실행을 해보았다. (char배열의 크기는 5로 변경하였다)

```
union uni //4byte
{
    int  a, b, c; //4byte
};
union uni2 //5byte
{
    union uni;  //4byte
    char d[5];  //5byte
    int e;  //4byte
};
```

위에 정의된 uni는 4byte로 출력이 되었지만, uni2는 8byte로 나왔고, char d를 제외한 모든 항목을 지워야 5byte로 나오는 것을 확인할 수 있었다.

위 코드들의 결과를 바탕으로 기본적인 자료형의 베이스가 다르면,
4byte(8byte)단위로 메모리가 더 크게 할당되는 것인가라는 생각이 들었다.

하지만 다음의 공용체의 크기에 대한 결과가 내 생각을 바꾸어 주었다.

```
union uni //4byte
{
    int  a, b, c; //4byte
};
union uni2 //12byte
{
    union uni;  //4byte
    char d[12];  //12byte
    int e;  //4byte
    double f;  //8byte
};
```

바로 이전 코드에서 d의 크기를 5에서 12로 바꾼 코드이다.

uni2는 산술적으론 12byte이지만 16byte가 나온다.

이전 결과들에 대한 내 생각에 따르면 12byte로 제대로 나오는 경우는 d를 제외한 모든 멤버를 지웠을 때이다.

하지만 doule f를 지웠을 때 12byte가 나오는 것을 확인할 수 있었다. 다른 자료형의 존재는 상관이 없는 것이다. 즉, 공용체의 메모리 할당은 무조건 제일 큰 멤버의 크기가 아닌 제일 큰 멤버의 크기가 기준이긴 하지만 구조체의 메모리 할당처럼 유동적으로 이루어진다는 결론에 도달했다.

아래는 공통으로 사용한 main함수의 코드이다.

```
int main(void) {
    union uni uni1;
    union uni2 uni2;
    printf("%d\n", sizeof(uni1));
    printf("%d\n", sizeof(uni2));

    return 0;
}
```

2 CHAPTER 13 자료형 재정의

01 키워드 typedef

01 키워드 typedef

개념

typrdef키워드는 이미 사용되는 자료 유형을 다른 새로운 자료형의 이름으로 재정의하는 키워드이다.

typedef키워드가 생겨난 이유는 프로그램 시스템 간의 호환성과 편의성을 위해서이다. 개발 환경에 따라 자료형의 저장 공간이 달라지는데 이때 변수 저장공간에 대한 오버플로우 문제가 발생한다. 이러한 오버플로우 문제를 해결하며 서로 다른 개발환경에도 호환이 되도록 자료형을 재정의하는 키워드를 개발했고 그것이 바로 typedef인 것이다.

사용 방법

typedef키워드로 재정의 시 하나의 새로운 이름이 아닌 여러 개의 이름으로도 사용이 가능하며 재정의를 해도 기존 자료 유형과 새로운 이름 둘다 사용할 수 있다.

typedef키워드는 다른 변수 선언과 마찬가지로 유효 범위에 대한 영향을 받는다. 만약 typedef를 함수 내부에서 사용할 경우, 해당 지역 유효 범위에서만 재정의가 적용되고 함수 외부에서 사용할 경우 전역 유효 범위에서 재정의가 적용된다.

typedef 키워드의 사용은 다음과 같다.

```
typedef 기존자료유형 새로운이름;
typedef 기존자료유형 새로운이름1,새로운이름2...;//여러 개 생성 가능
```

```
typedef int DearD; //이제 자료형 int를 int와 DearD로 사용할 수 있다
```

활용

지난 13-1장에서 구조체와 공용체 정의 시, 변수 선언을 할 때 키워드 태그명이 하나의 자료형으로 다음과 같이 인식된다고 언급했었다.

```
struct 구조체태그명 구조체변수명;
// struct 구조체태그명 이 하나의 자료형으로 인식
```

```
union 공용체태그명 공용체변수명;
// union 공용체태그명 이 하나의 자료형으로 인식
```

하지만 변수 선언 시 마다 저렇게 자료형을 입력하는 것은 손이 많이 간다. 그러므로 typedef 키워드로 위 자료형을 새로운 자료형명으로 만든다면 훨씬 편하게 사용할 수 있을 것이다.

그 사용의 예는 두가지 방법이 있다.
정의 후 자료형을 재정의 하는 방법과 정의와 동시에 재정의 하는 방법이다.

먼저 정의 후 자료형을 재정의 하는 방법은 다음과 같다.
(단, 구조체(or공용체)가 정의되어있다는 전제하에)

```
typedef struct 구조체태그명 새로운이름;
```

```
typedef union 공용체태그명 새로운이름;
```

정의와 동시에 재정의하는 방법은 다음과 같다.

```
typedef struct 구조체태그명{ //구조체태그명 생략 가능
//멤버
}새로운이름;
```

```
typedef union 공용체태그명{ //공용체태그명 생략 가능
//멤버
}새로운이름;
```

이 경우 어차피 해당 객체에 대한 새로운 이름이 정의되므로 태그명의 생략이 가능하다.

생략할 경우, 태그명이 정의되지 않았으므로 struct 구조체태그명(or union 공용체태그명)이라는 자료형을 사용할 수 없다.

3 CHAPTER 13 구조체와 공용체의 포인터와 배열

01 구조체와 공용체의 포인터

02 구조체의 배열

01 구조체와 공용체의 포인터

개념

포인터는 각각의 자료형 저장 공간의 주소를 저장하는 변수이다.
구조체와 공용체의 주소 값을 저장하는 변수인 포인터로 생성해 사용해 보자.

선언

구조체와 공용체의 포인터 변수를 선언하는 방법은 세가지가 있다.
정의와 동시에 포인터 변수를 선언하는 것과 정의 후 변수를 선언하기 전 포인터 변수로 선언하는 것, 변수 선언 후 해당 변수를 이용해 포인터 변수로 선언하는 방법이다.

정의와 동시에 포인터 변수를 선언하는 것은 지난 13-1에서 정의와 동시에 변수명을 지정하는 방식과 같다. 그에 대한 예는 다음 코드이다.

```
struct 구조체태그명{
자료형 변수명;
} *구조체포인터변수명; //구조체의 끝에서 포인터 변수를 선언
```

```
union 공용체태그명{
자료형 변수명;
} *공용체포인터변수명; //공용체의 끝에서 포인터 변수를 선언
```

다음은 정의 후 변수를 선언하기 전 포인터 변수로 선언하는 방법이다.

```
struct 구조체태그명 *구조체포인터변수명;
구조체명 *구조체포인터변수명; //typedef를 사용했다면
```

```
union 공용체태그명 *공용체포인터변수명;
공용체명 *공용체포인터변수명; //typedef를 사용했다면
```

다음 변수 선언 후 해당 변수를 이용해 포인터 변수로 선언하는 방법은 해당 변수의 주소값을 초기값으로 지정하면 된다.

방법은 다음과 같다.

```
struct 구조체태그명 *구조체포인터변수명 = &구조체변수;
구조체명 *구조체포인터변수명 = &구조체변수; //typedef를 사용했다면
```

```
union 공용체태그명 *공용체포인터변수명 = &공용체변수;
공용체명 *공용체포인터변수명 = &공용체변수; //typedef를 사용했다면
```

참조

생성한 포인터 변수를 사용하는 가장 익숙한 방법은 참조연산자(.)를 이용해 구조체의 멤버에 다음과 같은 방식으로 접근하는 것이다.

```
(*구조체포인터변수명).구조체멤버명
```

```
(*공용체포인터변수명).공용체멤버명
```

이렇게 단순히 포인터변수와 같이 참조연산자(.)를 이용해도 되지만 간접 연산자(->)를 이용할 수 도 있다.

간접연산자인(->)란, 구조체와 공용체의 접근하는 포인터를 위한 접근 연산자이다.
사용시 -와 >사이에 공백이 존재해선 안된다.
이러한 간접 연산자를 이용해 구조체 포인터 변수의 속하는 구조체 멤버에 접근하려고 할 때, 접근 방법은 다음과 같다.

```
구조체포인터변수명->구조체멤버명
```

```
공용체포인터변수명->공용체멤버명
```

사실 포인터의 개념이 어렵기 때문에 이해하기 힘이 든다.

다음 구조체를 기준으로 제작한 예제와 예제의 적혀진 주석을 보며 개념을 더 확립해 보자.

```c
#include <stdio.h>

typedef struct{
    char a[20];
    int b;
}stc;

int main(void) {
    stc var = { "DearD", 2}; //구조체 변수
    stc* pointer = &var; //구조체 포인터 변수

    printf("%s\n", var.a); //변수라 가능
    //printf("%s\n", pointer.a);포인터라 불가능(변수pointer로 인식)

    //원래는 이렇게 써야하지만
    printf("%s\n", (*pointer).a);
    //포인터를 위한 참조연산자(->)를 사용하면, 변수명만으로도 가능
    printf("%s\n", pointer->a);

    return 0;
}
```

```
DearD
DearD
DearD
```

올바르지 못한 참조

다음은 위에서 진행했던 멤버에 접근하려는 의도와 부합하지 않거나 오류가 발생하는 접근의 경우이다. 관련 예제를 통해 알아보자.

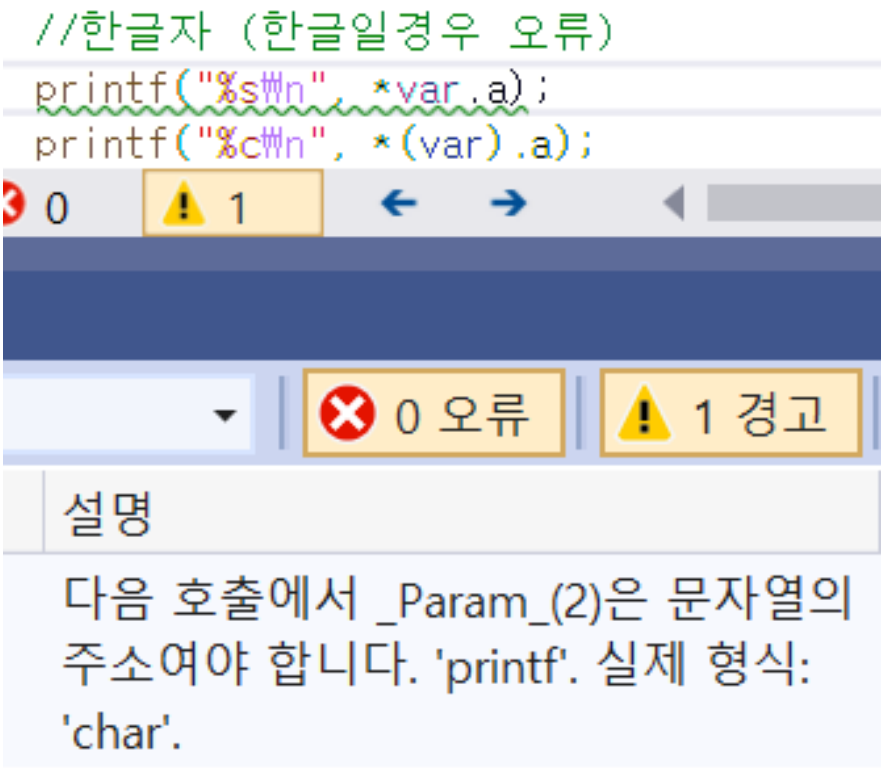
먼저 접근하려는 의도와 부합하지 않은 경우이다. 관련 예제는 다음과 같다.

```c
#include <stdio.h>

typedef struct {
    char a[20];
    int b;
}stc;

int main(void) {
    stc var = { "DearD", 2};
    stc* pointer = &var;

    //한글자 (한글일 경우 오류)
    printf("%c\n", *var.a);
    printf("%c\n", *(var).a);
    printf("%c\n", *(var.a));

    printf("%c\n", *pointer->a);
    printf("%c\n", *(pointer)->a);
    printf("%c\n", *(pointer->a));

    return 0;
}
```

```
D
D
D
D
D
D
```

우리는 한글자가 아니라 글자 전체, 멤버의 값을 보여주고 싶어한다. 하지만 위 와 같은 접근을 하게될 경우, 최종적으로 멤버(문자배열이므로)의 0번째 원소에 접근, 1byte를 출력하게 된다.

그에 대한 증거로 두번째 인자가 char형이므로 바꾸어야한다는 경고 메시지가 다음과 같이 발생한다.



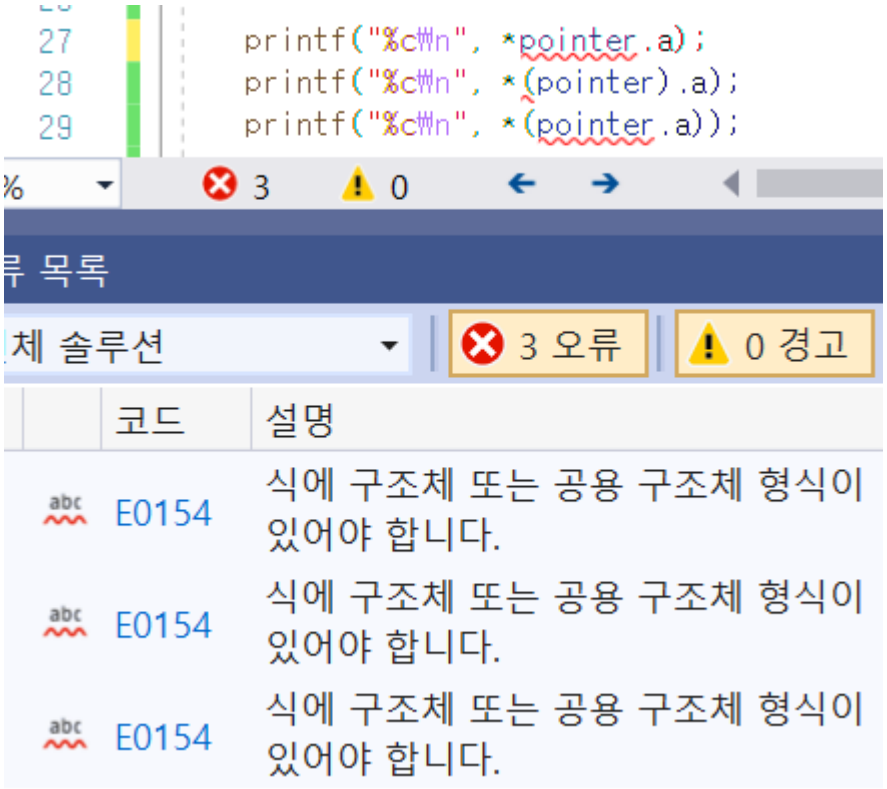
▲ 경고 메세지

만약 멤버의 값이 한글일 경우, 1byte를 출력하려하기 때문에 실행 오류가 발생한다.

▲ 멤버의 저장 값을 한글로 변경할 경우 글자가 깨지며 코드의 목적을 제대로 수행하지 못한다.

다음은 컴파일 오류가 나는 코드이다.

```
printf("%c\n", *pointer.a);
printf("%c\n", *(pointer).a);
printf("%c\n", *(pointer.a));
```



▲ 포인터의 접근을 잘못해 컴파일 오류 메시지가 발생한다.

쭉 구조체에 대한 예로 설명을 했지만 공용체의 접근도 마찬가지이다. 위 예제에서 struct키워드를 union키워드로 변경하면 똑같은 결과의 확인이 가능하다.

02 구조체의 배열

구조체 배열은 구조체를 배열로 선언하며, 한번에 여러 개의 구조체를 이차원 배열의 형식으로 생성하는 방식이다.

하나하나 구조체를 만들었던 지난 방식에서 구조체 배열로 생성하게 되면 더 편리한 생성과 관리가 가능하다.

단순히 구조체 자료형의 값을 이차원 배열로 생성한다고 생각하면 되며, 생성 방법과 참조 방식은 다음과 같다.

```
struct 구조체태그명 구조체배열명[] = {{값1,값2..},{값1,값2..}...};

구조체배열명[참조하려하는구조체가담긴원소의위치].구조체멤버명;
```

공용체는 값을 하나만 저장할 수 있으므로 배열은 의미가 없다.

다음 예제를 통해 구조체의 배열을 더 알기 쉽게 이해해보자.

```c
#include <stdio.h>

typedef struct {
    char a[20];
    int b;
}stc;

int main(void) {
    //이차원 배열로 생성
    stc var[] = { {"DearD", 2}, {"love", 2}, {"C", 2}};
```

```
    //각각의 원소의 구조체멤버를 참조해 사용
    printf("%s ", var[0].a);
    printf("%s ", var[1].a);
    printf("%s\n", var[2].a);

    return 0;
}
```

```
DearD love C
```

구조체를 2차원 배열로 생성하고, 각각의 원소를 참조 연산자로 참조해 출력하면 제대로 출력되는 것을 확인할 수 있다.

CHAPTER

14

함수와 포인터 활용

14.1 함수의 인자전달 방식
14.2 포인터 전달과 반환
14.3 함수 포인터와 void 포인터

1 CHAPTER 14 함수의 인자전달 방식

01 값에 의한 전달 방식

02 참조에 의한 전달 방식

03 가변인자 함수의 전달 방식

지금까지 우리가 사용했던 함수들을 생각해보면 함수의 괄호 안에 인자를 넣어서 전달을 했었다.

이번엔 그 인자에 대한 내용을 하려고 한다.

함수에 인자가 쓰여지면 어떻게 해당 함수로 넘어가서 실행이 되는지 전달 값에 따라 차이점이 있는지 대표적인 전달방식을 먼저 알아보고, 그 외의 전달인자에 대해 알아보도록 하자.

01 값에 의한 전달 방식 (Call by value)

값에 의한 전달 방식(call by value)는 가장 대표적인 인자 전달방식이다.

함수 호출 시, 실인자값이 형식 인자에 복사되어 저장된다는 의미로 함수로 넘어가서 인자의 내용을 수정하여도 그것은 형식인자가 바뀌는 것이지 실인자가 바뀌는 것이 아니기때문에 함수에서 수정을 한다고 본래의 값이 바뀌지 않는다. 즉, 함수 외부의 변수를 함수 내부에서 수정이 불가하다는 뜻이다.

코드로써의 설명은 다음과 같다. 주석과 함께 이해해 보자.

```
void plus(int a,int b){ //형식 인자 a, b
a+=b; //2. 실인자a가 아닌 형식인자의 a가 바뀜
}

int main(void){
int a=10; //실인자 a
plus(a,20); //1. 인자로 전달된 a는 형식 인자에 복사되어 전달
//3. main함수에서의 a는 그대로 10
return 0;
}
```

02 참조에 의한 전달 방식 (Call by reference)

일반 변수의 전달

참조에 의한 전달 방식(call by reference)은 값에 의한 전달 방식과 달리 일반 변수가 아닌 포인터를 전달하므로 결국 인자의 주소 값을 전달하게 된다. 이 때 주소 값을 전달하는 것을 참조한다고 한다.

실인자가 형식인자에 복사되어 다른 객체로 분류되는 것이 아니라 실인자의 주소를 전달하게 되므로 함수에서 실인자의 값을 바꿀 수 있게 된다.

함수를 호출해 참조에 의한 전달 방식으로 일반 변수를 전달할 때는 변수의 주소값을 전달하기 위해 주소연산자(&)를 사용한다. 변수의 앞에 주소 연산자를 붙이면 해당 변수의 주소값을 가리키게 된다.

코드로써의 설명은 다음과 같다. 주석과 함께 이해해 보자.

```
void plus(int* a,int b){ //실인자 a를 포인터로서 저장
a+=b; //2. a의 주소 값의 내용을 변경 수정됨
}

int main(void){
int a=10; //실인자 a
plus(&a,20); //1. a의 주소 값이 전달
//3. main함수에서의 a는 10+20=30이 됨
return 0;
}
```

배열 이름으로 전달

위 참조에 의한 전달 방식은 일반적인 자료형 변수의 주소값을 전달을 했다면 이젠 배열로써 배열 원소의 주소값을 전달해 참조에 의한 전달방식을 설명하겠다.

배열을 생성하게 되면 0번째 원소 주소의 다음 주소는 1번째 원소 값을 가리키고, 그 다음 주소도 마찬가지로 다음 원소 값을 가리키게 된다.

이러한 배열을 인자로 전달하게 되면 0번째 원소의 포인터 변수로 인식하기 때문에 참조하는 방식으로 값을 수정하고 이용할 수 있는 것이다.

단, 함수 내부에서 실인자로 전달된 배열의 배열 크기를 알 수 없으므로 첫번째 인자에 배열의 이름, 두번째 인자로 배열 크기를 전달하는 것이 적합하다.

해당 코드는 다음과 같다.

```
int plus(int a[],int n){ //int* a도 가능
int sum=0;
for(int i=0; i<n; i++){
sum+=a[i]; //a[0],a[1]...을 가리킴
//sum += *a++; //포인터를 이용
//sum += *(a++); //같은 기능 단 main함수에서 사용 시 상수기에 오류
//sum += *(a + i); //같은 기능
//sum += *a + i; //같은 기능
}
return sum; //2. 배열 a를 다 더한 값인 10 반환
}
```

```
int main(void){
int a[]= {1,2,3,4}; //배열 a
int aofsize = sizeof(a)/sizeof(int); //배열 크기
int sum = plus(a,aofsize); //1. 배열과 배열 크기를 전달인자로 전달
return 0;
}
```

앞서 설명했듯이 배열 a를 전달하게 되면 0번째 원소값의 주소를 포인터 변수로 인식하기 때문에 위 plus 함수헤더와 for문이 합 과정을 주석 처리한 것과 같이 해도 된다.

단, *(a++)의 경우엔 a가 주소 상수이기 때문에 외부로 주소 값을 전달해 사용하는 것은 괜찮지만, main함수에서 같은 형식으로 참조하는 경우는 오류가 발생한다.

TIP

배열 크기 구하는 법

sizeof(배열명)/sizeof(배열의자료형or배열의원소)

다차원 배열로 전달

다음은 1차원 배열이 아닌 다차원 배열을 전달 인자로 이용하는 경우이다.

이 경우는 위 1차원 배열과의 전달과 비슷하지만, 전달 인자에 있어서 차이가 있다.
1차원 배열과 마찬가치로 함수에서 해당 배열의 크기를 알지 못하기 때문이다.
그러므로 열과 행의 크기를 전달해 주어야하는 점을 잊지 말도록 하자.

사용에 대한 예제는 다음과 같다.

```c
#include <stdio.h>

void printall(int a[][2], int row, int col) { //배열, 행, 열
    //이중 반복문을 통해 각 원소 참조
    for (int i = 0; i <row; i++) {
        for (int j = 0; j < col; j++) {
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
}

int main(void) {
    //배열 a (열의 수는 꼭 기재)
    int a[][2] = { { 1,2 },{ 3,4 },{ 5,6 } };

    int rowsize = sizeof(a) / sizeof(a[0]); //행의 수 3 저장
    int colsize = sizeof(a[0]) / sizeof(a[0][0]); //열의 수 2 저장

    printall(a,rowsize,colsize); //인자로 행과 열의 크기 전달
    return 0;
}
```

```
1 2
3 4
5 6
```

sizeof()함수를 이용해 행과 열의 수를 모두 인자로 보내주었고, 2치원 배열 내 모든 원소가 출력된것을 확인할 수 있다.

03 가변 인자 함수의 전달 방식

개념

가변인자란 함수의 인자의 개수와 자료형이 변하는, 즉 결정되지 않은 인자이다.

대표적인 예로 printf함수가 있다. printf함수는 첫번째 인자의 문자열에 제어문자가 몇 개 인지, 제어 문자의 자료형이 어떻게 되는지에 따라 출력 양식과 그 뒤에 따라붙는 제어문자에 대한 값의 개수가 매번 변경된다.

이와 같이 전달 인자의 개수가 정해지지 않은 함수를 가변인자가 있는 함수라 칭한다.

그렇다면 이러한 가변인자를 가지고 있는 함수는 어떻게 헤더를 생성할지 다음 내용에서 알아보자.

가변인자가 있는 함수의 구현

가변 인자가 있는 함수 구현 시의 몇가지의 주의 점이 존재한다.

첫번째, 함수의 처음 매개 변수는 정해져 있어야 한다. 처음 매개변수는 인자의 개수와 같은 가변인자를 처리하는데 필요한 정보를 넘기는 것이 적합하다.

두번째, 마지막이 고정 매개변수일 수 없다. 처음 매개변수를 고정매개 변수로 그 이후를 가변 매개 변수를 두는데 가변 매개변수 등장 이후에 고정 매개 변수가 나타나면 안된다.

헤더의 가변인자는 ... 으로 정의하며, 구현 방법은 다음과 같다.

```
//첫 매개변수는 가변인자 처리시 필요한 정보로 지정
함수반환타입 함수명(int n, ...){ //고정매개변수 n, 가변인자 ...
   //함수 내부
}
```

가변인자가 있는 함수의 사용 – 선언(1단계)

STOP

가변 인자 사용 시, 헤더파일 <stdarg.h> 를 정의해야 한다

선언 단계는 일반적인 변수 선언과 같이 가변 인자로 처리할 변수를 정의한 함수 내부에서 선언하는 과정이다.

사용할 가변인자를 선언 시 자료형은 va_list 라는 자료형으로 선언을 한다.

선언 방법은 다음과 같다.

```
함수반환타입 함수명(int n, ...){
   va_list 선언할가변인자명; //va_list타입의 가변인자 변수 선언
}
```

가변인자가 있는 함수의 사용 – 처리 시작(2단계)

이제 선언한 가변인자 변수를 바탕으로 va_start라는 함수를 이용해 가변인자를 처리를 시작하게 된다.

va_start()는 헤더파일 <stdarg.h>에 정의된 매크로 함수로 가변인자의 시작 위치를 알리는 함수이다.

첫번째 인자에는 1단계에서 선언한 가변인자변수의 이름을, 두번째 인자로는 함수에 전달된 첫번째 고정 매개변수를 전달한다.

사용 방법은 다음과 같다.

```
함수반환타입 함수명(int n, ...){
   va_list 가변인자명;
   va_start(가변인자명, 고정인자);
}
```

가변인자가 있는 함수의 사용 – 얻기(3단계)

가변인자의 시작을 알렸으니 가변인자로 넘어온 값들을 얻을 차례이다. 3단계에서는 매크로 함수 va_arg()를 사용하게 된다.

va_arg()함수는 가변인자 각각의 자료형을 지정하여 가변인자를 하나하나 반환받는 함수이다. 하나하나 반환을 받기 때문에 보통 반복문을 이용해 가변인자를 반환받게 된다.

사용 시엔 첫번째 인자로 선언한 가변인자변수의 이름을, 두번째 인자로 가변인자를 전달받을 자료형을 전달한다.

사용 방법은 다음과 같다.

```
함수반환타입 함수명(int n, ...){
  va_list 가변인자명;
  va_start(가변인자명, 고정인자);
  va_arg(가변인자명, 가변인자값을받을자료형);//하나의 가변인자 반환
}
```

가변인자가 있는 함수의 사용 – 처리 종료(4단계)

위 과정을 다 거치고 나면, 밥을 다먹으면 다 먹었다고 말하는 듯이 가변 인자의 처리를 끝 마쳤다는 표시를 해주어야한다.

처리 종료는 var_end()함수를 이용하며, 함수의 인자로는 종료할 가변인자명을 전달한다.

사용방법은 다음과 같다.

```
함수반환타입 함수명(int n, ...){
   va_list 가변인자명;
   va_start(가변인자명, 고정인자);
   va_arg(가변인자명, 가변인자값을받을자료형);
   va_end(가변인자명); //가변인자 처리 종료
}
```

위 설명만으로는 이해가 잘 안될 수도 있으므로, 예제를 준비하였다. 다음 코드는 가변인자의 값을 출력하는 함수를 호출하는 예제이다.

예제를 보며 가변인자의 처리 과정에 대한 개념을 확립해보자.

```
#include <stdio.h>
#include <stdarg.h> //헤더파일 정의

void printall(int n, ...) {
    va_list var; //가변인자변수 선언
    va_start(var, n); //가변인자 처리 시작
```

```
    for (int i = 0; i < n; i++) { //전달받은 가변인자의 수만큼 반복
        //가변인자 값을 얻고 바로 출력
        printf("%d ", va_arg(var, int));
    }

    va_end(var); //가변인자 처리 종료
}

int main(void) {
    //5(고정매개변수) : 전달할 가변인자의 개수, 가변인자들
    printall(5,1,2,3,4,5);
}
```

```
1 2 3 4 5
```

주석을 같이 확인하면 앞서 말한 처리 과정대로 가변인자를 얻어와 가변인자의 값을 차례대로 출력하는 것을 확인할 수 있다.

2 CHAPTER 14 포인터 전달과 반환

01 함수의 포인터 반환

02 매개 변수 수정의 제한

03 함수의 구조체 전달과 반환

01 함수의 포인터 반환

지난 14-1장에서 참조에 의한 매개변수를 전달할 때 주소 연산자 (&)를 사용했었다.
즉, 전달 값이 주소이며, 참조에 의한 전달 방식이었다.

그렇다면 함수의 반환 값으로 주소값을 전달할 순 없을까?

함수의 반환 값으로 주소 값을 반환하려면, 함수의 반환타입을 포인터로 정의해 포인터 변수를 반환(return)하면 된다.

사용방법은 다음과 같다.

```
함수반환타입* 함수명(인자1...){
포인터변수선언;
return 포인터변수;
}
```

```
int* pointer(int a) {
    int* b = a;
    return b;
}
```

위 처럼 사용해도 되지만 포인터가 아닌 주소연산자를 이용해 다음과 같이 반환하는 방법도 존재한다.

```
int* pointer(int a) {
    int b = a;
    return &b;
}
```

하지만 위와 같은 방법은 함수의 변수 즉, 지역변수의 주소 값을 반환하게 된다. 지역변수는 함수가 종료되면 메모리에서 제거되기 때문에 아래 사진과 같은 경고 메세지가 뜨며, 나중에 문제가 발생할 수 있다. 그러므로 지양하는 것이 좋다.

▲ 반환하려는 값이 지역 변수임을 경고하고 있다.

02 매개 변수 수정의 제한

포인터를 매개변수로 이용하게되면 수정된 결과를 받을 수 있어 편리해 진다. 하지만 이러한 포인터를 이용한 참조에 의한 호출은 매개 변수가 가리키는 값이 다른 곳에서 맘대로 수정이 될 수 도 있다. 참조에 의한 호출은 주소값을 전달하기 때문이다. 그렇기 때문에 다른 곳에서 값이 바뀌게 되면 해당 매개변수도 영향을 받아 원치않는 값의 수정이 발생하게 될 수도 있다.

그럴 때 사용하는 것이 const 키워드 이다.

키워드 const

const 키워드는 해당 키워드가 붙여진 변수를 수정할 수 없는 상수로 만드는 키워드이다.

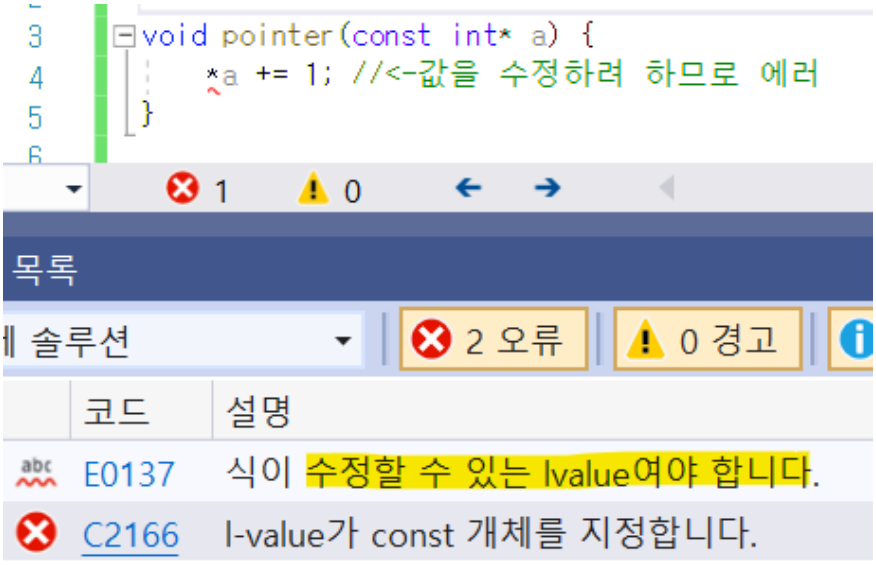
일반자료형과 더불어 포인터 변수에서도 사용이 가능하며, 사용 방법은 다음과 같다.

```
const 자료형 변수명;
자료형 const 변수명; //둘다 사용 가능하다.
```

const가 붙여진 함수는 상수이기 때문에 해당 함수 내에서도 수정되지 못한다.

그에 대한 예는 다음과 같다.

```
void pointer(const int* a) {
    *a+=1; //<-값을 수정하려 하므로 에러
}
```



▲ 위 함수를 호출 시, 수정할 수 없는 값이라는 오류 메시지를 발생시킨다.

03 함수의 구조체 전달과 반환

13장에서 배운 구조체를 함수의 매개변수와 반환 값으로 이용할 수도 있다.

다음 예제는 구조체 stc를 정의하고 값에 의한 호출을 하는 함수, 반환값이 구조체인 함수, 참조에 의한 호출을 하는 함수를 구현하고 실행시킨 예제이다.

```
#include <stdio.h>

//구조체 정의
struct stc
{
    char* name;
    char* place;
};
typedef struct stc stc;

//구조체 출력1 : 값에 의한 호출
void printstc(stc stc2) {
    printf("지금은 비어있어요 : %s %s\n",stc2.name,stc2.place);
}

//구조체에 데이터 삽입
stc insertstc(stc stc2) {
    stc2.name = "DearD";
    stc2.place = "Blog";

    return stc2; //구조체를 반환
}
```

```c
//구조체 출력2 : 참조에 의한 호출
//구조체 포인터를 매개변수로 한 출력을 위한 함수이므로
//참조에 의한 구조체 변수가 변하지 않기 위해 const 키워드 사용
void pointerstc(const stc* stc2) {
    printf("값이 들어갔어요 : %s %s\n", stc2->name, stc2->place);
}

int main(void) {
    stc stc1 = {"",""}; //구조체 변수 선언

    printstc(stc1); //빈 구조체를 출력
    stc1=insertstc(stc1); //데이터 삽입
    //구조체 포인터를 매개변수로 하므로 구조체의 주소를 전달
    pointerstc(&stc1); //데이터가 삽입된 구조체 출력
}
```

```
지금은 비어있어요 :
값이 들어갔어요 : DearD Blog
```

값에 의한 호출을 하는 함수와 참조에 의한 호출을 하는 함수의 차이점은 메모리 할당과 값의 복사에 드는 시간에 있다.

값에 의한 호출의 경우, 실인자를 형식 인자에 복사해 새로운 메모리를 할당해야하므로 만약 크기가 엄청나게 큰 구조체를 실인자로 둔다면 처리 시간이 오래 걸릴 수 있다.

하지만 참조에 의한 호출의 경우, 주소값만을 전달하기 때문에 시간이 걸리지 않는다.

3 CHAPTER 14 함수 포인터와 void포인터

01 함수 포인터

02 함수 포인터 배열

03 void포인터

01 함수 포인터 (pointer to function)

개념

지금까지 우리는 포인터를 사용하면서 변수의 저장을 주 목적으로 사용했다.
그렇다면 포인터에는 변수만 저장할 수 있는걸까?
바로 함수도 저장할 수 있다.

함수도 변수와 마찬가지로 메모리 공간 어디에 저장되어 있으며 주소를 가지고 있다.
이런 함수의 주소를 이용해 함수를 포인터에 저장해 사용한다. 즉, 함수의 주소 값을 함수 포인터 변수에 저장해 사용하는 것이다.

함수 포인터는 반환형, 인자 목록의 수, 각각의 자료형이 일치하는 함수의 주소를 저장할 수 있다.

선언 및 초기화

이러한 함수 포인터를 선언하기 위해선 저장할 함수의 반환타입과 인자들의 정보가 필요하며, 초기화를 하려면 저장할 함수의 이름을 알아야 한다.

이러한 정보를 바탕으로 선언과 초기화는 다음과 같이 진행한다.
초기화 시엔 함수포인터에 함수명을 저장할 때 함수 이름만을 써야한다. ' 함수명()'
이런식으로 소괄호를 붙이게 되면 함수의 호출로 인식하기 때문에 오류가 발생한다.

```
//선언
반환자료형 (*함수포인터변수명)(인자1,인자2...);
//초기화
함수포인터명 = 함수명;
//선언 및 초기화
반환자료형 (*함수포인터변수명)(인자1,인자2...) = 함수명;
```

호출

선언과 초기화의 과정을 거쳤다면 이젠 함수 포인터의 이름을 함수처럼 사용할 수 있게 된다. 함수를 사용하는 것과 똑같이 실행하면 된다.

사용의 예는 다음과 같다.

```
함수포인터명(인자1,인자2...);
(*함수포인터명)(인자1.인자2...);
```

예제

다음은 전달 인자의 덧셈 함수를 함수 포인터에 저장하고 호출하는 예제이다.
주석을 통해 다양한 방법들을 표시했으니 예제를 보며 개념을 확립시켜보자.

```c
#include <stdio.h>

int plus(int a, int b) {
    return a += b;
}

int main(void) {
    //선언
    int(*funpointer)(int, int);
    //int(*funpointer)(int a, int b);

    //초기화
    funpointer = plus;
    //funpointer = &plus;
```

```
    //선언 및 초기화
    //int(*funpointer)(int, int)=plus;
    //int(*funpointer)(int a, int b)=plus;
    //int(*funpointer)(int, int)=&plus;
    //int(*funpointer)(int a, int b)=&plus;

    //호출
    int sum = funpointer(10, 20);
    //int sum = (*funpointer)(10, 20);

    printf("%d", sum);

    return 0;
}
```

```
30
```

이러한 방법은 하나의 함수 포인터 변수를 선언시키고 필요에 따라 여러 함수를 포인터 변수에 저장해 사용하면 편리할 것이다.

02 함수 포인터 배열 (Array of function pointer)

개념

방금 설명한 함수 포인터에 더 나아가 함수 포인터 배열에 대해 알아보도록하자.
함수 포인터 배열은 말 그대로 함수 포인터의 개념과 배열의 개념이 합쳐진 것으로 배열을 통해 여러 개의 함수 포인터를 선언하기 위해 사용된다.

선언 및 초기화

선언 또한 함수 포인터와 동일하게 진행된다.

달라진 점은 함수 포인터에선 함수포인터명만 썼지만, 이젠 함수 포인터 배열이므로 함수포인터명 옆에 대괄호로 배열임을 명시해 주어야한다.

사용 방법은 다음과 같다.

```
//선언
반환자료형 (*함수포인터배열명[배열크기]) (인자1,...);
//초기화
함수포인터배열명[원소위치] = 함수명;
//선언 및 초기화
반환자료형 (*함수포인터배열명[배열크기])(인자1,...)={함수명1,...};
```

호출

선언과 초기화의 과정을 거쳤다면 이젠 함수 포인터 배열의 이름을 함수처럼 사용할 수 있게 된다. 함수를 사용하는 것과 똑같이 배열의 원소를 참조해 실행하면 된다.

사용의 예는 다음과 같다.

```
함수포인터배열명[원소위치](인자1,인자2...);
(*함수포인터배열명[원소위치])(인자1.인자2...);
```

예제

다음은 전달 인자의 덧셈 함수와 뺄셈 함수를 함수 포인터 배열에 저장하고 호출하는 예제이다. 주석을 통해 다양한 방법들을 표시했으니 예제를 보며 개념을 확립시켜보자.

```c
#include <stdio.h>

int plus(int a, int b) {
    return a += b;
}
int minus(int a, int b) {
    if (a > b) return a -= b;
    else  return b -= a;
}

int main(void) {
    //선언
    int(*funpointer[2])(int, int);
    //int(*funpointer[2])(int a, int b);

    //초기화
    funpointer[0] = plus; funpointer[1] = minus;
    //funpointer[0] = &plus; funpointer[1] = &minus;
```

```
    //선언 및 초기화
    //int(*funpointer[2])(int, int) = {plus,minus};
    //int(*funpointer[2])(int a, int b)={plus,minus};
    //int(*funpointer[2])(int, int)={&plus,&minus};
    //int(*funpointer[2])(int a, int b)={&plus,&minus};

    //호출
    int sum = funpointer[0](10, 20);
    int mn = funpointer[1](10, 20);
    //int sum = (*funpointer[0])(10, 20);
    //int mn = (*funpointer[1])(10, 20);
    //int sum = (*funpointer)(10, 20);

    printf("%d %d", sum,mn);

    return 0;
}
```

```
30 10
```

03 void 포인터

void포인터란 자료형이 무엇이든 간에 무시하고 주소 값만을 다루는 포인터이다. 즉 메모리가 할당되어 주소값이 있는 모든 것들은 void포인터로 다루어질수 있는 것이다.

자료형을 무시하고 다룬다는 점에서 만능 포인터라고도 불리운다. (일반 포인터, 배열, 구조체, 함수 등등 모든 주소 값을 다룬다)

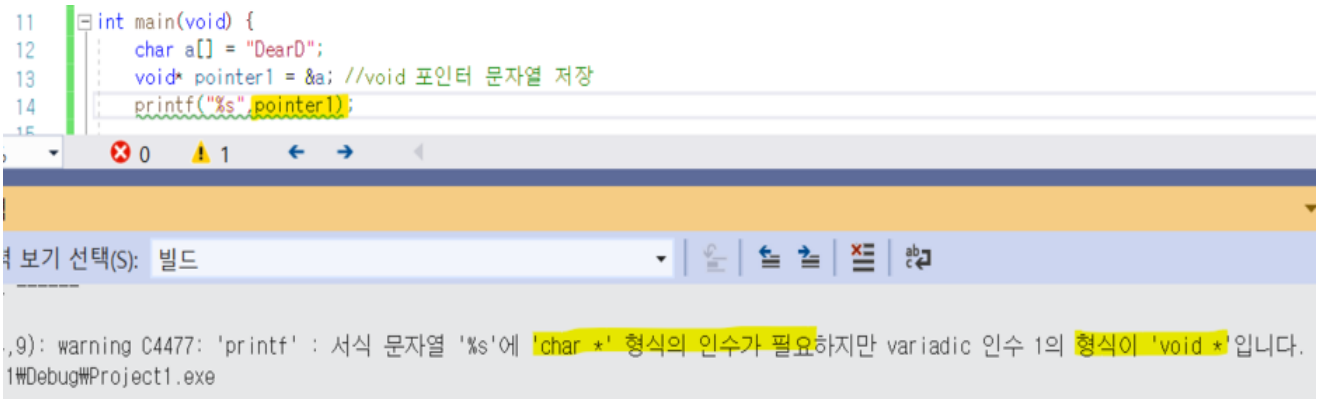
이처럼 void포인터는 모든 주소의 저장이 가능하지만 어디까지나 포인터이므로 수정이 불가능하며, 기존에 자료형이 기술되어 있지 않아 참조 범위를 알 수 없기 때문에 가리키는 변수를 참조할 수 없다.

그렇다고 참조할 수 있는 방법이 아예 없는 것도 아니다. 바로 자료형 변환을 이용하면 된다. void 포인터의 자료형 변환은 주소값이 저장되어 있기 때문에 기본적인 자료형이 아니라 포인터 자료형으로 변환을 해야 한다.

다음은 문자열을 void포인터로 사용하는 예제이다.

```c
#include <stdio.h>

int main(void) {
    char a[] = "DearD";
    void* pointer1 = &a; //void 포인터 문자열 저장
    printf("%s",(char*)pointer1); //참조 시 자료형 변환 필요

    return 0;
}
```

만약 자료형 변환을 안하게 된다면 다음과 같은 경고메세지가 뜬다.



▲ 자료형 변환을 안하면, 형식이 잘못되었다는 경고 메시지가 발생한다.

이번엔 void포인터 형식의 함수 포인터에 대한 내용이다.
만약, 함수 포인터를 사용하게 되더라도 앞 예제와 같은 방식으로 형변환을 하면 된다.
더 나아가 함수 포인터의 형변환의 경우 코드를 길게 써야 하는 상황이 생길 수도
있으므로 typedef 를 이용해 자료형을 정의해 사용하면 더욱 편리하다.

그에 대한 예제는 다음과 같다.

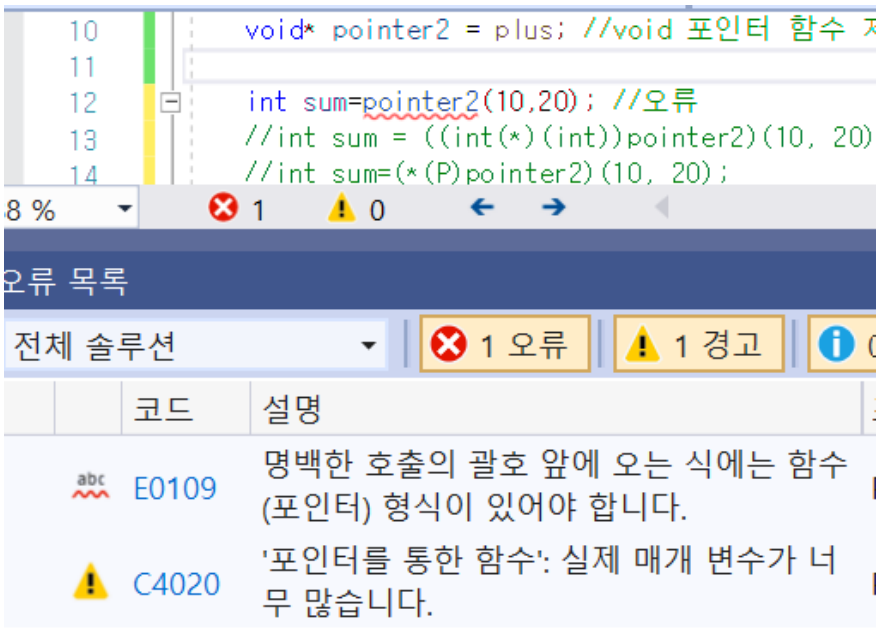
```
#include <stdio.h>

int plus(int a, int b) {
    return a += b;
}
typedef int(*P)(int,int); //typedef로 자료형 변환을 짧게 정의

int main(void) {
    void* pointer2 = plus; //void 포인터 함수 저장

//int sum=pointer2(10,20); //오류
int sum = ((int(*)(int,int))pointer2)(10, 20); //자료형변환 후 참조
//int sum=(*(P)pointer2)(10, 20);
```

```
    printf("%d", sum);

    return 0;
}
```

```
30
```

오류

만약 함수 포인터처럼 호출하게 된다면 다음과 같은 오류가 발생한다.



▲ 식 앞에 함수 형식이 있어야 한다는 오류 메시지가 발생한다.

void 포인터의 활용

void 포인터 형식의 함수 포인터는 보통 함수 포인터라는 것을 다른 개발자에게 숨길 때도 사용한다고 한다.

그에 대한 관점은 두가지 이다.
첫번째, 다른 개발자들을 돕기 위해서이다. 함수 포인터 기술을 사용하면 프로그램의 확장성이 좋아지는 장점이 있다. 하지만, 코드 내 함수 포인터의 의미를 몰라서 혼란스러워하는 경우가 있기 때문에 이러한 혼란을 최소화 하고자 void 포인터 형식의 함수 포인터로 대체한다.

두번째는 다른 개발자들에게 자신의 코드를 숨길 때이다. 앞 이유과 같은 숨긴다는 맥락으로 이어지지만, 자신의 서비스 구조나 기술 구조를 노출하기 싫을 때도 void 포인터 형식의 함수 포인터를 많이 사용한다. 다른 이에게 노출하고 싶지 않은 코드를 모두 함수로 만들고 해당 함수를 void 포인터 형식의 함수 포인터에 저장한 다음 필요할 때만 자신이 아는 형 변환을 사용해 함수를 호출하는 방식으로 기술 구조를 감춘다.

CHAPTER

15

파일 처리

15.1 파일 기초

15.2 텍스트 파일 입출력

15.3 이진 파일 입출력

1 CHAPTER 15 파일 기초

01 파일

02 파일 스트림

03 입출력 스트림

04 파일 스트림 열기 / 닫기

05 프로그램 강제 종료 함수

01 파일

개념

파일은 일상생활에서 자주 이용되는 파일의 개념과 같이 특정 자료의 집합체이다. 또한, 보조기억장치에 저장되며, 보조기억장치의 정보저장단위이다.

파일을 사용하는 이유는 내용에 대한 영구적인 보존에 있다. 변수와 같이 프로그램 내부에서 할당되어 사용되는 주기억장치(RAM)의 메모리 공간은 프로그램 실행 시 공간이 할당되었다가 프로그램이 종료되면 사라진다.

하지만 파일의 개념이 존재하면, 보조기억장치인 디스크(HardDisk)에 저장된다. 이 파일은 직접 삭제하지 않는 한 프로그램이 종료되더라도 계속 저장하고 사용될 수 있다.

이처럼 프로그램에서 사용하던 정보를 프로그램 종료 후에도 영구적으로 사용하고 싶다면 파일에 해당 내용을 저장해야 한다.

파일은 텍스트 파일(text file)과 이진 파일(binary file) 두가지 유형으로 나뉘며, 해당 내용은 다음에 나온다.

텍스트 파일

텍스트 파일은 문자 기반의 파일이며 대표적인 확장자로는 txt 등이 있다. (우리가 쓰는 Code도 텍스트 파일에 속한다.) 내용은 아스키 코드로 인코딩되며, 텍스트 편집기를 통하여 텍스트 파일의 내용을 볼 수 있고 수정할 수 있다.

이진 파일

이진파일은 그림, 동영상, 실행파일과 같이 각각의 목적에 알맞은 자료가 이진형태로 저장되는 파일이며, 대표적인 확장자로는 html,exe 등이 있다.

이진파일 자료는 컴퓨터 내부 형식으로 저장되는데, 메모리 자료 내용에서 어떤 변환도 거치지않고 그대로 파일에 기록되기 때문에 입출력 속도가 텍스트 파일보다 빠르다는 장점이 존재한다.

하지만 텍스트 파일과는 다르게 텍스트 편집기에서의 내용을 확인할 수 없고
이진파일과 관련된 특정 프로그램을 이용해야만 내용을 확인할 수 있다.

02 파일 스트림 (File Stream)

개념

파일 스트림이란 보조기억장치의 파일과 프로그램을 연결하는 전송 경로이다.

프로그램에서 보조기억장치에 파일로 정보를 저장하거나 파일에서 정보를 참조하기 위해선 파일에 대한 파일 스트림을 연결해야한다.

파일 스트림을 만들기 위해선 특정한 파일명과 파일 모드가 필요하다.

파일 모드

파일모드 란 파일 스트림의 특성을 뜻하는 말이며 텍스트 파일의 파일모드와 이진 파일의 파일 모드가 나뉘어 있다, 그에 대한 종류는 다음 장의 표에 기입해 두었다.

	모드	설명	
텍스트 모드	r	읽기	파일을 읽을 수 있게만 호출 (쓰기x) 파일이 없거나 찾을 수 없는 경우: 파일 호출 실패
	w	쓰기	파일을 쓸 수 있게만 호출 (읽기x) 지정한 파일이 있는 경우: 파일 내용을 모두 지우고, 새로운 파일 생성 지정한 파일이 없는 경우: 새로운 파일 생성
	a	추가	파일을 추가로 쓸 수 있게만 호출 지정한 파일이 있는 경우: 파일의 끝에서부터 내용을 추가 (중간에 쓸 수 없음) 지정한 파일이 없는 경우: 새로운 파일 생성
	r+	읽고 쓰기	지정한 파일이 있는 경우: 기존의 내용을 덮어씁니다. 파일이 없거나 찾을 수 없는 경우: 파일 호출 실패
	w+	읽고 쓰기	지정한 파일이 있는 경우: 파일 내용을 모두 지우고, 새로운 파일 생성 지정한 파일이 없는 경우: 새로운 파일 생성
	a+	읽고 추가	지정한 파일이 있는 경우: 파일의 끝에서부터 내용을 추가 (중간에 쓸 수 없음) 지정한 파일이 없는 경우: : 새로운 파일 생성
	wx	쓰기	지정한 파일이 있는 경우: 파일 호출 실패 지정한 파일이 없는 경우: 새로운 파일 생성
	w+x	읽고 쓰기	지정한 파일이 있는 경우: 파일 호출 실패 지정한 파일이 없는 경우: 새로운 파일 생성

▲ 텍스트 파일의 파일 모드

	모드	설명	
이진 모드	rb	읽기	파일을 읽을 수 있게만 호출 (쓰기x) 파일이 없거나 찾을 수 없는 경우: 파일 호출 실패
	wb	쓰기	파일을 쓸 수 있게만 호출 (읽기x) 지정한 파일이 있는 경우: 파일 내용을 모두 지우고, 새로운 파일 생성 지정한 파일이 없는 경우: 새로운 파일 생성
	ab	추가	파일을 추가로 쓸 수 있게만 호출 지정한 파일이 있는 경우: 파일의 끝에서부터 내용을 추가 (중간에 쓸 수 없음) 지정한 파일이 없는 경우: 새로운 파일 생성
	rb+ / r+b	읽고 쓰기	지정한 파일이 있는 경우: 기존의 내용을 덮어씁니다. 파일이 없거나 찾을 수 없는 경우: 파일 호출 실패
	wb+ / w+b	읽고 쓰기	지정한 파일이 있는 경우: 파일 내용을 모두 지우고, 새로운 파일 생성 지정한 파일이 없는 경우: 새로운 파일 생성
	ab+ / a+b	읽고 추가	지정한 파일이 있는 경우: 파일의 끝에서부터 내용을 추가 (중간에 쓸 수 없음) 지정한 파일이 없는 경우: : 새로운 파일 생성
	wbx	쓰기	지정한 파일이 있는 경우: 파일 호출 실패 지정한 파일이 없는 경우: 새로운 파일 생성
	wb+x / w+bx	읽고 쓰기	지정한 파일이 있는 경우: 파일 호출 실패 지정한 파일이 없는 경우: 새로운 파일 생성

▲ 이진 파일의 파일 모드

TIP

+가 붙는 파일 모드는 버퍼에서 읽기/쓰기를 모두 관여하게 되므로 문제 발생할 수 있다!

03 입출력 스트림 (io stream)

자료의 입출력은 자료의 이동의 과정이고, 자료의 이동이 가능하려면 이동할 경로가 필요하다. 여기서 이동할 경로를 제공해 주는 것이 입출력스트림이다.

입력 스트림 (input stream)

입력 스트림(input stream)이란 자료 원천부에서 프로그램으로 자료의 입력을 위한 스트림(경로)이다.

이때 자료 원천부(data source)는 자료가 떠나는 시작점으로 키보드에 의한 표준 입력이 될 수 도 있고, 파일, 스크린의 터치, 네트워크 입력 등이 될 수 있다.

출력 스트림 (output stream)

출력 스트림(output stream)이란 프로그램에서 자료 목적부로 자료의 출력을 위한 스트림(경로)이다.

이때 자료 목적부(data destination)는 자료의 도착 지점으로 콘솔에 의한 표준 출력, 파일, 프린트의 출력물, 네트워크 출력 등이 될 수 있다.

04 파일 스트림 열기 / 닫기

fopen() / fopen_s()

fopen()은 프로그램에서 특정한 파일과 파일 스트림을 연결하는 함수이다.
스트림 연결 성공 시 구조체 파일 포인터를, 실패 시 NULL을 반환한다.

사용 방법은 다음과 같다.

```
fopen(파일명,파일모드);
```

파일 스트림 열기를 실패할 경우를 대비해 아래와 같이 파일 스트림 연결에 성공했는지 확인하는 if문을 요한다. fopen()함수로 연 파일을 파일 포인터에 저장하는데 fopen()함수는 실패 시 NULL을 반환 하므로 아래의 조건의 실행문으로 연결 실패에대한 코드를 작성해주면 된다.

```
if((파일포인터변수=fopen(파일이름,파일모드))==NULl)
```

fopen()함수는 전처리기를 정의하지않으면 오류가 발생한다.

전처리기를 정의하고 싶지 않은 경우, fopen_s()를 첫번째 인자로 파일을 저장할 파일 포인터 주소를 전달하여 사용한다. 그 뒤는 fopen()과 마찬가지로 파일명과 파일모드를 인자로 전달해준다.

fopen()과의 차이점은 스트림 연결 성공 시 정수 0을, 실패 시 양수를 반환한다.

fclose()

flose는 fopen()으로 연결한 파일 스트림을 닫는 함수로, 파일스트림에 할당된 자원을 반납하고 파일과 메모리 사이에 있던 버퍼의 내용을 모두 지우는 역할을 수행한다.
파일 스트림의 사용이 끝난 경우 꼭 닫아주어야하기때문에 파일 스트림 연결 시 꼭 필요한 함수이다.

스트림 닫기 성공 시 0을, 실패시 EOF(End Of File)를 반환하며, 파일 포인터를 인자로 전달하여 다음과 같이 사용한다.

```
fclose(파일포인터변수);
```

05 프로그램 강제 종료 함수

exit(1)

exit()함수는 함수를 강제로 종료하는 함수이다.
사용 시, 헤더파일 stdib.h를 정의해야하며, 인자로 전달하게 되는 값 중 0은 정상종료, 1은 정상종료가 아님을 알려준다. 파일 스트림 열기가 실패할 경우, 이 함수를 실패할 조건의 실행문으로 작성해 프로그램을 강제로 종료 하자.

2 CHAPTER 15 텍스트 파일 입출력

01 입력 함수

02 출력 함수

03 검사 함수

01 입력 함수

fscanf() / fscan_s()

fscanf()는 표준 입력 함수로, fopen()함수를 통해 텍스트 파일을 열면, 해당 텍스트 파일을 입력(읽기)하는 함수이다.

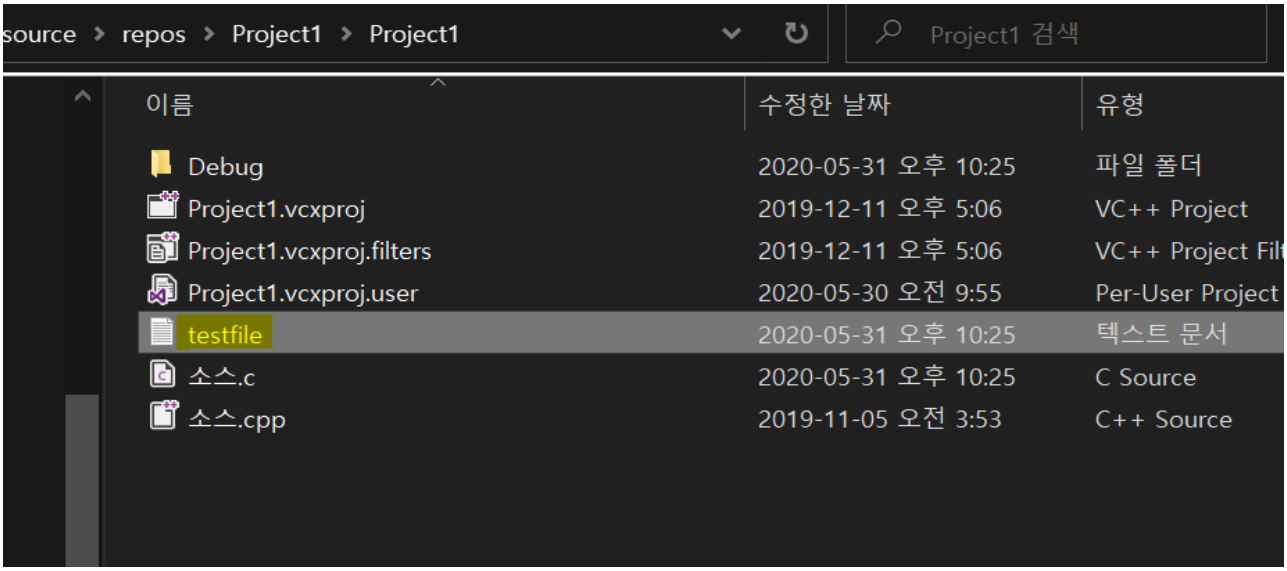
첫번째 인자의 파일 포인터의 내용을 두번째 인자의 형식 제어문자를 바탕으로 세번째 인자에 저장한다. 만약 첫번째 인자로 stdin을 넣는다면 단순히 scanf()와 scanf_s()의 기능을 수행한다.

fscanf()함수를 이용하고 나면 해당 텍스트파일의 내용을 확인 할 수 있다.

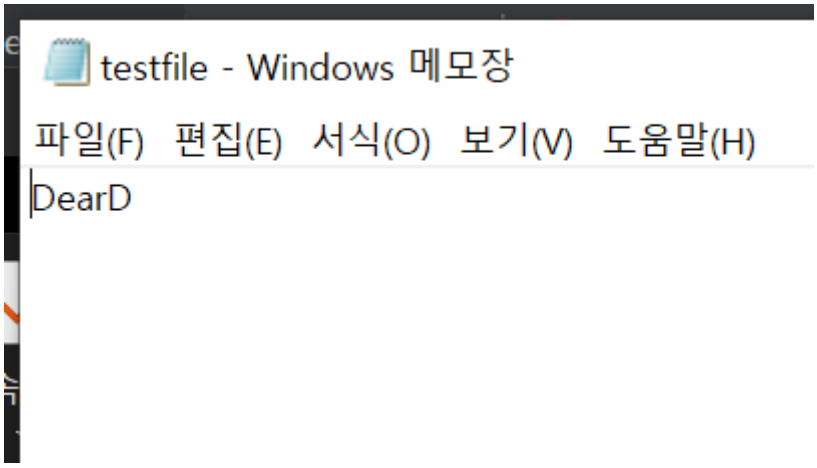
사용 방법은 다음과 같다.

```
fscanf(파일포인터, 제어문자열, 변수(상수)1, 변수(상수)2...);
```

그에 대한 예는 다음과 같다. 다음의 절차를 확인하며 개념을 확립해보자.



▲ 현재 텍스트 파일 testfile.txt 가 생성되어 있다.



▲ 텍스트파일에 DearD가 써져있다.

자료를 입력하기 위해 파일모드 r를 쓸 것이기 때문에 텍스트 파일은 생성돼 있어야 한다. 생성돼 있지 않으면 오류가 난다. (필자는 DearD라는 내용이 있는 testfile.txt을 미리 생성하였다.)

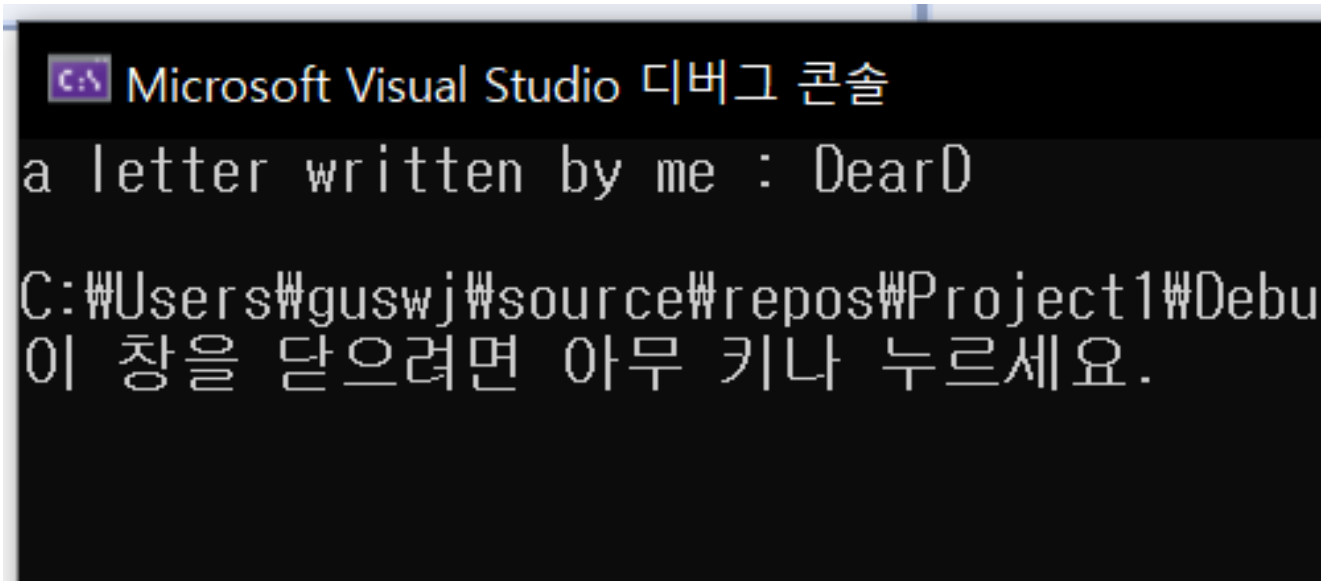
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main() {
    char str[101]; //입력할 문자를 저장할 변수
    FILE* f; //파일 포인터

    //파일을 열어 입력 스트림을 통해 파일의 내용을 읽어옴
    if ((f = fopen("testfile.txt", "r")) == NULL) {
        printf("not open");
        exit(1);
    }
    fscanf(f, "%s", str);
    fclose(f);
```

```
    printf("a letter written by me : %s \n", str);

    return 0;
}
```

읽어온 문자를 저장할 변수 str와 읽어온 파일을 저장할 파일 포인터 변수 f를 선언하였다.

파일 모드 r을 통해 읽기 모드로 파일이 무사히 열린다면 fscanf()함수를 이용해 파일의 내용을 문자열의 형태로 읽어와 str에 저장한다.

저장한 내용은 printf()함수를 통해 확인할 수 있다.



▲ 결과 화면 (open한 텍스트 파일의 내용으로 DearD가 저장되어 출력 됨)

마지막 printf()함수로 저장된 내용이 DearD였음을 확인하며,, 앞 텍스트 파일에 저장된 내용과 같음을 알 수 있다.

단, fsacnf()함수는 전처리기를 정의해주어야 하며 원치않는 경우 fscanf_s()함수를 사용해야 한다. 전달인자는 같다.

fgetc() / getc()

fgetc() 와 getc() 함수는 파일로부터 문자 하나를 입력받는 함수이다.

문자 하나의 입력 대상인 파일포인터를 인자로 이용한다.

사용 방법은 다음과 같다.

```
fgetc(파일포인터)
getc(파일포인터)
```

fgets()

fgets()는 파일로부터 한행의 문자열을 입력받는 함수이다.

세번째 인자인 파일포인터로부터 문자열을 개행 문자(₩n)까지 두번째 인자의 수만큼 읽어와 마지막 개행 문자를 null로 바꾸어 첫번째 인자인 입력버퍼문자열에 저장하게 된다.

사용 방법은 다음과 같다.

```
fgets(문자열포인터, 읽어올문자최대수, 파일포인터)
```

02 출력 함수

fprint()

fprint()는 표준 출력 함수로, fopen()함수를 통해 텍스트 파일을 열면, 해당 텍스트 파일에 자료를 출력(쓰기) 함수이다.

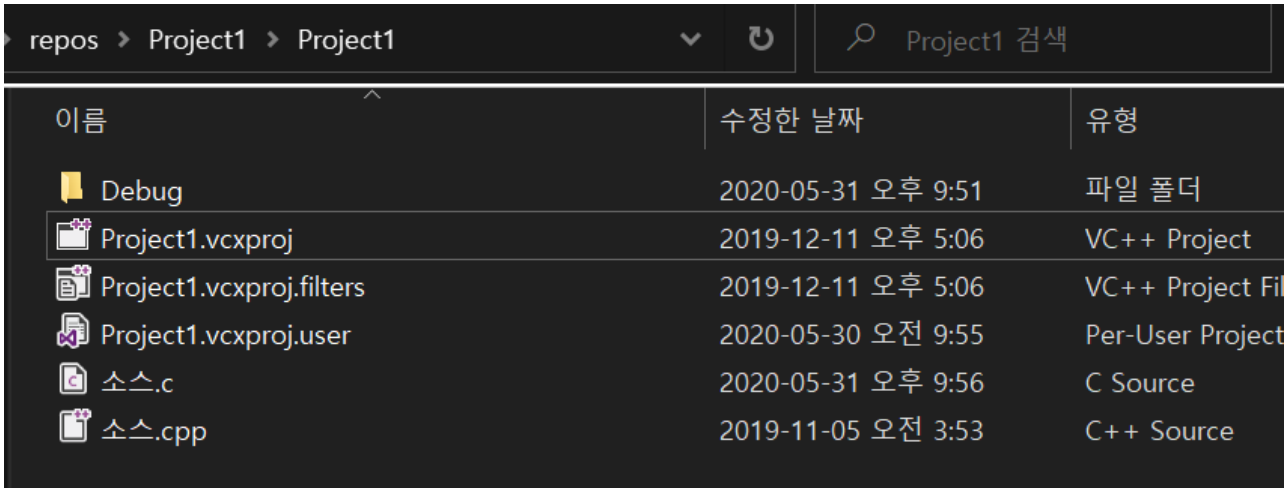
두번째 인자의 형식 제어문자를 바탕으로 세번째 인자의 값을 첫번째 인자의 자료를 쓸 파일 포인터에 저장한다. 만약 첫번째 인자로 stdout을 넣는다면 단순히 printf()의 기능을 수행한다.

fprint()함수를 이용하고 나면 해당 텍스트파일에 세번째 인자 값이 저장돼 있는 것을 확인할 수 있다.

사용 방법은 다음과 같다.

```
fprintf(파일포인터, 제어문자열, 변수(상수)1, 변수(상수)2...);
```

그에 대한 예는 다음과 같다. 다음의 절차를 확인하며 개념을 확립해보자.



▲ 현재 어떠한 텍스트 파일도 생성되어 있지 않다.

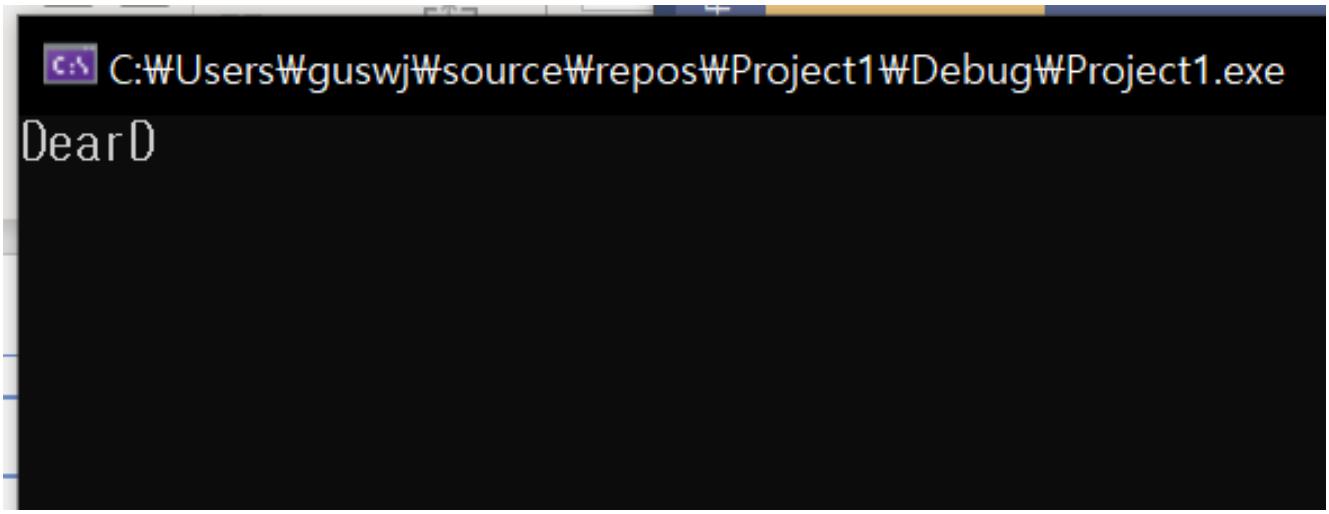
자료를 출력하기 위해 파일모드 w를 쓸 것이기 때문에 텍스트 파일은 생성돼 있어도 되지만, 굳이 생성돼 있지 않아도 상관 없다. (필자는 생성하지 않았다.)

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

int main() {
    char str[101]; //입력할 문자를 저장할 변수
    FILE* f; //파일 포인터

    //변수의 값을 파일에 저장함
    if ((f = fopen("test_file.txt", "w")) == NULL) {
    printf("not open");
    exit(1);
    }
    scanf("%s", str); //사용자에게 문자 입력 받기
    fprintf(f, "%s", str); //쓰기
    fclose(f);

    printf("a letter written by me : %s \n", str);

    return 0;
}
```

입력할 문자를 저장할 문자열 str과 파일을 저장할 파일 포인터 f를 선언해 주었다.

조건문을 통해 파일이 열리지 않을 경우엔 강제 종료를 하도록 하였고, 그렇지 않다면 표준 입력 함수인 scanf()함수를 사용해 사용자에게 문자열에 대한 입력을 받은 뒤 저장한다. 이렇게 값이 담긴 str을 fprint()함수를 호출해 파일에 출력한다.

마지막 printf()함수는 사용자가 무엇을 입력했는지 보여주기 위해 써 놓았으나 없어도 되는 코드이다.

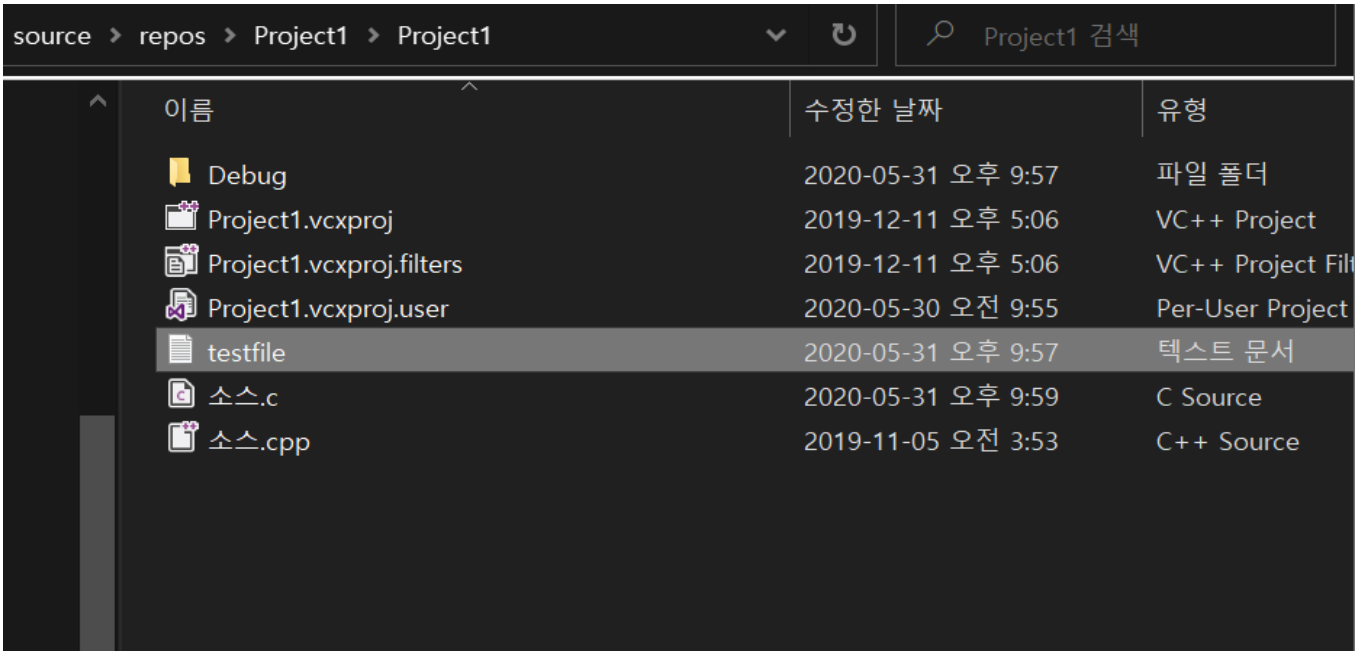


▲ DearD 입력

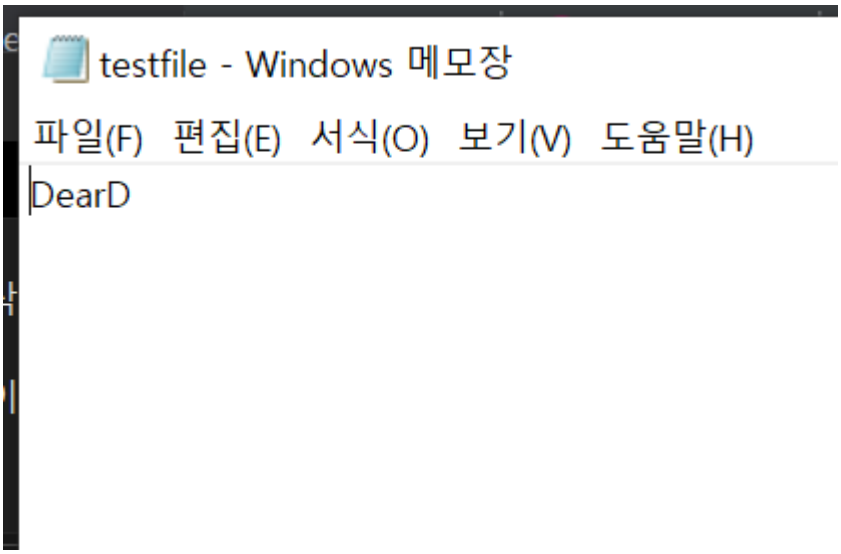
▲ Enter 수행 종료

위와 같이 사용자에게 문자를 입력 받고 위 코드를 수행한다.

이젠 예상한대로 잘 수행이 되었을지 확인을 해보겠다.



▲ 텍스트 파일이 생성된 것을 확인할 수 있다.



▲ 메모장 확인 결과 방금 쓴 내용이 들어갔음을 확인할 수 있다.

이렇게 파일모드 w와 fprint()를 이용해 무사히 텍스트 파일이 생성되었음을 알 수 있다.

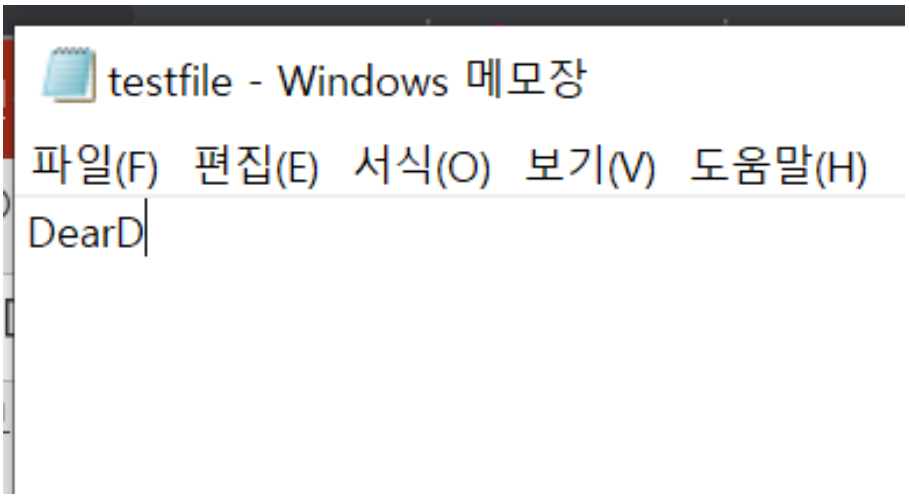
fprint() 와 fscanf()를 이용해 입출력 동시에 하기

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

int main() {
    char str[101]; //입력할 문자를 저장할 변수
    FILE* f; //파일 포인터

    f = fopen("testfile.txt", "r+");
    fscanf(f, "%s", str);
    printf("read : %s \n", str);

    rewind(f); //파일 포인터 시작 위치 변경
    fprintf(f, "%s", "apple");

    fclose(f);

    return 0;
}
```

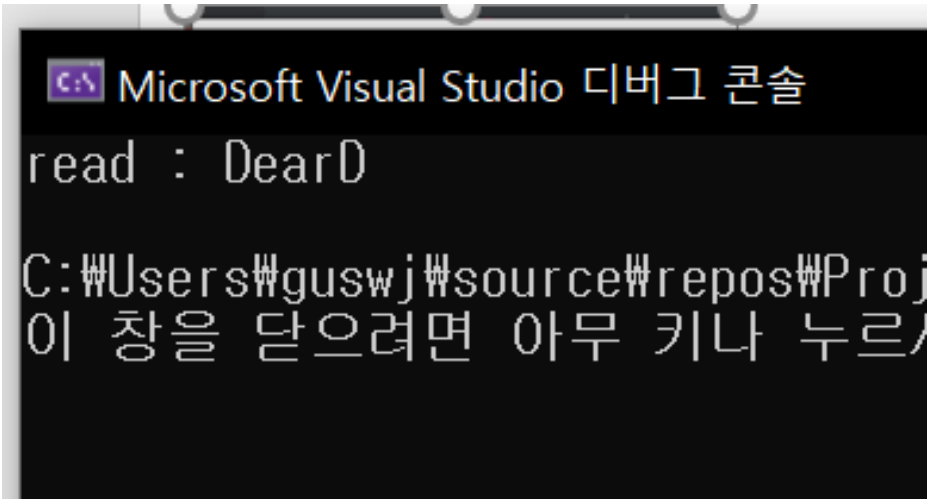
위 코드를 입력하면 입출력을 동시에 할 수 있다.

파일모드를 r+로 바꾸면 먼저 읽고 그다음에 쓰는 기능을 할 수 있게 된다.
다른 내용은 앞서 설명한 것과 같지만, rewind()라는 함수가 추가 되었다.

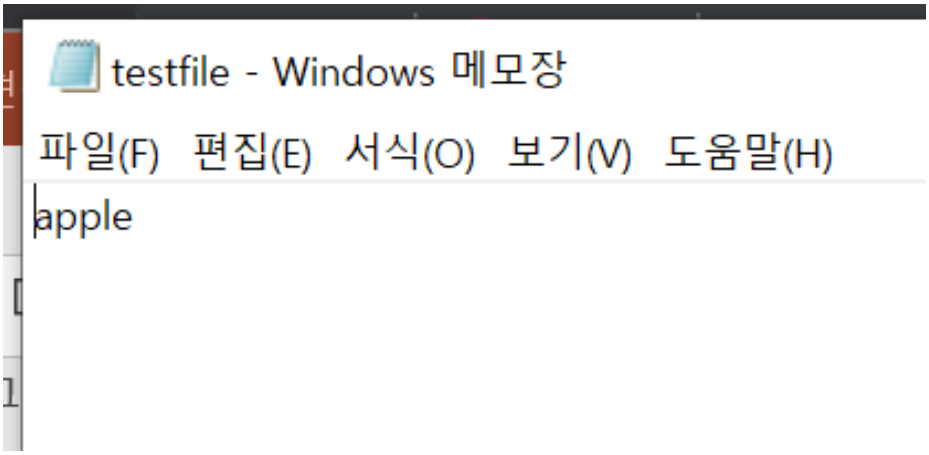
rewind()함수란 파일포인터의 위치를 파일 시작점으로 옮기는 함수이다. 데이터를 새로 쓸 때 마지막 위치가 아닌 처음 위치부터 하려고 했기 때문에 사용하였다.



▲ 기존 텍스트 파일의 내용



▲ 실행 결과 (기존 텍스트파일의 내용과 일치함을 확인함)



▲ 메모장을 다시 확인 결과 방금 쓴 내용이 들어갔음을 확인할 수 있다.

이렇게 파일모드 r+을 통해 기존 내용을 읽어오고 기존 내용을 바꾸는 처리를 동시에 할 수도 있다.

fputc() / putc()

fputc() 와 putc() 함수는 문자 하나를 파일로 출력(저장)하는 함수이다.

출력할 문자를 첫번째 인자로, 문자 하나의 출력 대상인 파일포인터를 두번째 인자로 이용한다.

사용 방법은 다음과 같다.

```
fputc(출력할문자, 파일포인터)
putc(출력할문자, 파일포인터)
```

fputs()

fputs()는 한 행의 문자열을 파일에 출력하는 함수이다.

세번째 인자인 파일포인터로부터 첫번째 문자열포인터의 마지막 null을 개행문자(₩n)로 바꾸어 두번째 인자인 파일포인터에 출력하게 된다.

사용 방법은 다음과 같다.

```
fputs(문자열포인터, 파일포인터)
```

03 검사 함수

feof()

feof()는 파일 스트림의 EOF(end of file)를 표시를 검사하는 함수이다.

이전 읽기 작업에서 파일이 EOF이면, 즉 파일의 끝이면 0이 아닌값, 파일이 끝이 아니면 0(true)을 반환한다.

사용 방법은 다음과 같다.

```
feof(파일포인터)
```

ferror()

ferror()는 이전 파일 처리에서 오류가 발생했는지 검사하는 함수이다.

오류가 발생하면 0이 아닌값, 오류가 발생하지 않으면 0을 반환한다.

사용 방법은 다음과 같다.

```
ferror(파일포인터)
```

3 CHAPTER 15 이진 파일 입출력

01 입력 함수

02 출력 함수

이진 파일은 15-2장에서 설명했던 함수가 텍스트파일처리와 함께 쓰이지만 내부 요소를 모두 유지하며 byte 단위의 저장이 되는 파일로 유지를 위해 파일 입출력 시 블록 단위의 처리를 해야하기 때문에 그에 대한 함수가 따로 구성되어 있다.

해당 함수에 대한 내용은 다음과 같다.

01 입력 함수

fread()

fread()는 이진 파일의 내용을 읽기(로드)위한 함수이다.

네번째인자로부터 두번째인자크기만큼 세번째 인자의 개수만큼 읽어와 최종적으로 첫번째인자에 읽어들인다.

반환 값은 읽기에 성공한 항목의 개수이며, 사용 방법은 다음과 같다.

```
fread(자료의주소값,자료항목의바이트크기,자료항목의개수,파일포인터)
```

02 출력 함수

fwrite()

fwrite()는 파일에 내용을 쓰기(write)위한 함수이다.

첫번째 인자가 가리키는 메모리에서 두번째 인자만큼 세번째인자 개수를 네번째 인자에 저장한다.

반환 값은 쓰기에 성공한 항목의 개수이며, 사용 방법은 다음과 같다.

```
fwrite(자료의주소값,자료항목의바이트 크기,자료항목의개수,파일포인터)
```

후기

포트폴리오 제작을 무사히 끝마쳤습니다.

이번 포트폴리오 제작은 포인터와 문자 관련 함수에 대한 저의 취약점을 보완할 수 있는 계기가 되었고, 제가 이해하기 쉽게 직접 코드를 다시 짠 과정이 큰 도움이 되었습니다. 제가 알지 못했던 부분에 대한 새로운 발견과 습득은 항상 즐겁고 짜릿합니다. 그 기분을 느낄 수 있게 되어 포트폴리오 제작도 과제에 국한되는 것이 아닌 본질적으로 나에게 도움이 된다는 느낌을 강하게 받았기에 더 열심히 임할 수 있었습니다.

현재 이러한 설명 방식으로 공부 블로그를 운영 중에 있지만 문서로써 제작하는 것은 사뭇 다른 느낌을 받았습니다. 정말 책과 같이 제작하려고 하니 보완해야할 점이 계속 눈에 보였기 때문입니다. 이러한 보완해야할 점을 수정해 나가는 과정에서 더 성장하는 느낌을 받았습니다.

이러한 포트폴리오 제작의 기회를 주신 동양미래대학 강환수 교수님께 감사의 말씀을 전해드리고 마치겠습니다.

감사합니다.

동양미래대학 컴퓨터정보공학과
신현정

Man is not the sum of what he has already,
사람은 이미 가지고 있는 것들에 대한 합이 아니라,

but rather the sum of what he does not yet have, of what he could have.
아직 갖지 않았지만 가질 수 있는 것들에 대한 합이다.

_ *Jean-Paul Sartre*