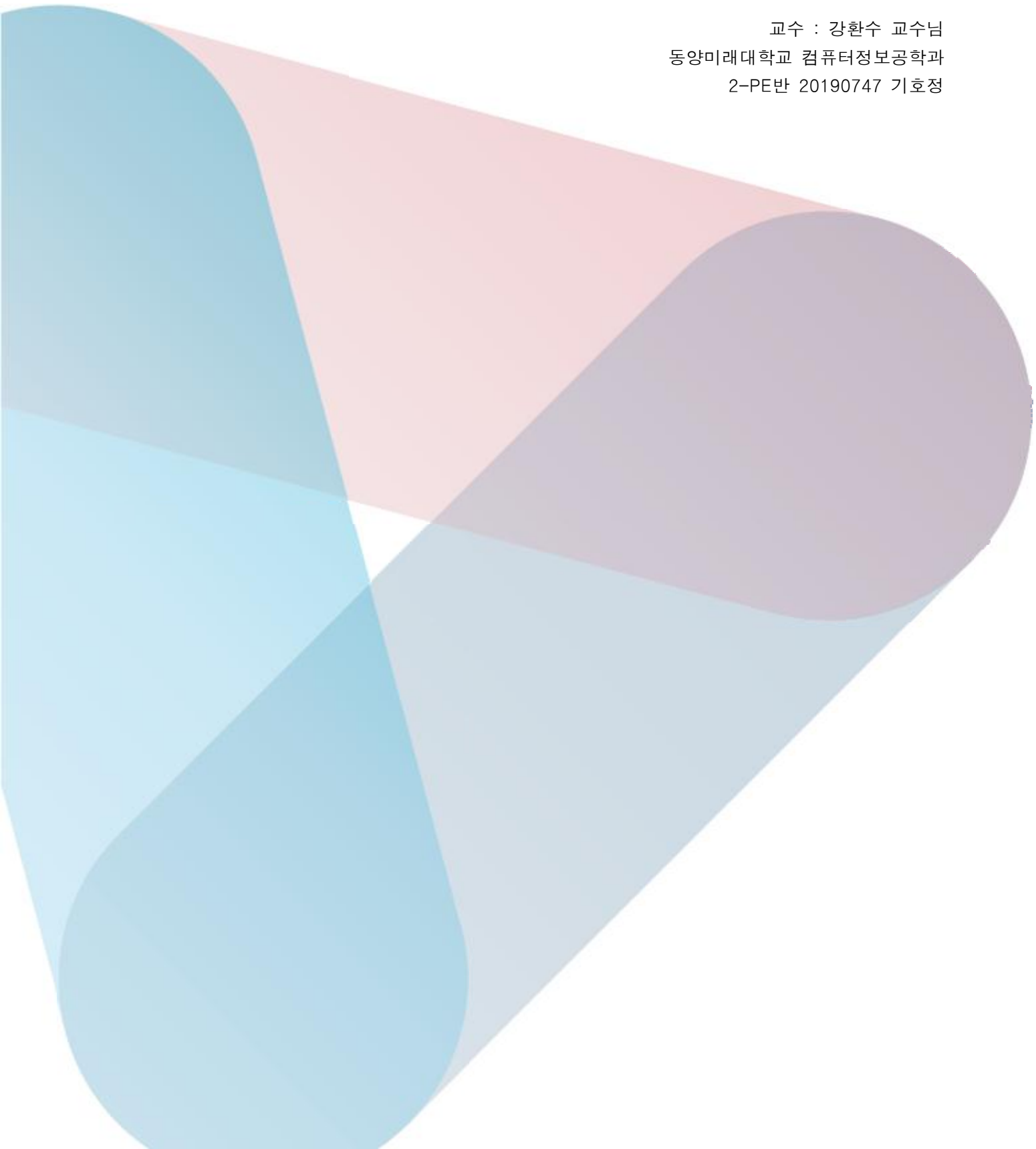
C Application

교수 : 강환수 교수님
동양미래대학교 컴퓨터정보공학과
2-PE반 20190747 기호정



강의계획서

Part 11

11.1 스트림과 데이터의 이동

11.2 문자와 문자열

11.3 여러 문자열 처리

Part 12

12.1 전역변수와 지역변수

12.2 정적 변수와 레지스터 변수

12.3 메모리 영역과 변수 이용

Part 13

13.1 구조체

13.2 자료형 재정의

13.3 구조체와 공용체의 포인터와 배열

Part 14

14.1 함수의 인자전달 방식

14.2 포인터 전달과 반환

14.3 함수 포인터와 void 포인터

Part 15

15.1 파일 기초

15.2 텍스트 파일 입출력

15.3 이진 파일 입출력

11.1 스트림과 데이터의 이동

- 무엇이 '입력'이고 무엇이 '출력'인가?

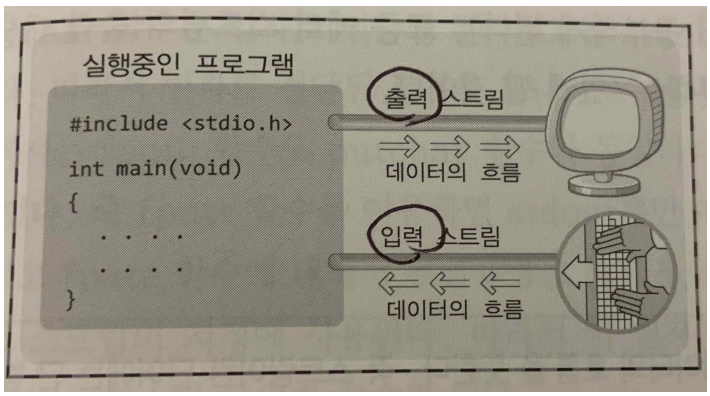
문자와 문자열에 대해 배우기 앞서서 스트림에 대해 다뤄보자. 데이터의 입력과 출력은 프로그램의 흐름을 뜻하는 것이다. 그렇다면 무엇이 '입력'이고, 무엇이 '출력'일까? 프로그램을 중심으로 프로그램 안으로 데이터가 들어오는 것이 입력이고, 프로그램 밖으로 데이터가 흘러나가는 것이 출력이다.

가장 대표적인 입력장치로는 키보드가 있으며, 파일도 입력의 대상이 될 수 있다. 그리고 가장 대표적인 출력 장치로는 모니터가 있으며, 파일 역시 출력의 대상이 될 수 있다. 이 외에도 컴퓨터에 연결이 가능한 마우스, 프린터, 화상 카메라와 같은 기타 장치들도 입출력 장치에 해당한다. 이렇듯 입출력의 대상은 매우 넓고, 또 그만큼 입출력에는 포괄적인 의미가 담겨있다.

- 데이터의 이동수단이 되는 스트림

> "프로그램상에서 모니터로 문자열을 출력할 수 있는 이유는 무엇인가요?"
>
> "문자열을 출력하는 printf 함수가 존재하기 때문입니다."

위의 문제에 대해 아래의 답변을 한 사람들이 많을 것이다. 하지만 여기서 묻고자 한 것은 'printf 함수호출 시 어떠한 경로 및 과정을 거쳐서 문자열이 출력되는가'이다. 우리가 구현하는 프로그램과 모니터, 그리고 프로그램과 키보드는 기본적으로 연결되어있는 개체가 아닌, 서로 떨어져 있는 개체이다. 따라서 프로그램상에서 모니터와 키보드를 대상으로 입출력하기 위해서는 이들을 연결시켜 주는 다리가 필요하다. 그리고 이러한 다리의 역할을 하는 매개체를 가리켜 '스트림(stream)'이라고 한다.



위의 그림을 보면, 실행 중인 프로그램과 모니터를 연결해주는 '출력 스트림'이라는 다리가 놓여있고, 실행 중인 프로그램과 키보드를 연결해주는 '입력 스트림'이라는 다리가 놓여있음을 알 수 있다. printf 함수와 scanf 함수를 통해서 데이터를 입출력할 수 있는 근본적인 이유는 바로 이 다리에 있다.

그렇다면 다리의 역할을 하는 스트림의 정체는 무엇일까? 이는 운영체제에서 제공하는 소프트웨어적인 (소프트웨어로 구현되어 있는) 가상의 다리이다. 다시 말해서, 운영체제는 외부 장치와 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. 지금껏 printf 함수와 scanf 함수를 호출해 오면서 스트림의 생성과 관련된 코드를 본 적이 있는가? 사실 우리는 스트림의 생성을 어떻게 요구해야 하는지조차 알지 못한다. 다시 말해서, 콘솔 입출력을 위한 스트림은 자동으로 생성된다.

정리하면, 콘솔 입출력을 위한 '입력 스트림'과 '출력 스트림'은 프로그램이 실행되면 자동으로 생성되고, 프로그램이 종료되면 자동으로 소멸되는 스트림이다. 즉, 이 둘은 기본적으로 제공되는 '표준 스트림(standard stream)'이다. 그리고 표준 스트림에는 '에러 스트림'도 존재하며 이들 각각에는 다음과 같이 stdin, stdout, stderr라는 이름이 붙어 있다.

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

11.2 문자와 문자열

- 문자와 문자열

프로그램에서 다루는 자료는 대부분이 수 또는 문자와 문자열이다. 문자는 영어의 알파벳이나 한글의 한 글자를 작은따옴표로 둘러싸서 'A'와 같이 표기하며, C 언어에서 저장공간 크기 1바이트인 자료형 char로 지원한다. 작은따옴표에 의해 표기된 문자를 문자 상수라 한다.

```
char ch = 'A' ;
```

문자의 모임인 일련의 문자를 문자열(string)이라 한다. 이러한 문자열은 일련의 문자 앞뒤로 큰따옴표로 둘러싸서 "java"로 표기한다. 이와 같이 큰따옴표에 의해 표기된 문자열을 문자열 상수라 한다. "A"처럼 문자 하나도 큰따옴표로 둘러싸면 문자열 상수이다. 그러나 문자의 나열인 문자열은 'ABC'처럼 작은따옴표로 둘러싸도 문자가 될 수 없으며 오류가 발생한다. 또한, 문자열은 시작 문자부터 '\0' 문자가 나올 때까지를 하나의 문자열로 처리된다.

```
char c[] = "C C++ Java" ;
c[5] = '\0';
printf( "%sn\n%s\n", c, (c+6));
```

C 언어에서는 문자열을 저장하기 위한 자료형을 제공하지 않기 때문에 문자열을 저장하려면 문자의 모임인 '문자 배열'을 사용한다. 여기서 주의해야 할 점은 문자열의 마지막을 의미하는 NULL 문자 '\0'이 마지막에 저장되어야 한다. 그러므로 문자열이 저장되는 배열 크기는 반드시 저장될 문자 수보다 1이 커야 한다.

```
char ch = 'A';

char csharp[3];
csharp[0] = 'C';   csharp[1] = '#';   csharp[2] = '\0';
```

배열 선언 시 초기화 방법을 이용하면 문자열을 쉽게 저장할 수 있다. 여기서 주의할 점은 중괄호를 사용해야 하며, 문자 하나하나를 쉼표로 구분하여 입력하고 마지막 문자로 널(NULL)인 '\0'을 삽입해야 한다는 것이다.

```
char java[] = { 'J', 'A', 'V', 'A', '\0' };
```

문자열을 선언하는 편리한 다른 방법은 배열 선언 시 저장할 큰따옴표를 사용해 문자열 상수를 바로 대입하는 방법이다. 여기서 배열 초기화 시 배열 크기는 지정하지 않는 것이 더 편리하며, 만일 지정한다면 마지막 문자인 '\0'을 고려해 실제 문자 수보다 1이 더 크게 배열 크기를 지정해야 한다. 만일 지정한 배열 크기가 (문자수 + 1)보다 크면 나머지 부분은 모두 '\0'문자로 채워진다. 반대로 배열 크기가 작으면 문제가 발생한다.

```
char c[] = "C language";
```

문자열을 처리하는 다른 방법은 문자열 상수를 문자 포인터에 저장하는 방식이다. 문자 포인터 변수에 문자열 상수를 저장할 수 있다. 문자열 출력도 함수 prinft()에서 포인터 변수와 형식제어문자 %s로 간단히 처리할 수 있다. 이와 같이 문자 포인터에 의한 선언으로는 문자 하나하나의 수정은 할 수 없다. 반복문을 이용하여 문자가 '\0'이 아니면 문자를 출력하도록 할 수도 있다.

```
int i = 0;
char *java = "java";
printf("%s", java);
```

```
while (java[i] != '\0')
    printf("%c", java[i++]);
printf("\n");
```

```c
#include <stdio.h>

int main(void) {
    //문자 선언과 출력
    char ch = 'A';
    printf("%c %d₩n", ch, ch);

    //문자열 선언 방법1
    char java[] = { 'J', 'A', 'V', 'A', '₩0' };
    printf("%s₩n", java);

    //문자열 선언 방법2
    char c[] = "C language";
    printf("%s₩n", c);

    //문자열 선언 방법3
    char csharp[5] = "C#";
    printf("%s₩n", csharp);

    //문자 배열에서 문자 출력
    printf("%c%c₩n", csharp[0], csharp[1]);

    return 0;
}
```

=> 결과

```
A 65
JAVA
C language
C#
C#
```

<해석>

변수 ch에 문자 'A'를 저장하여 %c로 ch를 출력하고, %d로 문자 코드값 출력한다. 문자 배열을 선언하면서 문자 하나 하나를 초기화할 경우, 마지막에 '\0'을 삽입해야 하고, 문자 배열을 선언하면서 문자열 상수를 초기화할 경우, 배열 크기를 생략하면 간편하다. 배열 크기를 지정한다면 문자열 상수의 문자수보다 1이 더 크게 지정하며, 남은 공간은 모두 '\0'가 자동으로 저장된다. 문자열이 저장된 문자배열에서 %c로 문자배열 원소를 지정하면 문자 하나를 출력할 수 있다.

- 문자 출력 함수 : putchar, fputc

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

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

=> 함수호출 성공 시 쓰여진 문자정보가, 실패 시 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, fputc 함수는 문자를 전송할 스트림을 지정할 수 있다. 즉, fputc 함수를 이용하면 stdout 뿐만 아니라, 파일을 대상으로도 데이터를 전송할 수 있다.

- 문자 입력 함수 : getchar, fgetc, getche, getch

키보드로부터 하나의 문자를 입력받을 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 입력받아서 반환하는 함수이다. 따라서 키보드로부터 하나의 문자를 입력받는 함수라 할 수 있다. getchar는 라인 버퍼링(line buffering) 방식을 사용하므로 문자 하나를 입력해도 반응을 보이지 않다가 [enter] 키를 누르면 그제야 이전에 입력한 문자마다 입력이 실행된다. 그리고 fgetc 함수도 하나의 문자를 입력받는 함수이다. 다만 getchar 함수와 달리 문자를 입력받을 스트림을 지정할 수 있다. getche 함수는 버퍼를 사용하지 않으므로 문자 하나를 입력하면 바로 함수가 실행된다. 예를 들어, "inputq"라면 화면에는 "iinnppuuttq"가 보이게 된다. 즉, 화면에 보이는 행이 표준입력과 표준출력이 번갈아 가면서 나오게 되므로 한 문자가 두 번씩 나오게 된다. getch 함수는 입력한 문자가 화면에 보이지 않는 특성이 있다. 그러므로 입력된 문자를 따로 출력하지 않으면 입력 문자가 화면에 보이지 않게 된다. 이 함수도 버퍼를 사용하지 않는 문자 입력 함수이다. 예를 들어, "inputq'를 입력으로 실행하면 "input"이 출력된다. getche와 getch 함수를 이용하기 위해서는 conio.h를 삽입해야 한다.

```c
#include <stdio.h>

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;
}
```

=> 결과

P

P

<해석>

소스코드상에서는 분명 두 개의 문자를 입출력하고 있다. 하지만 실행 결과만 놓고 보면, 하나의 문자가 입력되고 출력된 것으로 보인다. 그러나 실제로는 두 개의 문자가 입력되고 출력되었다. 다만 두 번째 문자가 '엔터 키'이다 보니 눈에 띄질 않았을 뿐이다. 사실 '엔터 키'도 아스키 코드값이 10인 '\n'으로 표현되는 문자이다. 7, 8행은 키보드로부터 각각 하나의 문자를 입력받고 있으며, 두 문장이 하는 일은 동일하다. 10, 11행은 모니터로 각각 하나의 문자를 출력하고 있으며, 두 문장이 하는 일도 동일하다.

```
1   #include <stdio.h>
2   #include <conio.h>
3
4   int main(void) {
5       char ch;
6
7       printf("문자를 계속 입력하고 Enter를 누르면>>₩n");
8       while ((ch = getchar()) != 'q')
9           putchar(ch);
10
11      printf("₩n문자를 누를 때마다 두 번 출력>>₩n");
12      while ((ch = _getche()) != 'q')
13          putchar(ch);
14
15      printf("₩n문자를 누르면 한 번 출력>>₩n");
16      while ((ch = _getch()) != 'q')
17          _putch(ch);
18      printf("₩n");
19
20      return 0;
21  }
22
```

=> 결과

java

java

python

python

q

문자를 누를 때마다 두 번 출력 >>

jjaavvaaq

문자를 누르면 한 번 출력 >>

java

<해석>

8행에서는 함수 getchar로 입력한 문자를 변수 ch에 저장하여 'q'가 아니면 반복 몸체인 putchar(ch)를 실행하고, 변수 ch에 'q'이면 반복 종료이기 때문에 출력이 되지 않는다. 12행에서는 함수 _getche로 입력한 문자를 변수 ch에 저장하여 'q'가 아니면 반복 몸체인 putchar(ch)를 실행하고, 변수 ch에 'q'이면 반복 종료라서 출력이 되지 않는다. 함수 _getche는 입력한 문자마다 바로 처리하는 특징이 있다. 16행에서는 함수 _getch로 입력한 문자를 변수 ch에 저장하여 'q'가 아니면 반복 몸체인 putchar(ch)를 실행하고, 변수 ch에 'q'이면 반복 종료라서 출력이 되지 않는다. 함수 _getch는 입력한 문자마다 바로 처리하며, 입력한 문자도 보이지 않는 특징이 있다.

- 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 없다'는 뜻이 된다.

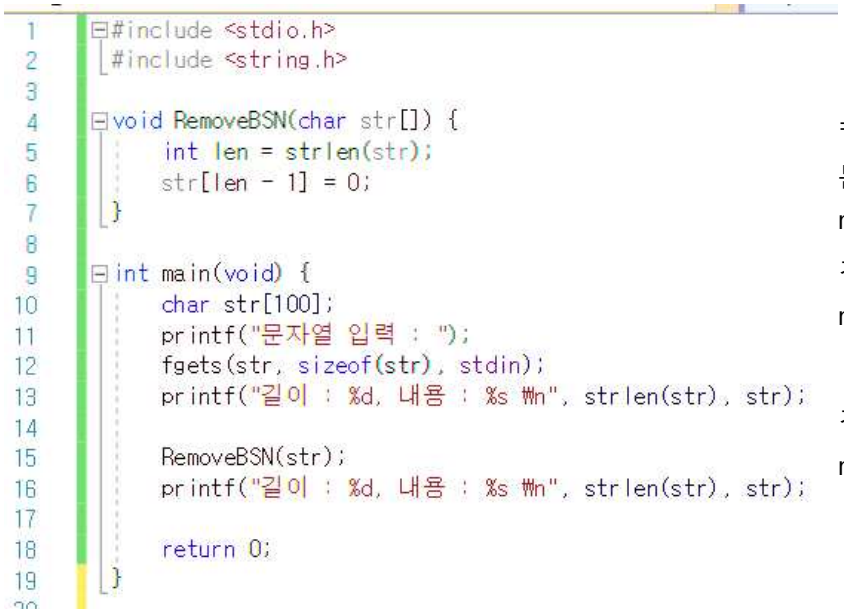
- 문자열 관련 함수

void *memchr(const void *str, int c, size_t n)	메모리 str에서 n바이트까지 문자 c를 찾아 그 위치를 반환
void *memcmp(const void *str1, const void *str2, size_t n)	메모리 str1과 str2를 첫 n 바이트를 비교 검색하여 같으면 0, 다르면 음수 또는 양수 반환
void *memcpy(void *dest, const void *src, size_t n)	포인터 src 위치에서 dest 에 n 바이트를 복사한 후 dest 위치 반환
void *memmove(void *dest, const void *src, size_t n)	포인터 src 위치에서 dest 에 n 바이트를 복사한 후 dest 위치 반환
void *memset(void *str, int c, size_t n)	포인터 str 위치에서부터 n 바이트까지 문자 c를 지정한 후 str 위치 반환
size_t strlen(const char *str)	포인터 str 위치에서부터 널 문자를 제외한 문자열의 길이 반환

- 문자열의 길이를 반환하는 함수 : strlen

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

=> 전달된 문자열의 길이를 반환하되, 널 문자는 길이에 포함하지 않는다.



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

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;
}
```

=> 결과

문자열 입력 : Good morning

길이 : 13, 내용 : Good morning

길이 : 12, 내용 : Good morning

<해석>

5, 6행에서 문자열의 길이를 계산하여 \n이 저장된 위치에 널 문자의 아스키 코드값 0을 저장학 있다. 이로써 \n은 문자열에서 사라진 셈이다. 12행에서 fgets 함수호출을 통해서 문자열을 입력받고 있다. 따라서 \n문자가 문자열의 일부로 포함된다. 13행을 통한 출력에서는 개행이 두 번 이뤄졌다. 그런데 15행의 RemoveBSN 함수호출 이후에 16행의 출력에서는 개행이 한 번 이뤄졌다. 이는 RemoveBSN 함수호출을 통해서 \n 문자가 소멸되었기 때문이다.

- 문자열을 복사하는 함수 : strcpy, strncpy

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

=> 복사된 문자열의 주소값 반환

```c
#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;
}
```

=> 결과

1234567890

(쓰레기값)

1234

<해석>

배열의 str3의 길이가 5이기 때문에 총 5개의 문자가 복사된다. 단, 이 5개의 문자 안에 널 문자가 포함되지 않는다는 문제가 있다. strncpy 함수는 문자열을 단순하게 복사한다. 5개의 문자를 복사하라고 하면 앞에서부터 딱 5개의 문자만 복사한다. 마지막 문자가 널 문자인지 아닌지는 상관하지 않는다. 그래서 15행의 출력결과가 이상한 것이다. 널 문자가 존재해야 널 문자 이전까지 출력을 할 텐데, 널 문자가 존재하지 않으니 엉뚱한 영역까지 출력을 하는 것이다.

- 문자열을 덧붙이는 함수 : strcat, strncat

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

=> 덧붙여진 문자열의 주소값 반환

```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";

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

    /**** case 2 ****/
    strncat(str3, str4, 7);
    puts(str3);

    return 0;
}
```

=> 결과

First~Second~

Simple num: 1234567

<해석>

12행에서 str2에 저장된 문자열을 str1에 저장된 문자열의 뒤에 덧붙이고 있다. 이 때 str2에 저장된 문자열을 덧붙일 수 있을 만큼의 충분한 공간이 str1에 있어야 한다. 16행에서는 str4에 저장된 문자열을 str3에 저장된 문자열의 뒤에 덧붙이되 7개의 문자만 덧붙이고 있다. 따라서 널 문자를 포함해서 8개의 문자가 덧붙여진다.

- 문자열을 비교하는 함수 : strcmp, strncmp

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

=>두 문자열의 내용이 같으면 0, 다른 경우 앞 문자가 작으면 음수, 뒤 문자가 작으면 양수

```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;
}
```

=> 결과

문자열 입력 1: Simple

문자열 입력 2: Simon

두 문자열은 동일하지 않습니다.

그러나 앞 세 글자는 동일합니다.

<해석>

12행에서 str1과 str2가 동일하면 거짓을 의미하는 0이 반환된다. 그런데 이 반환 값을 대상으로 ! 연산을 하였으니 거짓은 참으로 바뀐다. 즉, 이 if문은 str1과 str2의 문자열이 완벽히 동일할 때 참이 된다. 18행 이 문장은 두 문자열이 일치하지 않는 경우에 한해서 실행된다. 그리고 strncmp 함수의 세 번째 인자로 3이 전달되었으니, 앞의 세 문자가 동일한 경우에 한해서 if문이 참이 되어 19행을 실행하게 된다.

- 그 외의 변환 함수들

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
char * strlwr(char * str)	문자열 str을 모두 소문자로 변환
char * strupr(char * str)	문자열 str을 모두 대문자로 변환

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

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;
}
```

=> 결과

정수 입력 : 15

15

실수 입력 : 12.456

12.456

<해석>

7, 8행에서 프로그램 사용자로부터 문자열을 입력받고 있다. 그리고 그 문자열에 담긴 내용을 정수로 변환해서 서식문자 %d로 출력하고 있다. 11, 12행에서는 프로그램 사용자로부터 입력받은 문자열을 실수로 변환해서 서식문자 %g로 출력하고 있다. 만약에 위의 함수들을 모르는 상태였다면, 문자열 "123"을 정수 123으로 바꾸는 것이나 문자열 "32.45"를 실수 32.45로 바꾸는 것은 간단하지 않은 일이 되어버릴 것이다.

- 문자열을 구분하는 함수 : strtok

```c
#include <stdio.h>
char * strtok(char * str, const char * delim);
char * strtok_s(char * str, const char * delim, char ** context);
```

=> 구분자인 문자를 지정하여 문자열 구분

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

int main(void) {
    char str1[] = "C and C++₩t language are best!";
    char *delimiter = " ,₩t!";
    //char *next_token;

    printf("문자열 ₩"%s₩"을 >>₩n", str1);
    printf("구분자[%s]를 이용하여 토큰을 추출 >>₩n", delimiter);
    char* ptoken = strtok(str1, delimiter);
    //char *ptoken = strtok_s(str, delimiter, &next_token);
    while(ptoken != NULL) {
        printf("%s₩n", ptoken);
        ptoken = strtok(NULL, delimiter); //다음 토큰을 반환
        //ptoken = strtok_s(NULL, delimiter, &next_token);  //다음 토큰을 반환
    }

    return 0;
}
```

=> 결과

문자열 "C and C++ language are best!"을 >>

구분자[, !]를 이용하여 토큰을 추출 >>

C

and

C++

language

are

best

<해석>

문자열에서 \t는 탭 문자를 뜻하며, 10행에서는 원 문자열을 11행에서는 구분자 문자열을 출력한다. 함수 strtok(str1, delimiter) 호출에 의해 분리되는 문자열 토큰이 저장될 문자 포인터 선언하여 strtok() 호출값을 저장한다. 분리된 토큰이 NULL이 아니면 14행의 반복문을 실행하고, 15행에서 분리된 토큰을 한 줄에 출력한다.

- **LAB**

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

int main(void) {
    char s[50];
    memcpy(s, "C Programming!", strlen("C Programming!") + 1);
    printf("%s₩n", s);

    reverse(s);
    printf("%s₩n", s);


    return 0;
}

void reverse(char str[]) {
    for (int i = 0, j = strlen(str) - 1; i < j; i++, j--) {
        char c = str[i];
        str[i] = str[j];
        str[j] = c;
    }
}

```

=> 결과

C Programming!

!gnimmargorP C

<해석>

6행의 memcpy를 통해 변수 s에 C Programming!을 복사하여 저장한다. 3번째 인자의 strlen에서 마지막에 +1을 해줌으로써 '\0'까지 복사하여 문장의 끝을 알려주고, 오류가 나지 않도록 해준다. reverse(s)를 통해 reverse함수로 이동하여 str을 역순으로 다시 저장하고, 10행의 printf문을 통해 출력된다.

11.3 여러 문자열 처리

- 문자 포인터 배열과 이차원 문자 배열

문자 포인터 배열은 여러 개의 문자열을 참조할 수 있다. 문자 포인터를 사용해서는 문자열 상수의 수정은 불가능하다.

```
char * pa[] = {"JAVA", "C#", "C++"};
printf("%s ", pa[0]); printf("%s ", pa[1]); printf("%s ", pa[2]);
```

이차원 문자 배열은 배열선언에서 이차원 배열의 열 크기는 문자열 중에서 가장 긴 문자열의 길이보다 1 크게 지정해야 한다. 행의 크기는 문자열 수에 해당하므로 지정하거나 공백으로 비워 둔다. 이차원 배열은 배열 포인터 배열과는 달리 문자열을 수정할 수 있다.

```
char ca[][5] = {"JAVA", "C#", "C++"};
printf("%s ", ca[0]); printf("%s ", ca[1]); printf("%s ", ca[2]);
```

```c
#include <stdio.h>

int main(void) {
    char* pa[] = { "JAVA", "C#", "C++" };
    char ca[][5] = { "JAVA", "C#", "C++" };

    //각각 3개 문자열 출력
    //pa[0][2] = 'v';  //실행 오류 발생
    //ca[0][2] = 'v'; //수정 가능
    printf("%s ", pa[0]); printf("%s ", pa[1]); printf("%s₩n", pa[2]);
    printf("%s ", ca[0]); printf("%s ", ca[1]); printf("%s₩n", ca[2]);

    //문자 출력
    printf("%c %c %c₩n", pa[0][1], pa[1][1], pa[2][1]);
    printf("%c %c %c₩n", ca[0][1], ca[1][1], ca[2][1]);


    return 0;
}
```

=> 결과

JAVA C# C++

JAVA C# C++

A # +

A # +

<해석>

문자 포인터 배열 pa를 선언하여 초기값으로 3개의 문자열을 저장하고, 열 크기가 3인 이차원 문자 배열 ca를 선언하는 동시에 초기값을 저장한다. 10행의 pa[0], pa[1], pa[2]와 11행의 ca[0], ca[1], ca[2] 모두 문자열을 가리키는 포인터로 함수 prinf에서 %s로 문자열 출력이 가능하다. 14행에서의 pa[0][1]. pa[1][1], pa[2][3]과 15행에서의 ca[0][1], ca[1][1], ca[2][1] 모두 세 문자열에서 모두 두 번째 문자가 저장된 변수로 함수 printf에서 %c로 문자 출력할 수 있다.

- **LAB**

```c
#include <stdio.h>

int main(void) {
    char str1[] = "JAVA";
    char str2[] = "C#";
    char str3[] = "C++";

    char* pstr[] = { str1, str2, str3 };

    //각각의 3개 문자열 출력
    printf("%s ", pstr[0]);
    printf("%s ", pstr[1]);
    printf("%s₩n", pstr[2]);

    //문자 출력
    printf("%c %c %c₩n", str1[0], str2[1], str3[2]);
    printf("%c %c %c₩n", pstr[0][1], pstr[1][1], pstr[2][1]);


    return 0;
}
```

=> 결과

JAVA C# C++

J # +

A # +

<해석>

일차원 배열 str1, str2, str3을 선언하면서 문자열을 저장하고, pstr을 선언하면서 문자열 포인터인 str1, str2, str3을 저장한다. 11, 12, 13행에서 함수 printf를 통해 각각의 3개 문자열을 출력하고, 16, 17행에서 각각의 문자를 %c로 출력한다.

12.1 전역변수와 지역변수

- 변수에 대해

변수는 선언되는 위치에 따라 크게 '전역 변수'와 '지역 변수'로 나뉜다. 그리고 이 둘은 [메모리상에 존재하는 기간], [변수에 접근할 수 있는 범위]에 대해 차이점을 보인다.

- 함수 내에만 존재 및 접근 가능한 지역변수(Local Variable)

'지역변수'에서 말하는 '지역'이란 중괄호에 의해 형성되는 영역을 뜻한다. 따라서 중괄호 내에 선언되는 변수는 모두 지역변수다. 그러므로 선언된 지역 내에서만 유효하다는 특성을 지니고, '스택(stack)'이라는 메모리 영역에 할당된다. 지역변수는 선언된 부분에서 자도응로 생성되고 함수나 블록이 종료되는 순간 메모리에서 자동으로 제거된다. 그래서 자동변수라고도 한다. 키워드 auto를 사용할 수 있으며 생략이 가능하다. 지역변수는 선언 후 초기화하지 않으면 쓰레기값이 저장되므로 주의해야 한다.

```c
#include <stdio.h>

int SimpleFuncOne(void) {
    int num = 10; //이후부터 SimpleFuncOne의 num 유효
    num++;
    printf("SimpleFucOne num : %d ₩n", num);

    return 0; //SimpleFuncOne의 num이 유효한 마지막 문장
}

int SimpleFuncTwo(void) {
    int num1 = 20; //이후부터 num1 유효
    int num2 = 30; //이후부터 num2 유효
    num1++;  num2--;
    printf("num1 & num2 : %d %d ₩n", num1, num2);

    return 0; //num1, num2 유효한 마지막 문장
}


int main(void) {
    int num = 17;  //이후부터 main의 num 유효
    SimpleFuncOne();
    SimpleFuncTwo();
    printf("main num : %d ₩n", num);

    return 0; //main의 num이 유효한 마지막 문장
}
```

=> 결과

SimpleFuncOne num : 11

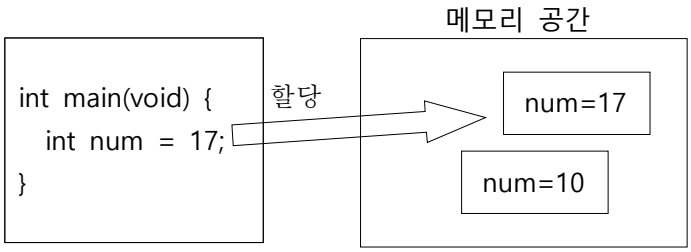
num1 & num2 : 21 29

main num : 17

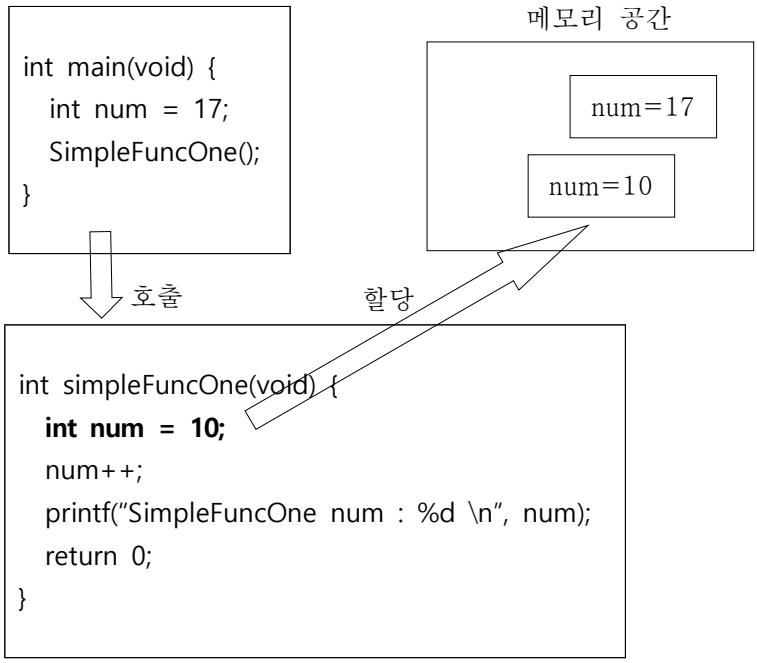
<해석>

위의 함수에 선언된 변수 num은 SimpleFuncOne이라는 함수의 중괄호 안에 선언되었으므로 지역변수이다. 따라서 이 변수는 선언된 이후부터 SimpleFuncOne 함수를 빠져나가기 직전까지만 유효하다. 왜냐하면 '지역변수는 해당지역을 벗어나면 자동으로 소멸'되기 때문이다. 그리고 지역변수는 선언된 지역 내에서만 유효하기 때문에 선언된 지역이 다르면 이름이 같아도 문제가 되지 않는다. 그렇기 때문에 main 함수 내에도, 그리고 SimpleFuncOne 함수 내에도 동일한 이름의 변수 num이 선언될 수 있는 것이다.

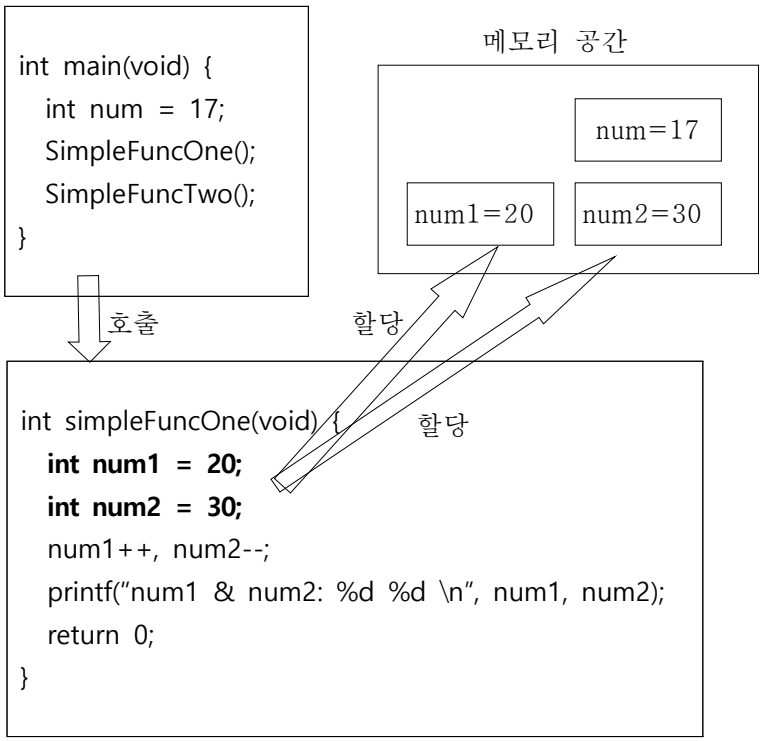
1. main 함수의 호출



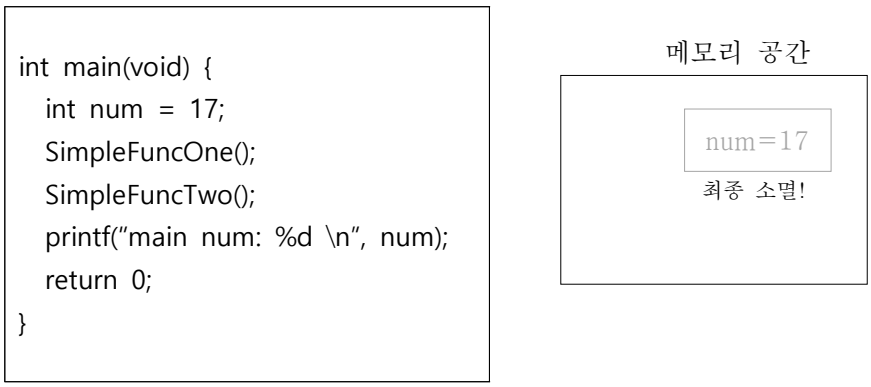
2. SimpleFuncOne 함수의 호출과 종료(반환)



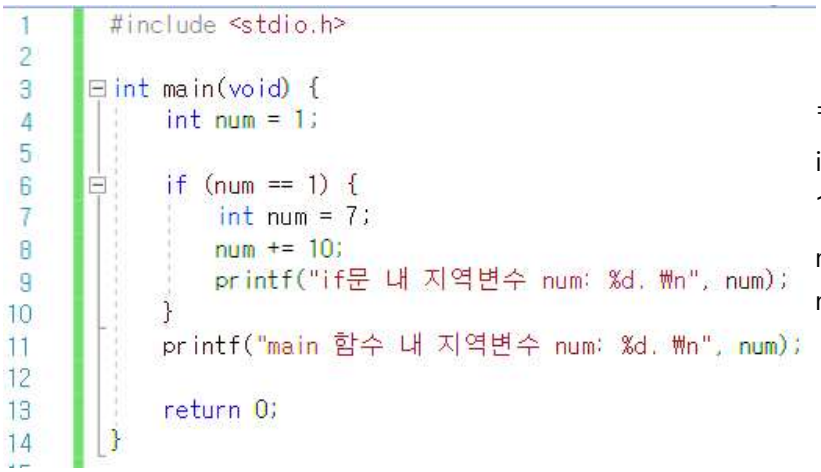
3. SimpleFuncTwo 함수의 호출과 종료(반환)



4. main 함수의 종료(반환)



지역 변수는 반복문이나 조건문에도 선언이 가능하다.



```c
#include <stdio.h>

int main(void) {
    int num = 1;

    if (num == 1) {
        int num = 7;
        num += 10;
        printf("if문 내 지역변수 num: %d. ₩n", num);
    }
    printf("main 함수 내 지역변수 num: %d. ₩n", num);

    return 0;
}
```

=> 결과

if문 내 지역변수 num: 17.

main 함수 내 지역변수 num: 1.

<해석>

4행에 지역변수 num이 선언되었다. 따라서 이 변수는 main함수 내에서 접근 가능하다. 그런데 if문의 내부인 7행에서 동일한 이름의 변수 num이 선언되었다. 이러한 경우 if문 내에서는 main함수의 num이 가리워진다. 그렇기 때문에 8행에서 num은 4행의 num이 아닌 7행의 num이 된다. 물론 7행의 변수 선언이 없었다면, 8행에서 num은 4행의 num이 된다.

- 전역변수의 선언방법

전역변수는 그 이름이 의미하듯이 어디서든 접근이 가능한 변수로써 지역변수와 달리 중괄호 내에 선언되지 않는다. 외부변수라고도 부르며, 일반적으로 프로젝트의 모든 함수나 블록에서 참조할 수 있다. 별도의 값으로 초기화하지 않으면 0으로 초기화되며, 프로그램의 시작과 동시에 메모리 공간에 할당되어 종료 시까지 존재한다. 전역변수와 같은 이름으로 지역변수를 선언하면 지역변수로 인식하므로 가능한 이러한 변수는 사용하지 않도록 한다. 키워드 extern을 사용하여 이미 다른 파일에서 선언된 전역변수임을 선언해야 한다. extern 참조 구문에서 자료형은 생략할 수 있으며, 변수 선언 맨 앞에 extern을 넣는 구조이다.

```c
#전역 변수와 동일한 이름의 지역변수 선언한 예시

#include <stdio.h>

int Add(int val);
int num = 1;

int main(void) {
    int num = 5;
    printf("num : %d ₩n", Add(3));
    printf("num : %d ₩n", num + 9);

    return 0;
}

int Add(int val) {
    int num = 9;
    num += val;

    return num;
}
```

=> 결과

num : 12

num : 14

<해석>

실행결과는 '동일한 이름의 지역변수는 해당영역 내에서 동일한 이름의 전역변수를 가린다'는 사실을 확인시켜준다. 위에서 10행의 출력결과가 12이다. 이는 인자로 전달된 정수 3이 17행에 선언된 지역변수 num과 더해진 결과이다. 그리고 11행의 출력결과는 14이다. 이는 9행에 선언된 지역변수 num과 더해진 결과이다.

- LAB

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

//전역변수
int count;
//함수원형
void fibonacci(int prev_number, int number);


void main() {
    //자동 지역변수
    auto prev_number = 0, number = 1;

    printf("피보나츠를 몇 개 구할까요?(3 이상) >> ");
    //전역변수를 표준입력으로 저장
    scanf("%d", &count);
    if (count <= 2) {
        return 0;
    }
    printf("1 ");
    fibonacci(prev_number, number);
    printf("₩n");
}
```

=> 결과

피보나츠를 몇 개 구할까요?(3 이상) >> 20

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

12.2 정적 변수와 레지스터 변수

- 기억부류와 레지스터 변수

기억부류 종류	전역	지역
auto	X	O
register	X	O
static	O	O
extern	O	X

키워드 exten을 제외하고 나머지 3개의 기억부류의 변수선언에서 초기값을 저장할 수 있다. auto는 지역변수 선언에 사용되며 생략가능하다.

- 키워드 register

레지스터 변수는 변수의 저장공간이 일반 메모리가 아니라 CPU 내부의 레지스터에 할당되는 변수이다. 레지스터는 CPU 내부에 있는 기억장소이므로 일반 메모리보다 빠르게 참조될 수 있다. 그러므로 레지스터 변수는 처리 속도가 빠른 장점이 있다. 레지스터 변수는 일반 메모리에 할당되는 변수가 아니므로 주소연산자 &를 사용할 수 없다. 주로 레지스터 변수는 처리 속도를 증가시키려는 변수에 이용한다. 특히 반복문의 횟수를 제어하는 제어변수에 이용하면 효과적이다. 레지스터 변수도 일반 지역변수와 같이 초기값이 저장되지 않으면 쓰레기값이 저장된다.

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

int main(void) {
    //레지스터 지역변수 선언
    register int sum = 0;

    //메모리에 저장되는 일반 지역변수 선언
    int max;
    printf("양의 정수 입력 >> ");
    scanf("%d", &max);

    //레지스터 블록 지역변수 선언
    for (register int count = 1; count <= max; count++) {
        sum += count;
    }
    printf("합 : %d₩n", sum);

    return 0;
}
```

=> 결과

양의 정수 입력 >> 8

합 : 36

<해석>

레지스터 변수 sum을 선언하면서 초기값으로 0을 저장하였는데 만약 초기값이 없으면 쓰레기값이 저장된다. 일반 지역변수 max도 마찬가지로 초기값이 없으면 쓰레기값이 저장된다. 레지스터 블록 변수 count 선언하면서 초기값으로 1을 저장하고, count는 max까지 반복하면서 함수 몸체인 sum += count를 실행한다. 함수에서 count가 1부터 max까지 변하므로 1부터 max까지의 합이 sum에 저장되어 출력된다.

- 정적 변수 static

정적 변수는 정적 지역변수와 정적 전역변수로 나눌 수 있다. 정적변수는 초기 생성된 이후 메모리에서 제거되지 않으므로 지속적으로 저장값을 유지하거나 수정할 수 있는 특징이 있다. 정적 변수는 프로그램이 시작되면 메모리에 할당되고, 프로그램이 종료되면 메모리에서 제거된다. 초기값을 지정하지 않으면 자동으로 자료형에 따라 0이 저장되고, 초기화는 단 한 번만 수행된다. 단, 초기화는 상수로만 가능하다.

```
int a = 1;
static s = a;
```

=> static에 변수를 대입할 수는 없다.

```
int a = 1;
static s = 1;
```

=> static은 상수를 대입해야 한다.

- 정적 지역변수

함수나 블록에서 정적으로 선언되는 변수가 정적 지역변수이다. 유효 범위는 선언된 블록 내부에서만 참조 가능하다. 정적 지역변수는 함수나 블록을 종료해도 메모리에서 제거되지 않고 계속 메모리에 유지 관리되는 특성이 있다.

```c
#include <stdio.h>

void SimpleFunc(void) {
    static int num1 = 0;  //초기화하지 않으면 0 초기화
    int num2 = 0;  //초기화하지 않으면 쓰레기 값 초기화
    num1++;  num2++;
    printf("static : %d, local : %d \n", num1, num2);
}

int main(void) {
    int i;
    for (i = 0; i < 3; i++) {
        SimpleFunc();
    }
    return 0;
}
```

=> 결과

static : 1, local : 1

static : 2, local : 1

static : 3, local : 1

<해설>

static으로 선언된 지역변수는 전역변수와 동일한 시기에 할당되고 소멸된다. 단, 지역변수와 마찬가지로 선언된 함수 내에서만 접근이 가능하다. num1은 static 지역변수로 함수가 끝나도 메모리가 남아있어 값이 계속 1씩 증가하는 반면, 일반변수인 num2는 함수가 끝나면 메모리도 삭제되어 다시 1이 된다.

- 정적 전역변수

함수 외부에서 정적으로 선언되는 변수가 정적 전역변수이다. 일반 전역변수는 파일 소스가 다르더라도 extern을 사용하여 참조가 가능한 반면, 정적 전역변수는 선언된 파일 내부에서만 참조가 가능한 변수이다. 즉, 정적 전역변수는 extern에 의해 다른 파일에서 참조가 불가능하다.

```c
#include <stdio.h>

//정적 전역변수 선언
static int svar;
//전역변수 선언
int gvar;

//함수원형
void increment();
void testglobal();
//void teststatic();

int main(void) {
    for (int count = 1; count <= 5; count++) {
        increment();
    }
    printf("함수 increment()가 총 %d번 호출되었습니다.₩n", svar);

    testglobal();
    printf("전역 변수 : %d₩n", gvar);
    //teststatic();
}

//함수 구현
void increment() {
    svar++;
}
```

=>결과

함수 increment()가 총 5번 호출되었습니다.

전역 변수 : 10

```c
void teststatic() {
    //정적 전역변수는 선언 및 사용 불가능
    //estern svar;
    //svar = 5l
}

void testglobal() {
    //전역변수는 선언 및 사용 가능
    extern gvar;
    gvar = 10;
}
```

<해설>

변수 count는 자동 지역변수로 for문 내부에서만 사용이 가능하며, 1~5까지 변하면서 몸체인 함수 increment을 호출한다. 정적 전역변수 svar는 다른 파일에서 사용 불가하여 실행 시 오류가 발생한다.

- **LAB**

```c
#include <stdio.h>

void process();

int main() {
    process();
    process();
    process();

    return 0;
}

void process() {
    //정적 변수
    static int sx;
    //지역 변수
    int x = 1;

    printf("%d %d₩n", x, sx);

    x += 3;
    sx += x + 3;
}
```

=> 결과

1 0

1 7

1 14

12.3 메모리 영역과 변수 이용

- **메모리 영역**

extern	auto	register	static
전역 변수	지역 변수	지역 변수	지역, 전역 변수
여러 프로그램 모듈에서 함께 참조	함수/블록 내부에서만 참조	함수/블록 내부에서만 참조	함수/블록/파일 내부에서 참조
주기억장치에 저장	주기억장치에 저장	CPU 레지스터에 저장	주기억장치에 저장
기본값 : 0	기본값 : 쓰레기값	기본값 : 쓰레기값	기본값 : 0
데이터 영역	스택 영역	CPU	데이터 영역

데이터 영역 : 전역변수와 정적변수가 할당되는 저장공간

메모리 주소가 낮은 값에서 높은 값으로 저장 장소가 할당된다.

프로그램이 시작되는 시점에 정해진 크기대로 고정된 메모리 영역이 확보된다.

=> 프로그램이 종료되어야 데이터 영역이 해제된다.

힙 영역 : 동적 할당되는 변수가 할당되는 저장공간

데이터 영역과 스택 영역 사이에 위치한다.

메모리 주소가 낮은 값에서 높은 값으로 사용하지 않은 공간이 동적으로 할당된다

=> 시스템이 알아서 지정한다.

프로그램이 실행되면서 영역 크기가 계속적으로 변한다.

스택 영역 : 함수 호출에 의한 형식 매개변수, 함수 내부의 지역변수가 할당되는 저장공간

=> 정적 지역변수는 해당되지 않는다.

메모리 주소가 높은 값에서 낮은 값으로 저장 장소가 할당된다

함수 호출과 종료에 따라 높은 주소에서 낮은 주소로 할당되었다가 다시 제거되는 작업이 반복된다.

=> 함수가 없으면 스택 영역은 없을 수도 있다.

프로그램이 실행되면서 영역 크기가 계속적으로 변한다

- 변수의 이용

* 일반적으로는 전역변수의 사용을 자제하고, 지역변수를 주로 이용한다.
* 실행 속도를 개선하고자 하는 경우에는 레지스터 변수를 이용한다.
* 함수나 블록 내부에서 함수나 블록이 종료되더라도 계속적으로 값을 저장하고 싶을 때는 정적 지역변수를 이용한다.
* 파일 내부에서만 변수를 공유하고자 하는 경우는 정적 전역변수를 이용한다.
* 프로그램의 모든 영역에서 값을 공유하고자 하는 경우는 전역변수를 이용한다.

변수의 종류

<table>
<tr><th>선언위치</th><th>상세 종류</th><th>키워드</th><th>유효범위</th><th>기억장소</th><th>생존기간</th></tr>
<tr><td rowspan="2">전역</td><td>전역 변수</td><td>참조선언/extern</td><td>프로그램 전역</td><td rowspan="3">메모리(데이터 영역)</td><td rowspan="3">프로그램 실행 시간</td></tr>
<tr><td>정적 전역변수</td><td>static</td><td>파일 내부</td></tr>
<tr><td rowspan="3">지역</td><td>정적 지역변수</td><td>static</td><td rowspan="3">함수나 블록 내부</td></tr>
<tr><td>레지스터 변수</td><td>register</td><td>레지스터</td><td rowspan="2">함수 또는 블록 실행 시간</td></tr>
<tr><td>자동 지역변수</td><td>auto(생략가능)</td><td>메모리(스택 영역)</td></tr>
</table>

변수의 유효범위

구분	종류	메모리할당 시기	동일 파일 외부 함수에서의 이용	다른 파일 외부 함수에서의 이용	메모리제거 시기
전역	전역변수	프로그램시작	O	O	프로그램종료
	정적 전역변수	프로그램 시작	O	X	프로그램종료
지역	정적 지역변수	프로그램 시작	X	X	프로그램종료
	레지스터 변수	함수(블록)시작	X	X	함수(블록)종료
	자동 지역변수	함수(블록)시작	X	X	함수(블록)종료

변수의 초기값

지역,전역	종류	자동 저장되는 기본 초기값	초기값 저장
전역	전역변수	0	프로그램 시작 시
	정적 전역변수		
지역	정적 지역변수		
	레지스터 변수	쓰레기값	함수나 블록이 실행될 때마다
	자동 지역변수		

```c
#include <stdio.h>

void infunction(void);
void outfunction(void);

/*전역변수*/
int global = 10;
/*정적 전역변수*/
static int sglobal = 20;

int main(void) {
    auto int x = 100;  /*main() 함수의 자동 지역변수*/

    printf("%d, %d, %d₩n", global, sglobal, x);
    infunction(); outfunction();
    infunction(); outfunction();
    infunction(); outfunction();
    printf("%d, %d, %d₩n", global, sglobal, x);

    return 0;
}
void infunction(void) {
    /*infunction() 함수의 자동 지역변수*/
    auto int fa = 1;
    /*infunction() 함수의 정적 지역변수*/
    static int fs;

    printf("₩t%d, %d, %d, %d₩n", ++global, ++sglobal, fa, ++fs);
}
```

```c
#include <stdio.h>

void outfunction() {
    extern int global, sglobal;

    printf("₩t₩t%d₩n", ++global);

    //외부 파일에 선언된 정적 전역변수이므로 실행 시 오류
    //printf("%d₩n", ++sglobal);
}
```

=> 결과

10, 20, 100

11. 21. 1. 1

12

13. 22. 1. 2

14

15, 23. 1, 3

16

16, 23, 100

<해석>

전역변수 global 선언하면서 10으로 초기화를 했다. 함수 외부이므로 전역변수이고 모든 프로젝트에서 사용이 가능하나 다른 파일이나 이 위치 위에서 사용하려면 extern int global; 선언이 필요하다. 정적 전역변수 sglobal 선언하면서 20으로 초기화했다. 함수 외부이므로 전역변수이지만 이 파일의 이 위치 이하에서는 사용이 가능하다. 변수 x는 자동 지역변수이고 100으로 초기화했다. 지역변수이므로 main함수 내에서만 사용가능하다.

```c
#include <stdio.h>

//전역변수
int total = 10000;

//입금 함수원형
void save(int);
//출금 함수원형
void withdraw(int);

int main(void) {
    printf(" 입금액 출금액   총입금액   총출금액    잔고₩n");
    printf("=============================================₩n");
    printf("%46d₩n", total);
    save(50000);
    withdraw(30000);
    save(60000);
    withdraw(20000);
    printf("=============================================₩n");

    return 0;
}

//입금액을 매개변수로 사용
void save(int money) {
    //총입금액이 저장되는 정적 지역변수
    static int amount;
    total -= money;
    amount += money;
    printf("%15d %20d %9d₩n", money, amount, total);
}
```

=> 결과

```
입금액      출금액      총입금액      총출금액      잔고
=======================================
                                      10000
50000                 50000           60000
           30000                30000  30000
60000                 110000          90000
           20000                50000  70000
=======================================
```

=> 해석

total은 전역변수로 파일 내부 어디서든 유효하고, money와 amount는 지역변수로 save 함수 내부에서만 유효하다.

13.1 구조체

- 구조체 정의

구조체란 정수, 문자, 실수, 포인터, 배열 등을 묶어 하나의 자료형으로 정의하는 도구이다. 즉 구조체를 기반으로 우리는 새로운 자료형을 정의할 수 있다. 예를 들어 프로그램상에서 마우스의 좌표정보를 저장한다고 가정한다면 아래처럼 두 개의 변수를 선언해야 한다.

```
int xpos;  // 마우스의 x 좌표
int ypos;  // 마우스의 y 좌표
```

하지만 이 둘은 항상 함께하기 때문에 서로 독립된 정보를 표현하는 것이 아니다. 마우스의 위치라는 하나의 정보를 표현하기 때문이다. 이런 이유로 등장한 것이 구조체이며, 구조체를 정의함으로써 위의 두 변수를 하나로 묶을 수 있게 된다.

<구조체 틀 정의>

```
struct point {  // point라는 이름의 구조체 정의
    int xpos; // point 구조체를 구성하는 xpos
    int ypos; // point 구조체를 구성하는 xpos
};
```

이 때, point라는 이름이 int나 double과 같은 자료형의 이름이 되는 것이다. 이 자료형을 대상으로 변수를 선언할 수 있는데 이렇게 선언된 변수를 '구조체 변수'라고 한다. 선언 방법은 아래처럼 맨 앞에 struct 선언을 추가해야하며, 이어서 구조체의 이름과 구조체 변수의 이름을 선언해야 한다.

<구조체 변수 선언>

```
struct point pos;
```

이렇게 구조체 변수가 선언되면, 아래의 형태로 존재하게 된다.

이처럼 구조체 변수 pos에는 int형 변수 xpos와 ypos가 존재한다. 구조체의 멤버에 접근할 때에는 접근연산자(.)을 사용하여 멤버를 참조할 수 있다.

```
pos.xpos = 20;  // 구조체 변수 pos의 멤버 xpos에 20을 저장
```

<정리>

① 키워드 struct 다음에 구조체 태그이름 기술

② 중괄호를 이용하여 원하는 멤버를 여러 개의 변수로 선언

③ 구조체 내부의 멤버 선언 구문은 모두 하나의 문장이므로 반드시 세미콜론으로 종료

④ 각 구조체 멤버의 초기값을 대입할 수 없다.

⑤ 멤버가 같은 자료형으로 연속적으로 놓일 경우, 간단히 콤마 연산사(,)를 사용하여 선언

⑥ 마지막 멤버 선언에도 반드시 세미콜론이 빠지지 않도록 주의

⑦ 구조체 내부에서 선언되는 구조체 멤버의 이름은 모두 유일해야 한다.

⑧ 구조체 정의는 변수의 선언과는 다른 것으로 변수 선언에서 이용될 새로운 자료형을 정의하는 구문이다.

⑨ 구조체 멤버로는 일반변수, 포인터 변수, 배열, 다른 구조체 변수, 구조체 포인터 모두 가능하다.

⑩ 실제 구조체의 크기는 멤버의 크기의 합보다 크거나 같다.

- 구조체 변수의 초기화

배열과 같이 구조체 변수도 선언 시 중괄호를 이용한 초기화 지정이 가능하다. 초기화 값은 중괄호 내부에서 구조체의 각 멤버 정의 순서대로 초기값을 쉼표로 구분하여 기술한다. 초기값이 지정되지 않은 멤버값은 자료형에 따라 0, 0.0, '\0' 등으로 저장된다.

```
struct point {
    int xpos;
    int ypos;
};

struct point pos = { 100, 200 };
```

=> 구조체의 크기는 자료형이 int형인 xpos와 ypos의 크기인 4바이트를 더한 8바이트보다 크거나 같다.

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

//은행 계좌를 위한 구조체 정의
struct account {
    char name[12];  //계좌주 이름
    int actnum;  //계좌번호
    double balance;  //잔고
};

int main(void) {
    //구조체 변수 선언 및 초기화
    struct account mine = { "홍길동", 1001, 300000 };
    struct account yours;

    strcpy(yours.name, "이동원");
    //strcpy_s(yours.name, 12, "이동원");  //가능
    //yours.name = "이동원";  //오류
    yours.actnum = 1002;
    yours.balance = 500000;


    printf("구조체크기 : %d₩n", sizeof(mine));
    printf("%s %d %.2f₩n", mine.name, mine.actnum, mine.balance);
    printf("%s %d %.2f₩n", yours.name, yours.actnum, yours.balance);

    return 0;
}
```

=> 결과

구조체 크기 : 24

홍길동 1001 300000.00

이동원 1002 500000.00

<해석>

6~10행까지 구조체를 정의한 후, 14, 15행에서 구조체 변수를 선언과 초기화를 한다. 구조체 정의 시 멤버변수와 중괄호 끝에 세미콜론 넣는 것 주의해야 한다. 또한, 선언과 동시에 중괄호를 사용하여 초기화하는 것이 아니라면 배열 name은 접근 연산자를 이용하여 참조할 수 없다. strcpy를 이용하여 변수에 복사하여야 한다. 포인터일 때는 string값이라도 접근 연산자를 이용하여 참조할 수 있다.

- 구조체 활용

구조체 멤버로 이미 정의된 다른 구조체 형 변수와 자기 자신을 포함한 구조체 포인터 변수를 사용할 수 있다.

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

 //날짜를 위한 구조체
struct date {
    int year;  //년
    int month;  //월
    int day;  //일
};

struct account
{
    struct date open;  //계좌 개설일자
    char name[12];  //계좌주 이름
    int actnum;  //계자번호
    double balance;  //잔고
};

int main(void) {
    struct account me = { {2018,3,9}, "홍길동", 1001, 300000 };

    printf("구조체크기 : %d₩n", sizeof(me));
    printf("[%d %d %d]₩n", me.open.year, me.open.month, me.open.day);
    printf("%s %d %.2f₩n", me.name, me.actnum, me.balance);

    return 0;
}
```

=> 결과

구조체크기 : 40

[2018, 3, 9]

홍길동 1001 300000.00

<해석>

5~9행은 날짜를 위한 구조체 struct date를 정의하기 위한 문장이고, 구조체 date 멤버 변수 선언문 마지막에는 세미콜론이 필요하다. 11~17행은 은행 계좌를 위한 구조체 struct date 정위한 부분으로 13행에 구조체 멤버로 구조체 struct date 형인 변수 open을 선언한 부분을 통해 구조체 안에 구조체 변수 선언이 가능하다는 것을 알 수 있다. 초기화 시 open은 구조체이므로 중괄호를 사용하여 초기화하는 편이 좋으나 없어도 상관은 없다.

- 구조체 변수의 대입과 동등비교

구조체형의 변수는 대입문이 가능하다. 하지만 구조체끼리의 동등비교는 할 수 없다. 만일 구조체를 비교하려면 구조체 멤버, 하나하나를 비교해야 한다.

```c
if( one == bae )  //오류
   printf("내용이 같은 구조체입니다. \n");
```

```c
if( one.snum == bae.snum )
   printf("학번이 %d로 동일합니다. \n", one.snum);
```

```c
if( one.snum == bae.snum && !strcmp(one.name, bae.name))
   printf("내용이 같은 구조체입니다. \n");
```

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

int main(void) {
    //학생을 위한 구조체
    struct student
    {
        int snum;  //학번
        char* dept;  //학과 이름
        char name[12];  //학생 이름
    };
    struct student hong = { 201800001, "컴퓨터정보공학과", "홍길동" };
    struct student na = { 201800002 };
    struct student bae = { 201800003 };

    //학생이름 입력
    scanf("%s", na.name);
    //na.name = "나한국";  //오류
    //scanf("%s", na.dept);  //오류

    na.dept = "컴퓨터정보공학과";
    bae.dept = "기계공학과";
    memcpy(bae.name, "배상문", 7);
    strcpy(bae.name, "배상문");
    strcpy_s(bae.name, 7, "배상문");

    printf("[%d, %s, %s]₩n", hong.snum, hong.dept, hong.name);
    printf("[%d, %s, %s]₩n", na.snum, na.dept, na.name);
    printf("[%d, %s, %s]₩n", bae.snum, bae.dept, bae.name);

    struct student one;
    one = bae;
    if (one.snum == bae.snum)
        printf("학번이 %d으로 동일합니다.₩n", one.snum);
    //if(one == bae)  //오류
    if (one.snum == bae.snum && !strcmp(one.name, bae.name) && !strcmp(one.dept, bae.dept))
        printf("내용이 같은 구조체입니다.");

    return 0;
}
```

=> 결과

나한국

[201800001, 컴퓨터정보공학과, 홍길동]

[201800002, 컴퓨터정보공학과, 나한국]

[201800003, 기계공학과, 배상문]

학번이 201800003으로 동일합니다.

내용이 같은 구조체입니다.

<해석>

19행에서 오류가 난 이유는 포인터가 아닌 문자배열인데 참조 연산자를 사용하여 참조했기 때문이고, 20행에서는 scanf는 포인터 변수는 사용이 불가하기 때문이다. 또한, 36행에서는 구조체끼리는 동등비교가 불가하기 때문에 오류가 발생한 것이다. 필드는 가능하다.

13.2 자료형 재정의

- 자료형 재정의 typedef

typedef는 이미 사용되는 자료 유형을 다른 새로운 자료형 이름으로 재정의할 수 있도록 하는 키워드이다. 일반적으로 자료형을 재정의하는 이유는 프로그램의 시스템 간 호환성과 편의성을 위해 필요하다.

```
typedef int integer, word;

integer myAge;    // int myAge와 동일
word yourAge;     // int yourAge와 동일
```

```c
#include <stdio.h>

typedef int INT;
typedef int* PTR_INT;

typedef unsigned int UINT;
typedef unsigned int* PTR_UINT;
typedef unsigned char UCHAR;
typedef unsigned char* PTR_UCHAR;

int main(void) {
    INT num1 = 120;          // int num1 = 120;
    PTR_INT pnum1 = &num1;   // int * pnum1 = &num1;

    UINT num2 = 190;         // unsigned int num2 = 190;
    PTR_UINT pnum2 = &num2;  // unsigned int * pnum2 = &num2;

    UCHAR ch = 'Z';          // unsigned char ch = 'Z';
    PTR_UCHAR pch = &ch;     // unsigned char * pch = &ch;

    printf("%d, %u, %c ₩n", *pnum1, *pnum2, *pch);

    return 0;
}
```

=> 결과

120, 190, Z

<해석>

typedef 선언에 있어서 새로운 이름의 부여는 가장 마지막에 등장하는 단어를 중심으로 이뤄진다. typedef 선언을 통해 복잡한 유형의 자료형 선언을 매우 간결하게 처리할 수 있다. 또한, typedef로 정의되는 자료형의 이름은 대문자로 시작하는 것이 관례이다. 그래서 기본 자료형의 이름과 typedef로 새로 정의된 자료형의 이름을 구분할 수 있기 때문이다.

- 구조체의 정의와 typedef 선언

구조체 변수의 선언에 있어서 struct 선언을 생략할 수 있다. 사실 모든 구조체의 이름을 대상으로 struct 선언의 생략을 위한 typedef 선언이 등장한다. 그렇기 때문에 구조체의 정의와 typedef의 선언으로 한데 묶을 수도 있고, 또 이렇게 선언하는 것이 일반적이다.

```
typedef struct point {
    int xpos;
    int ypos;
} Point;
```

```c
#include <stdio.h>

struct point {
    int xpos;
    int ypos;
};

typedef struct point Point;

typedef struct person {
    char name[20];
    char phoneNum[20];
    int age;
} Person;

int main(void) {
    Point pos = { 10, 20 };
    Person man = { "이승기", "010-1212-0001", 21 };
    printf("%d %d \n", pos.xpos, pos.ypos);
    printf("%s %s %d \n", man.name, man.phoneNum, man.age);

    return 0;
}
```

=> 결과

10 20

이승기 010-1212-0001 21

<해석>

typedef를 통해 구조체 point와 person을 Point와 Person으로 재정의하였다. 이를 통해 구조체 선언을 생략하고 구조체 변수를 사용할 수 있다. 참고로 typedef 선언이 추가되었다고 해서 struct 선언을 통한 구조체 변수의 선언이 불가능한 것은 아니다. 즉, 구조체가 정의되면, 아래의 두 가지 방식으로 구조체 변수를 선언할 수 있다.

```
typedef struct point {
    int xpos;
    int ypos;
} Point;
```

⇨

```
Point pos1;   // typedef 선언을 이용한 변수 선언 가능
struct point pos2;    // struct 선언을 추가한 형태의
                      변수선언 가능
```

=> 이런 경우에 구조체 변수 point는 생략 가능하다.

- **LAB**

```c
#include <stdio.h>

int main(void) {
    //영화 정보 구조체
    typedef struct movie {
        char* title;  //영화제목
        int attendance;  //관객수
    } movie;

    movie assassination;

    assassination.title = "암살";
    assassination.attendance = 12700000;

    printf("[%s] 관객수 : %d₩n", assassination.title, assassination.attendance);

    return 0;
}
```

=> 결과

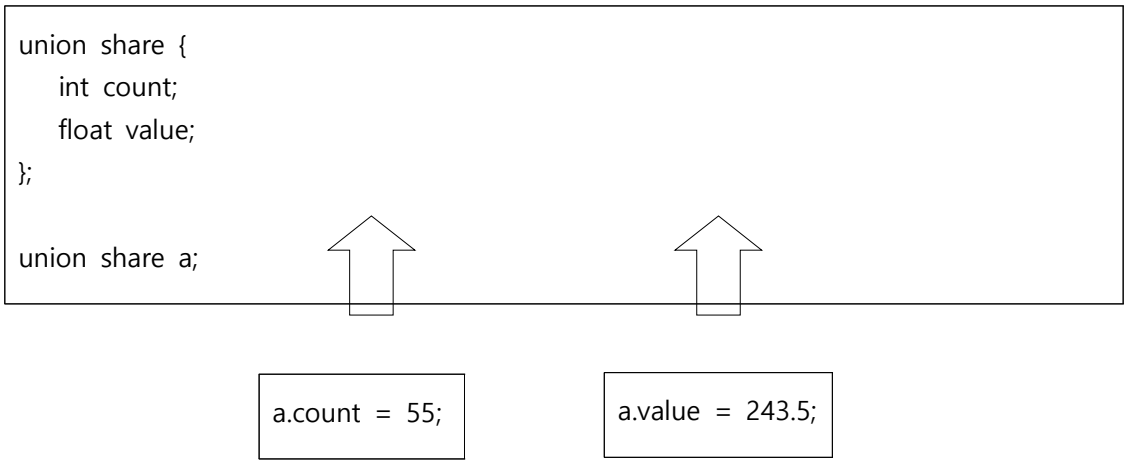
[암살] 관객수 : 12700000

<해석>

구조체 struct movie의 자료형을 movie로 다시 재정의하고, 변수 assassination에 하나의 영화 정보를 저장한 후 출력한다. 여기서 구조체 태그 이름인 movie는 생략가능하다.

- 공용체의 정의와 의미

공용체는 서로 다른 자료형의 값을 동일한 저장공간에 저장하는 자료형으로, 공용체를 구성하는 멤버에 한 번에 한 종류만 저장하고 참조할 수 있다.



=====> count를 먼저 저장하고 value 값을 저장하면 count 값 사라진다. <=====

공용체 변수의 크기는 멤버 중 가장 큰 자료형의 크기로 정해진다. 공용체의 멤버는 모든 멤버가 동일한 저장공간을 사용하므로 동시에 여러 멤버의 값을 동시에 저장하여 이용할 수 없으며, 마지막에 저장된 단 하나의 멤버 자료값만을 저장한다. 공용체의 초기화 값은 공용체 정의 시 처음 선언한 멤버의 초기값으로만 저장이 가능하다. 만일 다른 멤버로 초기값을 지정하면 컴파일 시 경고가 발생한다. 초기값으로 동일한 유형의 다른 변수의 대입도 가능하다. 공용체 변수로 멤버를 접근하기 위해서는 접근 연산자를 사용한다. 구조체와 마찬가지로 공용체도 typedef를 이용하여 새로운 자료형으로 정의할 수 있다.

```
typedef struct sbox { //구조체의 정의
   int mem1;
   int mem2;
   double mem3;
} SBox;
```

```
typedef union ubox { //공용체의 정의
   int mem1;
   int mem2;
   double mem3;
} UBox;
```

=> 구조체와 공용체의 정의방식에서의 유일한 차이점은 struct 선언을 하느냐, union 선언을 하느냐에 있다. 하지만 각각의 변수가 메모리 공간에 할당되는 방식과 접근의 결과에는 많은 차이가 있다. 위의 구조체와 공용체를 대상으로 SBox와 UBox의 크기를 구하면 SBox는 16, UBox는 8이 나온다. 여기서 16은 모든 멤버의 크기를 합한 결과이고, 8은 멤버 중에서 가장 크기가 큰 double의 크기만 계산된 결과이다.

```c
#include <stdio.h>

typedef struct sbox {  //구조체 sbox정의
    int mem1;
    int mem2;
    double mem3;
} SBox;

typedef union ubox {  //공용체 ubox정의
    int mem1;
    int mem2;
    double mem3;
} UBox;

int main(void) {
    SBox sbx;
    UBox ubx;
    printf("%p %p %p ₩n", &sbx.mem1, &sbx.mem2, &sbx.mem3);
    printf("%p %p %p ₩n", &ubx.mem1, &ubx.mem2, &ubx.mem3);
    printf("%d %d ₩n", sizeof(SBox), sizeof(UBox));

    return 0;
}
```

=> 결과

002CFC28 002CFC2C 001CDC30

002CFC18 002CFC18 002CFC18

16 8

<해석>

실행결과에서 가장 주목할 부분은 UBox형 변수를 구성하는 멤버 mem1, mem2, mem3의 주소 값이 동일하다는 것이다. 이는 공용체의 할당 특성의 결과이다.

```
1   #include <stdio.h>
2
3   typedef union ubox {
4       int mem1;
5       int mem2;
6       double mem3;
7   } UBox;
8
9   int main(void) {
10      UBox ubx;  //8바이트 메모리 할당
11      ubx.mem1 = 20;
12      printf("%d ₩n", ubx.mem2);
13      ubx.mem3 = 7.15;
14      printf("%d ₩n", ubx.mem1);
15      printf("%d ₩n", ubx.mem2);
16      printf("%g ₩n", ubx.mem3);
17
18      return 0;
19  }
```

=> 결과

20

-1717986918

-1717986918

7.15

<해석>

11행에서 상위 4바이트의 메모리 공간에 20을 저장한다. 12행에서 mem2는 int형 변수이므로 이 이름으로 접근할 경우 상위 4바이트의 메모리 공간을 참조하게 된다. 그런데 앞서 11행에서 이 공간에 20을 저장했으므로 20이 출력된다. 13행은 mem3에 실수를 저장하고 있다. 결과적으로 11행을 통해 저장된 값을 덮어써버리게 된다. 14, 15행에서는 16행에서 실수를 저장하면서 덮어써버렸기 때문에, 상위 4바이트를 읽어서 출력하면 쓰레기 값이 출력된다. 구조체 변수가 선언되면 ,구조체를 구성하는 멤버는 각각 할당이 된다. 반면 공용체 변수가 선언되면, 공용체를 구성하는 멤버는 각각 할당되지 않고, 그 중 크기가 가장 큰 멤버의 변수만 할당되어 이를 공유하게 된다. 실행결과는 공용체의 멤버들이 메모리 공간을 공유하고 있음을 확인시켜주고 있다.

- **LAB**

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

 //지구 위치 구조체
struct position
{
    double latitude;  //위도
    double longitude;  //경도
};

int main(void) {
    //도시 정보 구조체
    struct city {
        char* name;   //이름
        struct position place;   //위치
    };
    struct city seoul, newyork;
    seoul.name = "서울";
    seoul.place.latitude = 37.33;
    seoul.place.longitude = 126.58;
    newyork.name = "뉴욕";
    newyork.place.latitude = 40.8;
    newyork.place.longitude = 73.9;

    printf("[%s] 위도= %.1f 경도= %.1f₩n", seoul.name, seoul.place.latitude, seoul.place.longitude);
    printf("[%s] 위도= %.1f 경도= %.1f₩n", newyork.name, newyork.place.latitude, newyork.place.longitude);

    return 0;
}
```

=>결과

[서울] 위도 = 37.3 경도 = 126.6

[뉴욕] 위도 = 40.8 경도 = 73.9

13.3 구조체와 공용체의 포인터와 배열

- **구조체 포인터**

포인터는 각각의 자료형 저장 공간의 주소를 저장하듯이 구조체 포인터는 구조체의 주소값을 저장할 수 있는 변수이다. 구조체 포인터 변수의 선언은 일반 포인터 변수 선언과 동일하다.

```c
struct lecture {
   char name[20];
   int type;
   int credit;
   int hours;
};
typedef struct lecture lecture;
lecture *p;
```

- 포인터 변수의 구조체 멤버 접근 연산자 ->

연산식 p->name은 포인터 p가 가리키는 구조체 변수의 멤버 nae을 접근하는 연산식이다. ->는 하나의 연산자이므로 –와 > 사이에 공백이 들어가서는 절대 안된다. 연산식 p->name은 접근연산자(.)와 간접연산자(*)를 사용한 연산식 (*p).name으로도 사용 가능하다. 그러나 (*p).name은 *p.name과는 다르다는 것에 주의해야 한다. 연산식 *p.name은 접근연산자가 간접연산자보다 우선순위가 빠르므로 *(p.name)과 같은 연산식이다. p가 포인터이므로 p.name는 문법오류가 발생한다. 연산자 ->와 .은 우선순위 1이고 결합성은 좌에서 우이며, 연산자 *은 우선순위 2이고 결합성은 우에서 좌이다.

접근 연산식	구조체 변수 os와 구조체 포인터변수 p인 경우의 의미
p->name	포인터 p가 가리키는 구조체의 멤버 name
(*p).name	포인터 p가 가리키는 구조체의 멤버 name
*p.name	*(p.name)이고 p가 포인터이므로 p.name은 문법오류가 발생
*os.name	*(os.name)를 의미하며, 구조체 변수 os의 멤버 포인터 name이 가리키는 변수로, 이 경우는 구조체 변수 os 멤버 강좌명의 첫 문자임, 다만 한글인 경우에는 실행 오류
*p->name	*(p->name)를 의미하며, 포인터 p이 가리키는 구조체 멤버 name이 가리키는 변수로, 이 경우는 구조체 포인터 p이 가리키는 구조체의 멤버 강좌명의 첫 문자임, 마찬가지로 한글인 경우에는 실행 오류

```c
#include <stdio.h>

struct point {
    int xpos;
    int ypos;
};

int main(void) {
    struct point pos1 = { 1,2 };
    struct point pos2 = { 100, 200 };
    struct point* pptr = &pos1;

    (*pptr).xpos += 4;
    (*pptr).ypos += 5;
    printf("[%d, %d] ₩n", pptr->xpos, pptr->ypos);

    pptr = &pos2;
    pptr->xpos += 1;
    pptr->ypos += 2;
    printf("[%d, %d] ₩n", (*pptr).xpos, (*pptr).ypos);

    return 0;
}
```

=> 결과

[5, 7]

[101, 202]

<해석>

11행에서 포인터 변수 pptr이 구조체 변수 pos1을 가리키게 된다. 13~15행에서 현재 pptr이 pos1을 가리키므로, 여기서 진행하는 모든 연산은 pos1을 대상으로 한다. 17행에서는 포인터 변수 pptr이 구조체 변수 pos2를 가리키게 된다. 18~20행에서 현재 pptr이 pos2를 가리키므로, 여기서 진행하는 모든 연산은 pos2를 대상으로 한다.

- **공용체 포인터**

공용체 변수도 포인터 변수 사용이 가능하며, 공용체 포인터 변수로 멤버를 접근하려면 접근연산자 ->를 이용한다. 아래는 공용체 변수 value를 가리키는 포인터 p를 선언하여 p가 가리키는 공용체 멤버 ch에 'a'를 저장하는 소스이다.

```
union data {
  char ch;
  int cnt;
  double real;
} value, *p;

p = &value;  // 포인터 p에 value의 주소값을 저장
p->ch = 'a';  // value.ch = 'a';와 동일한 문장
```

```c
#include <stdio.h>

int main(void) {
    //공용체 union data 정의
    union data {
        char ch;
        int cnt;
        double real;
    };

    //유니온 union data를 다시 자료형 udata로 정의
    typedef union data udata;

    //udata 형으로 value와 포인터 p 선언
    udata value, * p;

    p = &value;
    p->ch = 'a';
    printf("%c %c₩n", p->ch, (*p).ch);
    p->cnt = 100;
    printf("%d ", p->cnt);
    p->real = 3.14;
    printf("%.2f ₩n", p->real);

    return 0;
}
```

=> 결과

a a

100 3.14

<해석>

12행에서 유니온 union data를 다시 자료형 udata로 정의하고 udata 형으로 value와 포인터 p를 선언한다. 17행에서는 포인터 변수 p에 공용체 변수 value이 주소를 저장한다. 18행, p가 가리키는 공용체 변수 value의 멤버 ch를 연산자 ->를 사용하여 p->cnt로 참조, 멤버 cnt에 정수 100을 저장한다. 19행은 p가 가리키는 공용체 변수 value의 멤버 ch를 p-<ch와 (*p).ch로 참조하여 출력하고, 20행에서 p가 가리키는 공용체 변수 value의 멤버 cnt를

연산자 ->를 사용하여 p->cnt로 참조, 멤버 cnt에 정수 100을 저장한다. 21행에서 p가 가리키는 공용체 변수 value의 멤버 cnt를 p->cnt((*p).cnt도 가능)로 참조하여 출력하고, p가 가리키는 공용체 변수 value의 멤버 real을 연산자 ->를 사용하여 p->real로 참조하여 멤버 real에 실수 3.14를 저장한다. 그 후 23행에서 p가 가리키는 공용체 변수 value의 멤버 real을 p->real((*p).real도 가능)로 참조하여 출력한다.

- 구조체 배열

다른 배열과 같이 동일한 구조체 변수가 여러 개 필요하면 구조체 배열을 선언하여 이용할 수 있다. 구조체 배열의 초기값 지정 구문에서는 중괄호가 중첩되게 나타난다. 외부 중괄호는 배열 초기화의 중괄호이고, 내부 중괄호는 배열원소인 구조체 초기화를 위한 중괄호이다.

```
lecture c[] = { {“인간과사회”, 0, 2, 2},
                {“경제학이론”, 1, 3, 3},
                {“자료구조”, 2, 3, 3} };
```

```c
#include <stdio.h>

//유니온 구조체를 정의하면서 변수 data1도 선언한 문장
struct lecture {
    char name[20];  //강좌명
    int type;  //강좌구분
    int credit;  //학점
    int hours;  //시수
};
typedef struct lecture lecture;

char* lectype[] = { "교양", "일반선택", "전공필수", "전공선택" };
char* head[] = { "강좌명", "강좌구분", "학점", "시수" };

int main(void) {
    //구조체 lecture의 배열 선언 및 초기화
    lecture course[] = { {"인간과 사회", 0, 2, 2}, {"경제학개론", 1, 3, 3}, {"자료구조", 2, 3, 3} };

    int arysize = sizeof(course) / sizeof(course[0]);

    printf("배열크기 : %d₩n₩n", arysize);
    printf("%12s   %12s %6s %8s₩n", head[0], head[1], head[2], head[3]);
    printf("=============================================");
    for (int i = 0; i < arysize; i++)
        printf("%16s %10s %5d %5d₩n", course[i].name, lectype[course[i].type], course[i].credit, course[i].hours);

    return 0;
}
```

=> 결과

배열크기 : 5

강좌명 강좌구분 학점 시수

==========

인간과 사회 교양 2 2

경제학개론 일반선택 3 3

자료구조 전공필수 3 3

<해석>

구조체 struct이 배열을 선언하면서 바로 초기값을 대입하였다. 배열 크기를 계산하여 변수 arysize에 저장하고 배열 크기와 제목을 출력한다. 24행에서 첨자 I는 0에서 arysize-1까지 실행하고, 24, 25행에서 구조체 struct의 모든 배열 원소를 각각 출력한다.

-LAB

```
1   #define _CRT_SECIRE_NO_WARNINGS
2   #include <stdio.h>
3   #include <string.h>
4
5   int main(void) {
6       //영화 정보 구조체
7       typedef struct movie {
8           char* title;  //영화제목
9           int attendance;  //관객수
10          char director[20];  //감독
11      } movie;
12
13      movie box[] = {
14          {"명량", 17613000, "김한민"}, {"국제시장", 14257000, "윤제균"}, {"베테랑", 13383000}
15      };
16
17      //영화 베테랑의 감독을 류승완으로 저장
18      stycpy(box[2].director, "류승관");
19
20      printf("   제목   감독  관객수₩n");
21      printf("=========================₩n");
22      for (int i = 0; i < 3; i++) {
23          printf("[%8s] %6s %d₩n", box[i].title, box[i].director, box[i].attendance);
24      }
25
26      return 0;
27  }
```

=> 결과

제목 감독 관객수

==========

[명량] 김한민 17613000

[국제시장] 윤제균 14257000

[베테랑] 류승완 13383000

<해석>

영화 제목, 감독, 관객수를 표현하는 구조체 struct movie는 멤버로 title, attendace, director로 구성되어 있다. 구조체 movie의 배열을 선언하고, 영화 세 편의 정보를 저장한다. for문을 통해 각각의 영화 정보를 출력한다.

14.1 함수의 인자전달 방식

- 값에 의한 호출

C언어는 함수의 인자 전달 방식이 기본적으로 값에 의한 호출 방식이다. 값에 의한 호출 방식이란 함수 호출 시 실인자의 값이 형식인자에 복사되어 저장된다는 의미이다. 값에 의한 호출 방식을 사용해서는 함수 외부의 변수를 함수 내부에서 수정할 수 없다는 특징이 있다.

```c
#include <stdio.h>

void increase(int origin, int increment);

int main(void) {
    int amount = 10;
    //amount가 20 증가하지 않음
    increase(amount, 20);
    print("%d₩n", amount);

    return 0;
}

void increase(int origin, int increment) {
    origin += increment;
}
```

=> 결과

10

<해석>

지역변수 amount를 선언하면서 10을 저장하고, 함수 호출을 increase(amount, 10)로 하여 함수 increase()에서 변수 origin에 10이, 변수 increment에 20이 각각 저장된다. amount를 출력하면 그대로 10이 출력된다. 매개변수인 origin에 매개변수인 increment의 값을 더하여 저장하면 origin은 20이 증가하나 main의 변수에는 영향을 미치지 않으므로 10이 출력된다.

- 참조에 의한 호출

C언어에서 포인터를 매개변수로 사용하면 함수로 전달된 실인자의 주소를 이용하여 그 변수를 참조할 수 있다. 이와 같이 함수에서 주소의 호출을 참조에 의한 호출이라 한다.

```c
#include <stdio.h>

void increase(int *origin, int increment);

int main(void) {
    int amount = 10;
    //&amount: amount의 주소로 호출
    increase(&amount, 20);
    print("%d₩n", amount);

    return 0;
}

void increase(int *origin, int increment) {
    //*origin은 origin이 가리키는 변수 자체
    *origin += increment;  //그러므로 origin이 가리키는 변수값이 20 증가
}
```

=> 결과

30

<해석>

지역변수 amount를 선언하면서 10을 저장하고, 함수 호출을 increase(&amount, 20)로 하여 함수 increase()에서 포인터 변수인 origin에 amount의 주소를 저장하고, 변수 increment에 20이 각각 저장된다. amount를 출력하면 20이 증가하여 30이 출력된다. 매개변수인 origin에서 *origin를 참조하여 main의 변수 amount를 참조하므로 매개변수인 increment의 값을 더하여 amount에 저장한다. 그러므로 변수 amount는 20이 증가하여 main에서 30이 출력된다.

- 배열이름으로 전달

함수의 매개변수로 배열을 전달하는 것은 배열의 첫 원소를 참조 매개변수로 전달하는 것과 동일하다. 함수 내부에서 실인자로 전달된 배열의 배열크기는 알 수 없다. 그 이유는 매개변수를 double ary[]처럼 배열형태로 기술해도 단순히 double *ary처럼 포인터 변수로 인식하기 때문이다. 그러므로 배열 크기를 두 번째 인자로 사용한다.

```c
#include <stdio.h>

#define ARYSIZE 5
double sum(double g[], int n);  //배열원소 값을 모두 더하는 함수원형

int main(void) {
    //배열 초기화
    double data[] = { 2.3, 3.4, 4.5, 6.7, 9.2 };

    //배열원소 출력
    for (int i = 0; i < ARYSIZE; i++) {
        printf("%5.1f", data[i]);
    }
    puts("");

    //배열 원소값을 모두 더하는 함수호출
    printf("합 : %5.1f₩n", sum(data, ARYSIZE));

    return 0;
}
//배열 원소값을 모두 더하는 함수정의
double sum(double ary[], int n) {
    double total = 0.0;
    for (int i = 0; i < n; i++) {
        total += ary[i];
    }

    return total;
}
```

=> 결과

2.3 3.4 4.5 6.7 9.2

합 : 26.1

- 다양한 배열원소 참조 방법

int *address = point; 라는 문장이 있다면, sum += *(address++);로 배열을 구할 수 있고, sum += *(point + i); 로도 가능하다. 하지만 sum += *(point++)는 사용할 수 없다. 증가 연산식 point++의 피연산자로 상수인 point를 사용할 수 없기 때문이다.

int ary[]	int *ary
sum += ary[i]	sum += *(ary + i)
sum += *ary++	sum += *(ary++)

=> 왼쪽과 오른쪽은 같은 의미

- 배열크기 계산방법

- 배열의 전체 원소 수 : sizeof(x) / sizeof(x[0][0])
- 배열의 행 수 : sizeof(x) / sizeof(x[0])
- 배열의 열 수 : sizeof(x[0]) / sizeof(x[0][0])

- 가변 인자

가변 인자란 함수에서 처음 또는 앞 부분의 매개변수는 정해져 있으나 이후 매개변수 수와 각각의 자료형이 고정적이지 않고 변하는 인자를 말한다. 함수의 매개변수에서 중간 이후부터 마지막에 위치한 가변 인자만 가능하고, 헤더파일 stdarg.h가 필요하다.

<가변 인자 구현하기>

1. 가변인자 선언 : va_list argp;
2. 가변인자 처리 시작 : va_start(va_list argp, prevarg)
3. 가변인자 얻기 : type va_arg(va_list argp, type)
4. 가변인자 처리 종료 : va_end(va_list argp)

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

double avg(int count , ...);    //int count 이후는 가변인자 ...

int main(void) {
    printf("평균 %.2f₩n", avg(5, 1.2, 2.1, 3.6, 4.3, 5.8));

    return 0;
}
//가변인자 ... 시작 전 첫 고정 매개변수는 이후의 가변인자를 처리하는데 필요한 정보를 지정
//여기서는 가변인자의 수를 지정
double avg(int numagrs, ...) {
    //가변인자 변수 선언
    va_list argp;
    //numargs 이후의 가변인자 처리 시작
    va_start(argp, numagrs);

    double total = 0;  //합이 저장될 변수
    for (int i = 0; i < numagrs; i++) {
        //지정하는 double 형으로 가변인자 하나 하나를 반환
        total += va_arg(argp, double);
    }
        //가변인자 처리 종료
        va_end(argp);

    return total / numagrs;
}
```

=> 결과

평균 3.40

<해석>

가변인자 처리 함수 avg(5, 1.2, 2.1, 3.6, 4.3, 5,8)를 호출하여 평균을 출력하는데 실인자에서 5는 처리할 가변인자의 수 5이며, 나머지는 5개의 double형 기변인자의 실인자이다. 그렇기 때문에 5를 제외한 실인자들의 평균값을 구한 것이다.

14.2 포인터 전달과 반환

- 주소값 반환

함수의 결과를 포인터로 반환할 때 반환값인 포인터가 가리키는 변수를 바로 참조할 수 있다. 하지만 지역변수의 반환은 문제를 발생시킬 수 있다.

```c
int *add(int *psum, int a, int b)
{
    *psum = a + b;
    return psum;
}
```

```c
#include <stdio.h>

int* add(int*, int, int);
int* subtract(int*, int, int);
int* multiply(int, int);

int main(void) {
    int m = 0, n = 0, sum = 0, diff = 0;

    printf("두 정수 입력: ");
    printf("%d %d", &m, &n);

    printf("두 정수 합: %d₩n", *add(&sum, m, n));
    printf("두 정수 차: %d₩n", *subtract(&diff, m, n));
    printf("두 정수 곱: %d₩n", *multiply(m, n));

    return 0;
}
int* add(int* psum, int a, int b) {
    *psum = a + b;
    return psum;
}
int* subtract(int* pdiff, int a, int b) {
    *pdiff = a - b;
    return pdiff;
}
int* multiply(int a, int b) {
    int mult = a * b;
    return &mult;
}
```

=> 결과

두 정수 입력 : 6 9

두 정수 합 : 15

두 정수 차 : -3

두 정수 곱 : 54

- 상수를 위한 키워드 const

포인터를 매개변수로 이용하면 수정된 결과를 받을 수 있어 편리하지만 이러한 참조에 의한 호출은 매개변수가 가리키는 변수값이 원하지 않는 값으로 수정될 수 있다. 이럴 때 수정을 원하지 않는 함수의 인자 앞에 키워드 const를 삽입하여 참조되는 변수가 수정될 수 없게 한다.

```
[
// 매개변수 포인터 a, b가 가리키는 변수의 내용은 수정하지 못함
void multiply(double *result, const double *a, const double *b)
{
      *result = *a * *b;
      //다음 2문장 오류 발생
      *a = *a + 1;
      *b = *b + 1;
}
```

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

void multiply(double*, const double*, const double*);
void devideandincrement(double*, double*, double*);

int main(void) {
    double m = 0, n = 0, mult = 0, dev = 0;

    printf("두 실수 입력: ");
    scanf("%lf %lf", &m, &n);
    multiply(&mult, &m, &n);
    devideandincrement(&dev, &m, &n);
    printf("두 실수 곱: %.2f, 나눔: %.2f₩n", mult, dev);
    printf("연산 후 두 실수: %.2f %.2f₩n", m, n);

    return 0;
}
//매개변수 포인터 a, b가 가리키는 변수의 내용을 곱해 result가 가리키는 변수에 저장
//매개변수 포인터 a, b가 가리키는 변수의 내용은 수정하지 못함
void multiply(double* result, const double* a, const double* b) {
    *result = *a * *b;
}
//매개변수 포인터 a, b가 가리키는 변수의 내용을 나누어 result가 가리키는 변수에 저장한 후
//a, b가 가리키는 변수의 내용을 모두 1 증가시킴
void devideandincrement(double* result, double* a, double* b) {
    *result = *a / *b;
    ++ *a;  //++(*a)와 같음
    (*b)++;  //*b++과 다름
}
```

=> 결과

두 실수 입력 : 10.5 2.5

두 실수 곱 : 26.25, 나눔 : 4.20

연산 후 두 실수 : 11.50 3.50

<해석>

const의 유무에 따라 변수 a와 b의 수정 가능 여부가 달라진다. 따라서 devideandincrement에서 만약 const double *a, const double *b였다면 ++*a와 (*b)++에서 오류가 발생한다.

14.3 함수 포인터와 void 포인터

- 함수 포인터

포인터의 장점은 다른 변수를 참조하여 읽거나 쓰는 것도 가능하다는 것이다. 이처럼 하나의 함수 이름으로 필요에 따라 여러 함수를 사용하면 편리할 수 있다. 함수 포인터는 함수의 주소값을 저장하는 포인터 변수이다. 즉 함수 포인터는 함수를 가리키는 포인터를 말한다. 함수도 변수처럼 메모리 어딘가에 저장되어 있으며 그 주소를 갖고 있다. 함수 포인터는

반환형, 인자목록의 수와 각각의 자료형이 일치하는 함수의 주소를 저장할 수 있는 변수이다. 함수 포인터를 선언하려면 함수원형에서 함수이름을 제외한 반환형과 인자목록의 정보가 필요하다.

```
void (*pf) (double *z, double x. double y);
pf = add;
pf = &add;
pf = subtract;
pf = &subtract;
```

인자목록이 add()와 동일하다면 subtract()도 가리킬 수 있다. 하지만 문장 pf = subtract;와 같이 함수 포인터에는 괄호가 없이 함수이름만으로 대입해야 하며, &add나 &subract로도 사용 가능하다.

- 함수 포인터 배열

함수 포인터 배열은 함수 포인터가 원소인 배열이다.

```
void (*fpary[4]) (double *, double. double);
fpary[0] = add;
fpary[1] = subtract;
fpary[2] = multiply;
fpary[3] = devide;


void (*fpary[4]) (double *, double. double) = {add, subract, multiply, devide};
```

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

void multiply(double*, double, double);
void devide(double*, double, double);

int main(void) {
    char op[4] = { '*', '/' };
    //함수 포인터 선언과 동시에 초기화
    void(*fpary[4])(double*, double, double) = { multiply, devide };

    double m, n, result;
    printf("사칙연산을 수행할 실수 2개를 입력하세요, >> ");
    scanf("%lf %lf", &m, &n);
    //사칙연산을 배열의 첨자를 이용하여 수행
    for (int i = 0; i < 4; i++) {
        fpary[i](&result, m, n);
        printf("%.2lf %c %.2lf₩n", m, op[i], n, result);
    }

    return 0;
}
//x * y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void multiply(double* z, double x, double y) {
    *z = x * y;
}
//x / y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void devide(double* z, double x, double y) {
    *z = x / y;
}
```

=> 결과

사칙연산을 수행할 실수 2개를 입력하세요. >> 3.87 6.93

3.87 * 6.93 == 26.82

3.87 / 6.93 == 0.56

<해석>

각각의 포인터는 (void) 없으며, 인자의 유형은 double *, double, double 인 하무의 주소를 저장할 수 있다. 제어변수 I에 따라 fpary[0]는 multiply()를 호출하고, fpary[1]은 devide를 각각 호출한다. 호출할 때 인자는 모두 &result, m, n으로 동일하며 2가지 연산에 따라 출력된다.

- **void 포인터**

void 포인터는 자료형을 무시하고 주소값만을 다루는 포인터이다. 그러므로 포인터 void 포인터는 대상에 상관없이 모든 자료형의 주소를 저장할 수 있는 만능 포인터로 사용할 수 있다. void 포인터에는 일반변수 포인터는 물론 배열과 구조체 심지어 함수 주소도 담을 수 있다.

```
char ch = 'A';
int data = 5;
double value = 34.76;
void *vp;                 //void 포인터 변수 vp 선언
vp = &ch;                 //ch의 주소만 저장
vp = &data;               //data의 주소만 저장
vp = &value;              //value의 주소만 저장
```

void 포인터는 모든 주소를 저장할 수 있지만 가리키는 변수를 참조하거나 수정이 불가능하다. 주소값으로 변수를 참조하려면 결국 자료형으로 참조범위를 알아야하는데 void 포인터는 이러한 정보가 전혀 없이 주소만을 담는 변수에 불과하기 때문이다. 그러므로 실제 void 포인터로 변수를 참조하기 위해서는 자료형 변환이 필요하다.

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

void myprint(void) {
    printf("void 포인터, 신기하네요!₩n");
}

int main(void) {
    int m = 10;
    double d = 3.98;
    char str[][20] = { {"C 언어, "}, {"재미있네요!"} };

    void* p = &m;
    printf("p 참조 정수 : %d₩n", *(int*)p); //int *로 변환

    p = &d;
    printf("p 참조 실수 : %.2f₩n", *(double*)p);  //double *로 변환

    p = myprint;
    printf("p 참조 함수 실행 : ");
    ((void(*)(void)) p)();  //함수 포인터인 void(*)(void)로 변환하여 호출

    return 0;
}
```

=> 결과

p 참조 정수 : 10

p 참조 실수 : 3.98

p 참조 함수 실행 : void 포인터, 신기하네요!

15.1 파일 기초

- 파일의 필요성

변수와 같이 프로그램에서 내부에서 할당되어 사용되는 주기억장치의 메모리 공간은 프로그램이 종료되면 모두 사라진다. 하지만 보조기억장치인 디스크에 저장되는 파일은 직접 삭제하지 않은 한 프로그램이 종료되더라도 계속 저장할 수 있다. 그러므로 프로그램에서 사용하던 정보를 종료 후에도 계속 사용하고 싶다면 프로그램에서 파일에 그 내용을 저장해야 한다.

- 텍스트 파일과 이진 파일

파일은 보조기억장치의 정보저장 단위로 자료의 집합이다. 파일은 텍스트 파일과 이진 파일로 나뉘는데 대표적인 텍스트 파일은 메모장 같은 편집기로 작성된 파일이며, 이진 파일은 실행 파일과 그림 파일, 음악 파일, 동양상 파일 등을 예로 들 수 있다. 텍스트 파일은 문자 기반의 파일로서 내용이 아스키 코드와 같은 문자 코드값으로 저장된다. 메모리에 저장된 실수와 정수와 같은 내용도 텍스트 파일에 저장될 때는 문자 형식으로 변환되어 저장된다. 그러므로 텍스트 파일은 텍스트 편집기를 통하여 그 내용을 볼 수 있고 수정할 수도 있다. 이진 파일은 그림 파일, 동영상 파일, 실행 파일과 같이 각각의 목적에 알맞은 자료가 이진 형태로 저장되는 파일이다. 이진 파일은 컴퓨터 내부 형식으로 저장되는 파일이며, 자료는 메모리 자료 내용에서 어떤 변환도 거치지 않고 그대로 파일에 기록된다. 그러므로 입출력 속도도 텍스트 파일에 비해 빠르다. 이러한 이진 파일은 텍스트 편집기로는 그 내용을 볼 수 없고, 그 내용을 이미 알고 있는 특정한 프로그램에 의해 인지될 때 의미가 있다.

- 입출력 스트림

자료의 입력과 출력은 자료의 이동이라고 볼 수 있고, 자료가 이동하려면 이동 경로가 필요하다. 입출력 시 이동 통로가 입출력 스트림이며, 키보드에서 프로그램으로 자료가 이동하는 경로가 바로 표준입력 스트림이고, 프로그램에서 모니터의 콘솔로 자료가 이동하는 경로가 표준출력 스트림이다. 함수 scanf()는 표준 입력 스트림에서 자료를 읽을 수 있는 함수이고, pinrtf()는 바로 표준출력 스트림으로 자료를 보낼 수 있는 함수이다. 다른 곳에서 프로그램으로 들어오는 경로가 입력스트림이며, 자료가 떠나는 시작 부분이 자료 원천부이다. 반대로 프로그램에서 다른 곳으로 나가는 경로가 출력 스트림이며, 자료의 도착 장소가 자료 목적부이다.

- 파일 스트림 이해

프로그램에서 보조기억장치에 파일로 정보를 저장하거나 파일에서 정보를 참조하려면 파일에 대한 파일 스트림을 먼저 연결해야 한다. 파일 스트림이란 보조기억장치의 파일과 프로그램을 연결하는 전송경로이다. 파일 스트림은 입력을 위한 파일 입력 스트림과 출력을 위한 출력 스트림으로 나뉠 수 있다.

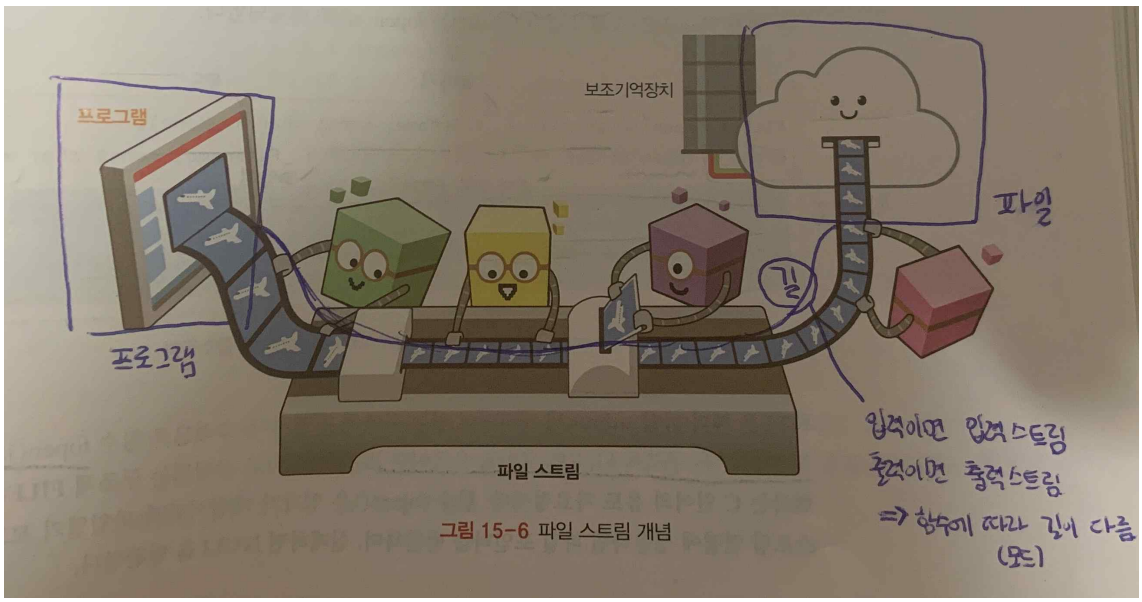


그림 15-6 파일 스트림 개념

- 함수 fopen()으로 파일 스트림 열기

프로그램에서 특정한 파일과 파일 스트림을 연결하기 위해서는 함수 fopen() 또는 fopen_s()를 이용하며 헤더 파일 stdio.h에 정의되어 있다. FILE은 헤더 파일 stdio.h에 정의되어 있는 구조체 유형이다. 그러므로 함수 fopen()의 반환값 유형 FILE *은 구조체 FILE의 포인터 유형이다. 함수 fopen()은 인자가 파일이름과 파일열기 모드이며, 파일 스트림 연결에 성공하면 파일 포인터를 반환하며, 실패하면 NULL을 반환한다.

```
FILE * fopen(const char * _Filename, const char * _Mode);
```

인자인 파일열기 종류에는 텍스트 파일인 경우 "r", "w", "a" 등의 종류가 있다. 읽기모드 r은 읽기가 가능한 모드이며, 쓰기는 불가능하다. 반대로 쓰기모드 w는 파일 어디에든 쓰기가 가능한 모드이나 읽기는 불가능하다. 추가모드 a는 파일 중간에 쓸 수 없으며 파일 마지막에 추가적으로 쓰는 것만 가능한 모드이며, 읽기는 불가능하다. 그러므로 파일모드 a에서 파일에 쓰는 내용은 무조건 파일 마지막에 추가된다.

- 함수 fopen()으로 파일 스트림 열기

함수 fclose()는 fopen()으로 연결한 파일 스트림을 닫는 기능을 수행한다. 그러므로 파일 스트림을 연결한 후 파일 처리가 모두 끝났으면 파일 포인터 f를 인자로 함수 fclose()를 호출하여 반드시 파일을 닫도록 한다.

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

int main(void) {
    char *fname = "basic.txt"; //파일 이름
    FILE* f;  //파일 포인터

    //파일에 쓰려는 자료
    char name[30] = "강미정";
    int point = 99;

    //파일 열기 함수 fopen()과 fopen_s()
    if ((f = fopen(fname, "w")) == NULL) {
    //if (fopen_s(&f, fname, "w") != 0)
        printf("파일이 열리지 않습니다.₩n");
        exit(1);
    };

    //파일 "basic.txt"에 쓰기
    fprintf(f, "이름이 %s인 학생의 성적은 %d입니다.₩n", name, point);
    fclose(f);
    //표준출력 콘솔에 쓰기
    printf("이름이 %s인 학생의 성적은 %d입니다.₩n", name, point);
    puts("프로젝트 폴더에서 파일 basic.txt를 메모장으로 열어 보세요.");

    return 0;
}
```

=> 결과

이름이 강미정인 학생의 성적은 99입니다.

프로젝트 폴더에서 파일 basic.txt.를 메모장으로 열어 보세요.

15.2 텍스트 파일 입출력

- 함수 fprintf()와 fscanf()

텍스트 파일에 자료를 쓰거나 읽기 위하여 함수 fprintf()와 fscanf() 또는 fscanf_s()를 이용한다. 이 함수를 이용하기 위해서는 헤더 파일 stdio.h를 포함해야 한다.

```
int fprintf(FILE * _File, const char * _Format, ...);
int fscanf(FILE * _File, const char * _Format, ...);
int fscanf_s(FILE * _File, const char * _Format, ...);
```

첫 번째 인자는 입출력에 이용될 파일이고, 두 번째 인자는 입출력에 이용되는 제어 문자열이며, 다음 인자들은 입출력될 변수 또는 상수 목록이다. 함수 원형에서 기호 …은 인자 수가 정해지지 않은 다중 인자를 뜻한다. 또한, 함수 fprintf()와 fscanf()의 첫 번째 인자에 각각 stdin 또는 stdout을 이용하면 표준 입력, 표준 출력으로 이용이 가능하다.

- 표준입력 자료를 파일에 쓰기

```
scanf_s("%s%d%d", name, 30, &point1, &point2);
=> 문자열이 저장되는 name과 그 크기를 지정해야 한다.

fprintf(f, "%d %s %d %d\n", ++cnt, name, point1, point2);
```

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

int main(void) {
    char fname[] = "grade.txt"; //파일 이름
    FILE* f;  //파일 포인터

    //파일에 쓰려는 자료
    char name[30];
    int point1, point2, cnt = 0;

    //파일 열기 함수 fopen()과 fopen_s()
    if ((fopen_s(&f, fname, "w")) != 0) {
        printf("파일이 열리지 않습니다.₩n");
        exit(1);
    };
    printf("이름과 성적(중간, 기말)을 입력하세요.₩n");
    scanf("%s %d %d", name, &point1, &point2);
    //파일 "grade.txt"에 읽기
    fscanf(f, "%d %s %d %d₩n", &cnt, name, &point1, &point2);
    fprintf(stdout, "₩n%6s%16s%10s%8s₩n", "번호", "이름", "중간", "기말");
    fprintf(stdout, "%5d%18s%8d%8d₩n", cnt, name, point1, point2);
    fclose(f);

    return 0;
```

=>결과

이름가 성적(중간, 기말)을 입력하세요.

김소현 90 98

```
번호      이름    중간    기말
  1       김소현   90      98
```

- 함수 fgets()와 fputs()

함수 fgets()는 파일로부터 한 행의 문자열을 입력받는 함수이다. 함수 fputs()는 파일로 한 행의 문자열을 출력하는 함수이다.

```
char * fgets(char * _Buf, int _MaxCount, FILE * _File);
int fputs(char * _Buf, FILE * _File);

char names[80]
FILE * f;
fgets(names, 80, f);
fputs(names, f);
```

함수 fgets()에서 첫 번째 인자는 문자열이 저장될 문자 포인터이고, 두 번째 인자는 입력할 문자의 최대 수이며, 세 번째 인자는 입력 문자열이 저장될 파일이다. 함수 fputs()에서 첫 번째 인자는 출력될 문자열이 저장된 문자 포인터이고, 두 번째 인자는 문자열이 출력되는 파일이다. fgets()는 파일로부터 문자열을 개행문자(₩n)까지 읽어 마지막 개행문자를 '₩0'문자로 바꾸어 입력 버퍼 문자열에 저장한다. 마찬가지로 함수 fputs()는 문자열을 한 행에 출력한다.

- 함수 feof()와 ferror()

함수 feof()는 파일 스트임의 EOF(End Of File) 표시를 검사하는 함수이고, 함수 ferror()는 파일 처리에서 오류가 발생했는지 검사하는 함수이다. 함수 ferror()는 이전 파일 처리에서 오류가 발생하면 0이 아닌 값을 반환하고, 오류가 발생하지 않으면 0을 반환한다. 함수 feof()는 읽기 작업이 파일의 이전 부분을 읽으면 0을 반환하고 그렇지 않으면 0이 아닌 값을 반환한다.

```
int feof(FILE * _File);
int ferror(FILE * _File);

while (!feof(stdin)) {
    ....
```

```
    fgets(name, 80, stdin);    //표준입력
}
```

여러 줄의 표준입력을 처리하기 위하여 while(!feof(stdin)){....} 구문을 이용한다. 함수 fputs()를 이용하기 전에 함수 fprintf()를 이용하여 줄 번호를 출력한다. 즉 파일 grade.txt. 저장 시 맨 앞에 1부터 순차적으로 번호가 삽입되도록 한다. 표준입력에서 입력을 종룧려면 파일의 끝(EOF)을 의미하는 키 ctrl+Z를 새로운 행의 처음에 누른다.

- 함수 fgetc()와 fputc()

함수 fgetc()와 getc()는 파일로부터 문자 하나를 입력받는 함수이다. 함수 fputc()와 putc()는 문자 하나를 파일로 출력하는 함수이다. 이 함수들은 문자 하나의 입출력의 대상인 파일 포인터를 인자로 이용한다.

```
int fgetc(FILE * _File);
int fputc(int _Ch, File * _File);

int getc(FILE * _File);
int putc(int _Ch, FILE * _File);
```

15.3 이진 파일 입출력

- 함수 fprintf()와 fscanf_s()

함수 fprintf()와 scanf(), fscanf_s()는 자료의 입출력을 텍스트 모드로 처리한다. 함수 fprintf()에 의해 출력된 텍스트 파일은 텍스트 편집기로 그 내용을 볼 수 있으며, 텍스트 파일의 내용은 모두 지정된 아스키 코드와 같은 문자 코드값을 갖고 있어 그 내용을 확인할 수 있을 뿐만 아니라 인쇄할 수 있다.

- 함수 fwrite()와 fread()

텍스트 파일과는 다르게 이진 파일은 C 언어의 자료형을 모두 유지하면서 바이트 단위로 저장되는 파일이다. 이진 모드로 블록 단위 입출력을 처리하려면 함수 fwrite()와 fread()를

이용한다.

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE * f);
size_t fread(void *dstbuf, size_t size, size_t n, FILE * f);

int cnt = 10;
fwrite(&cnt, sizeof(int), 1, f);
fread(&cnt, sizeof(int), 1, f);
```

함수 fwrite()에서 첫 번째 인자 ptr은 출력될 자료의 주소값이며, 두 번째 인자 size는 출력될 자료 항목의 바이트 크기이고, 세 번째 인자는 출력될 항목의 개수이며, 마지막 인자는 출력될 파일 포인터이다. 함수 fwrite()는 파일 f에 ptr에서 시작해서 size*n 바이트만큼의 자료를 출력하며, 반환값은 출력된 항목의 개수이다. 첫 번째 인자의 자료형 void *는 모든 자료형의 포인터를 대신할 수 있는 포인터이다. 이진 파일에 저장되어 있는 자료를 입력하는 함수 fread()는 출력 함수 fwrite()와 인자는 동일하다. 함수 fwrite()는 바이트 단위로 원하는 블록을 파일에 출력하기 위한 함수이고, 함수 fwrite()에 의하여 출력된 자료는 함수 fread()로 입력해야 그 자료유형을 유지할 수 있다.

모드	의미
rb	이진파일의 읽기 모드로 파일을 연다.
wb	이진파일의 쓰기 모드로 파일을 연다.
ab	이진파일의 추가 모드로 파일을 연다.

- 구조체의 파일 입출력

구조체는 멤버를 가지며, 이 구조체 자료형으로 한 멤버의 정보를 저장하기 위해 표준입력으로 입력받는다. 표준입력은 멤버별로 한 행씩 입력받도록 한다. 표준 입력은 한 행마다 fgets()를 이용하여 하나의 문자열로 받고, 이 입력된 문자열에서 각 구조체의 멤버 자료를 추출한다. 문자열에서 자료를 추출하기 우하여 함수 sscanf()를 이용한다. 파일에서 구조체를 읽기 위해서는 함수 feof()을 이용하여 파일의 마지막까지 구조체 자료를 읽어 적당한 출력형태가 되도록 함수 fprintf()와 printf()를 이용하여 출력한다. 이를 실행하려면 생성된 파일 bin 파일을 반드시 프로젝트 하부 폴더에 복사해야 한다.

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

//구조체 자료형 student 정의
typedef struct student {
    char dept[40];  //학과
    char name[20];  //이름
    int snum;  //학번
} student;

int main(void) {
    student mylab[] = {
        { "컴퓨터정보공학과", "김하늘", 201698657}, { "컴퓨터정보공학과", "백규정", 201648762}, { "컴퓨터소프트웨어공학과", "김효주", 201665287} };

    FILE* f;
    char fname[] = "student.bin";
    fopen_s(&f, fname, "wb");
    int size = sizeof(mylab) / sizeof(student);
    fwrite(mylab, sizeof(student), size, f);
    fclose(f);

    //다시 읽기 위해 오픈
    fopen_s(&f, fname, "rb");
    //파일에서 구조체 배열 모두를 한번에 읽어 다시 저장된 배열을 출력
    student lab[10];  //다시 파일의 내용을 저장할 배열 선언
    //파일 f에서 sizeof(student) 크기로 size 수만큼 읽어 lab에 저장
    fread(lab, sizeof(student), size, f);
    for (int i = 0; i < size; i++)
        fprintf(stdout, "%24s%10s%12d\n", lab[i].dept, lab[i].name, lab[i].snum);
    fclose(f);

    return 0;
}
```

=> 결과

컴퓨터정보공학과 김하늘 201698657

컴퓨터정보공학과 백규정 201648762

컴퓨터소프트웨어공학과 김효주 201665287

<해석>

함수 fwrite()의 첫 번째 인자 ptr은 출력될 자료의 주소값이며, 두 번째 인자 size는 출력될 자료 항목의 바이트 크기이고, 세 번째 인자는 출력될 항목의 개수이며, 마지막 인자는 출력될 파일 포인터이다. 함수 fread()의 첫 번째 인자는 저장될 버퍼 자료의 주소값이며, 두 번째 인자 size는 입력될 자료 항목의 바이트 크기이고, 세 번째 인자는 입력될 항목의 개수이며, 마지막 인자는 입력 파일 포인터이다.

마무리글...

수업은 들었지만, 복습을 하지 않아 모르는 것들이 계속 쌓였었는데 책을 다시 읽으며 정리하는 그 과정부터 도움이 된 것 같다.

또한, 정리에 그치지 않고, 정리한 것을 다시 읽으면서 밑줄을 침으로써 중요한 부분들이 더 쉽게 눈에 띄어서 더 많이 읽으며 복습할 수 있도록 했다.

물론 중간고사 대체 과제이긴 하지만 과제라는 생각은 잊고, 책의 내용들을 정리하고, 코딩하며 학습하는 데에 집중했던 것 같다.

수업 교재인 Perfect C뿐만 아니라 열혈 C 프로그래밍과 유튜브, 블로그 등을 통해 다양한 자료들을 통해 조사하고, 참고하여 정리하였다.

수업시간에 다루지 않은 새로운 코드를 작성하고 코딩하며 더 나아간 학습을 하였다.

수업시간에 한 번이라도 다뤘다면 어렴풋이 기억에 남아 답만 도출해내는 상황이 발생하기 때문이다.

이런 상황들은 내가 이 부분을 이해했다고 착각하기 쉽기 때문에 다른 코드들을 가져와 코딩했다.

남은 시간동안 이해가 안 된 부분은 반복적으로 보며, 이해하고 다음 부분으로 넘어갈 수 있도록 할 것이다.

이번 포트폴리오를 계기로 머릿 속에 엉켜있던 내용들을 조금이나마 정리할 수 있었다.

참고서적 : Perfect C, 열혈 C 프로그래밍