C Language Portfolio

C 애플리케이션 구현 중간고사 과제

컴퓨터정보공학과 2학년 E반
20191789 윤한조

1. 문자와 문자열
2. 변수 유효범위
3. 구조체와 공용체
4. 함수와 포인터 활용
5. 파일 처리

1 문자와 문자열

1. 문자와 문자열

    1) 문자와 문자열 선언

    2) 함수 printf()를 사용한 문자와 문자열 출력

    3) 다양한 문자 입출력

    4) 문자열 입력

2. 문자열 관련 함수

    1) 문자 배열 라이브러리와 문자열 비교

    2) 문자열 복사와 연결

    3) 문자열 분리 및 다양한 문자열 관련 함수

3. 여러 문자열 처리

    1) 문자 포인터 배열과 이차원 문자 배열

    2) 명령행 인자

1. 문자와 문자열

1) 문자와 문자열 선언

[문자와 문자열의 개념]

문자	문자열
‘ ’ 표기	“ ” 표기
‘@’ , ‘씨’ , ‘A’	“C language” , “G” , “Java”
char형 (1byte)	char형 배열
char ch = ‘A’;	char c[] = “C language”;

[문자와 문자열의 선언]

문자열을 저장하기 위해 문자 배열을 사용하는데 문자열의 마지막을 의미하는 NULL 문자 ‘\0’가 마지막에 저장되어야 한다. 문자열이 저장되는 배열 크기는 반드시 저장될 문자 수 + 1

```
char java[] = {‘J’, ‘A’, ‘V’, ‘A’, ‘\0’};
→ 문자 하나 하나를 쉼표로 구분하여 입력하고 마지막 문자로 ‘\0’ 삽입

char c[] = “C language”;
→ 크기를 생략하고 초기화하는 것이 간편

char c[11] = “C language”;
→ 크기 지정 시 (문자수 + 1)

char go[5] = “go”;
→ 크기가 (문자수 + 1)보다 크면 나머지는 모두 ‘\0’으로 채워짐
```

2) 함수 printf()를 사용한 문자와 문자열 출력

[문자와 문자열 출력]

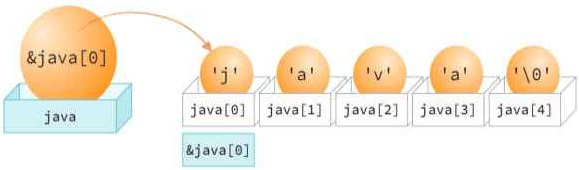
printf()에서 %c → 문자 출력
%s → 배열 이름 또는 문자 포인터를 사용하여 문자열 출력

puts() → 한 줄에 문자열 출력 후 다음 줄에서 출력을 준비
→ 배열 이름을 인자로 사용해도 문자열 출력 가능

[문자열 구성하는 문자 참조]

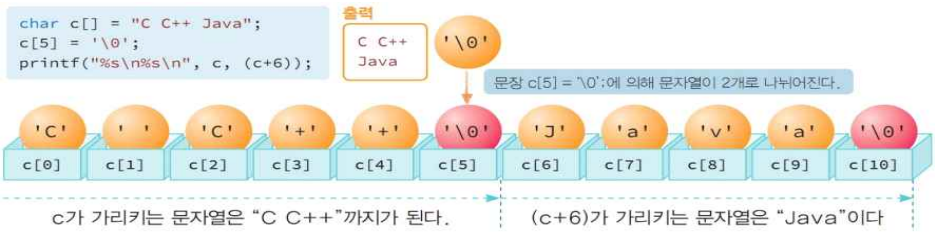
```
int i = 0;                          while (java[i] != '\0')
char *java = “java”;                  printf(“%s “, java[i++]);
printf(“%s ”, java);                printf(“\n“);
```

문자열을 구성하는 하나 하나의 문자를 배열 형식으로 직접 참조하여 출력할 수 있지만, 문자를 수정하거나 수정될 수 있는 함수의 인자로 사용하면 실행 오류가 발생



→ 반복문을 이용하여 문자가 '\0'이 아니면 문자를 출력
→ 변수 java가 가리키는 문자열은 상수이므로 수정 불가능

['\0' 문자에 의한 문자열 분리]
함수 printf()에서 %s는 문자 포인터가 가리키는 위치에서 NULL 문자까지를 하나의 문자열로 인식



3) 다양한 문자 입출력

	scanf("%c", &ch)	getchar()	_getche()	_getch()
헤더파일	stdio.h		conio.h	
버퍼 이용	O		X	
반응	enter 키를 눌러야 작동		문자 입력마다 반응	
입력 문자의 표시	누르면 바로 표시			표시 X
입력 문자 수정	가능		불가능	

4) 문자열 입력

[문자배열 변수로 scanf()에서 입력]
scanf() : 공백으로 구분되는 하나의 문자열을 입력 받음
① 문자 배열 str 선언
② scanf("%s", str)에서 형식 제어 문자 %s를 사용하여 문자열 입력
③ printf("%s", str)에서 %s를 사용하여 문자열 출력

[gets()와 puts()]
gets() : 한 행의 문자열 입력에 유용한 함수 (stdio.h 삽입)
: 문자열을 입력 받아 buffer에 저장하고 입력 받은 첫 문자의 주소값을 반환
: 표준입력으로 enter 키를 누를 때까지 공백을 포함한 한 행의 모든 문자열을 입력 받음
: 마지막에 입력된 '\n'가 '\0'로 교체되어 배열에 저장

puts() : 한 행에 문자열을 출력하는 함수 (stdio.h 삽입)
: 문자열의 마지막에 저장된 '\0'을 '\n'로 교체하여 버퍼로 전송한 후 문자열이 한 행에 출력
: 일반적으로 0인 정수를 반환하지만 오류 발생 시 EOF 반환

printf()와 scanf()는 다양한 입출력에 적합, puts()와 gets()는 처리 속도가 빠르다는 장점이 있음

2. 문자열 관련 함수

1) 문자 배열 라이브러리와 문자열 비교

[다양한 문자열 라이브러리 함수]

void *memchr(const void *str, int c, size_t n)
→ 메모리 str에서 n 바이트까지 문자 c를 찾아 그 위치를 반환

int *memcmp(const void *str1, const void *str2, size_t n)
→ 메모리 str과 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 dest에 n 바이트를 복사한 후 dest 위치 반환

void *memmove(void *dest, const void *src, size_t n)
→ 포인터 src 위치에서 dest에 n 바이트를 복사한 후 dest 위치 반환

void *memset(void *str, int c, size_t n)
→ 포인터 src 위치에서부터 n 바이트까지 문자 c를 지정한 후 src 위치 반환

size_t strlen(const char *str)
→ 포인터 src 위치에서부터 널 문자를 제외한 문자열의 길이 반환

[함수 strcmp()] 두 문자열을 비교하는 함수

int strcmp(const char *s1, const char *s2);

: 두 인자인 문자열에서 같은 위치의 문자를 앞에서부터 다를 때까지 바교하여 같으면 0, 앞이 크면 상수를, 뒤가 크면 음수를 반환

: 두 문자열을 사전상의 순서로 비교하는 함수

int strncmp(const char *s1, const char *s2, size_t maxn);

: 두 인자 문자열을 같은 위치의 문자를 앞에서부터 다를 때까지 비교하나 최대 n까지만 비교하여 같으면 0, 앞이 크면 양수를, 뒤가 크면 음수를 반환 / 두 문자를 비교할 문자의 최대 수를 지정

2) 문자열 복사와 연결

[함수 strcpy()] 문자열을 복사하는 함수

char *strcpy(char *dest, const char *source);

: 앞 문자열 dest에 처음에 뒤 문자열 null 문자를 포함한 source를 복사하여 복사된 문자열 반환

: 앞 문자열은 수정 가능, 뒤 문자열은 수정 불가능

char *strncpy(char *dest, const char *source, size_t maxn);

: 앞 문자열 dest에 처음에 뒤 문자열 source에서 n개 문자를 복사하여 복사된 문자열을 반환

: 만일 지정된 maxn이 source의 길이보다 길면 나머지는 모두 널 문자가 복사

: 앞 문자열은 수정 가능, 뒤 문자열은 수정 불가능

[함수 strcat()] 앞 문자열에 뒤 문자열의 null 문자까지 연결하여 문자열 주소를 반환하는 함수

char *strcat(char *dest, const char *source);

: 앞 문자열 dest에 뒤 문자열 source를 연결해 저장, 연결된 문자열 반환, 뒤 문자열 수정 불가능

char *strncat(char *dest, const char *source, size_t maxn);

: 앞 문자열 dest에 뒤 문자열 source 중에서 n개의 크기만큼을 연결해 저장, 연결된 문자열 반환, 뒤 문자열 수정 불가능

: 지정한 maxn이 문자열 길이보다 크면 null 문자까지 연결

3) 문자열 분리 및 다양한 문자열 관련 함수

[함수 strtok()] 문자열에서 구분자인 문자를 여러 개 지정하여 토큰을 추출하는 함수

① 문장 ptoken = strtok(str, delimiter);으로 첫 토큰 추출

② 결과를 저장한 ptoken이 NULL이면 더 이상 분리할 토큰이 없는 경우임

char *strtok(char *str, const char *delim);

: 앞 문자열 str에서 뒤 문자열 delim을 구성하는 구분자를 기준으로 순서대로 토큰을 추출 후 반환하는 함수, 뒤 문자열 delim은 수정 불가능

[문자열의 길이와 위치 검색]

strlen() : NULL 문자를 제외한 문자열 길이를 반환하는 함수

strlwr() : 인자를 모두 소문자로 변환하여 반환하는 함수

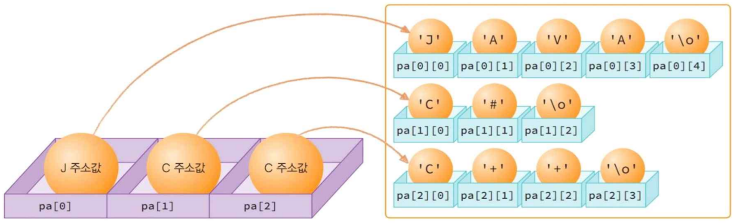
strupr() : 인자를 모두 대문자로 변환하여 반환하는 함수

3. 여러 문자열 처리

1) 문자 포인터 배열과 이차원 문자 배열

[문자 포인터 배열]

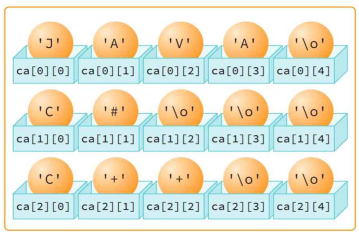
```
char *pa[] = {"JAVA", "C#", "C++"}; //배열의 크기는 공백이나 문자열 개수만큼 지정
printf("%s ", pa[0]); printf("%s ", pa[1]); printf("%s\n", pa[2]);
```



→ 문자열 수정 불가능
→ 메모리 낭비 X

[이차원 문자 배열]

```
char ca[][5] = {"JAVA", "C#", "C++"}; //행은 공백이나 문자열 개수 지정, 열은 가장 긴 문자열 길이+1로 지정
printf("%s ", ca[0]); printf("%s ", ca[1]); printf("%s\n", ca[2]);
```



→ 문자열 수정 가능
→ 메모리 낭비 0

2) 명령행 인자

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

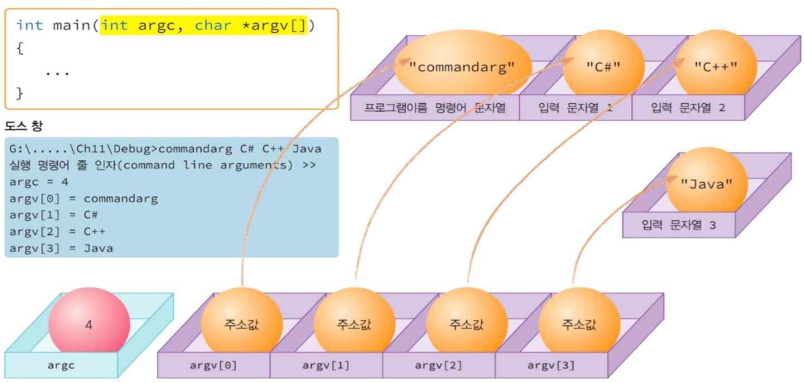
명령행 인자 사용법 : 명령행에서 입력하는 문자열을 프로그램으로 전달

프로그램에서 명령행 인자 받기

① 두 개의 인자 argc와 argv를 (int argc, char *argv[])로 기술

② 매개변수 argc는 명령행에서 입력한 문자열을 전달 받는 문자 포인터 배열

③ 실행 프로그램 이름도 하나의 명령행 인자에 포함되므로 주의할 것



2 변수 유효범위

1. 전역 변수와 지역 변수

1) 변수 범위와 지역 변수

2) 전역 변수와 extern

2. 정적 변수와 레지스터 변수

1) 기억 부류와 레지스터 변수

2) 정적 변수

3. 메모리 영역과 변수 이용

1) 메모리 영역

2) 변수의 이용

1. 전역 변수와 지역 변수

1) 변수 범위와 지역 변수

[변수 scope]

변수의 유효범위

- 지역 유효범위 : 함수나 블록 내부에서 선언, 그 지역에서 변수의 참조 가능
- 전역 유효범위 : 하나의 파일에서나 프로젝트를 구성하는 모든 파일에서 변수 참조 가능

파일 main.c

```
...
int global; //전체 전역
int main(void)
{

}

void asub()
{
    int local; //지역
    지역 유효 범위
    ...
}
```

파일 sub1.c

```
extern int global;
int sub1(void)
{

    ...
}

void asub1()
{
    ...
}
```

파일 sub2.c

전역 유효범위(파일 내부)

```
//파일 전역
static int staticvar;
int sub2(void)
{


}

void asub2()
{
    ...
}
```

main.c의 global - 전체 파일 전역 변수

main.c의 local - 함수 지역 변수

sub1.c의 global - 전체 파일 전역 변수를 사용하기 위해 선언

sub2.c의 staticvar - 현재 파일 전역 변수

[지역 변수]

- 함수 또는 블록에서 선언된 변수 (내부 변수 or 자동 변수)
- 선언 문장 이후에 함수나 블록의 내부에서만 사용 가능
- 함수의 매개변수도 함수 전체에서 사용 가능한 지역 변수
- 선언 후 초기화하지 않으면 쓰레기값이 저장되므로 주의
- 변수가 선언된 함수 또는 블록에서 선언 문장이 실행되는 시점에서 메모리에 할당[1)]

```
int main(void)
{
   //지역변수 선언
   int n = 10;

   printf("%d\n", n);

   //m, sum은 for 문 내부의 블록 지역변수
   for (int m = 0, sum = 0; m < 3; m++)
   {
      sum += m;
      printf("%d %d\n", m, sum);
   }

   printf("%d %d\n", n, sum);

   return 0;
}
```

지역변수 n의 영역 유효 범위

지역변수 sum 영역 유효 범위

오류발생: error C2065: 'sum' : 선언되지 않은 식별자입니다.

지역 변수를 자동 변수라 하는 이유
: 선언된 부분에서 자동으로 생성되고 함수나 블록이 종료되는 순간 메모리에서 자동으로 제거되기 때문

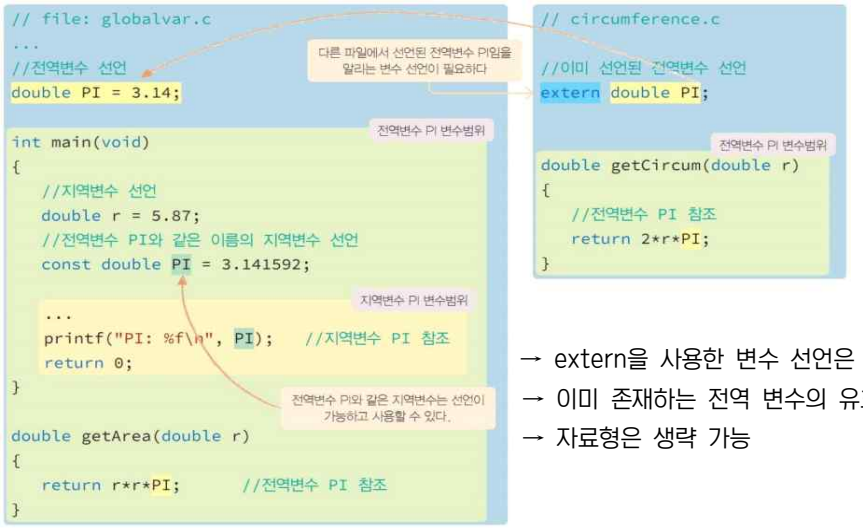
자료형 앞에 auto가 사용될 수 있으나 생략 가능, 일반적으로 auto가 없는 경우가 대부분

1) 스택(stack) : 지역 변수가 할당되는 메모리 영역

2) 전역 변수와 extern

[전역 변수]

- 함수 외부에서 선언되는 변수 (외부 변수)
- 일반적으로 프로젝트의 모든 함수나 블록에서 참조 가능
- 선언 후 자동으로 초기값이 자료형에 맞는 0으로 지정 (정수형:0, 문자형:'\0', 실수형:0.0, 포인터형:NULL)
- 함수나 블록에서 전역 변수와 같은 이름으로 지역 변수를 선언할 수 있지만, 함수 내부나 블록에서 그 이름을 참조하면 지역 변수로 인식하기 때문에 전역 변수는 참조 불가능



→ extern을 사용한 변수 선언은 새로운 변수 선언 X
→ 이미 존재하는 전역 변수의 유효범위를 확장
→ 자료형은 생략 가능

[전역 변수 장단점]

전역 변수의 선언 위치가 변수를 참조하려는 위치보다 뒤에 있는 경우

- 전역 변수를 사용하기 위해서는 extern을 사용한 참조 선언이 필요
- 소스 파일 중간이나 하단에 전역 변수를 배치하지 않도록 주의

장단점

- 장점 : 전역 변수는 어디에서든지 수정할 수 있으므로 사용이 편리
- 단점 : 전역 변수에 예상하지 못한 값이 저장되어 프로그램 어느 부분에서 수정됐는지 알기 어려움

2. 정적 변수와 레지스터 변수

1) 기억 부류와 레지스터 변수

[auto, register, static, extern]

- 변수는 선언 위치에 따라 전역과 지역으로 나뉘고 4가지 기억 부류에 따라 할당되는 메모리 영역, 메모리의 할당과 제거 시기가 결정
- 자동 변수인 auto는 일반 지역 변수로 생략될 수 있으므로 주의
- extern이 선언되는 위치에 따라 이 변수의 사용 범위는 전역 또는 지역으로 한정
- 새로운 변수의 선언에 사용되는 키워드 : auto, register, static

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

기억 부류 사용 구문

- 변수 선언 문장에서 자료형 앞에 하나의 키워드를 넣는 방식
- extern을 제외하고 나머지 3개의 기억 부류의 변수 선언에서 초기값 저장 가능
- auto는 지역 변수 선언에 사용되며 생략 가능 (함수에 선언된 모든 변수가 auto가 생략된 자동 변수)

```
기억부류 자료형 변수이름;
기억부류 자료형 변수이름 = 초기값;
```

```
auto int n;
register double yield;
static double data = 5.85;
int age;
extern int input;
```

[키워드 register]

- 변수의 저장 공간이 일반 메모리가 아니라 CPU 내부의 레지스터에 할당되는 변수
- 키워드 register를 자료형 앞에 넣어 선언, 지역 변수에만 이용 가능
- 지역 변수로서 함수나 블록이 시작되면서 CPU의 내부 레지스터에 값이 저장
- 함수나 블록을 빠져나오면서 값이 소멸되는 특성
- CPU 내부에 있는 기억 장소이므로 일반 메모리보다 빠르게 참조 가능
- 일반 메모리에 할당되는 변수가 아니므로 주소연산자 &를 사용 불가능 (& 사용하면 문법 오류 발생)
- 시스템의 레지스터는 한정되어 있어 모자라면 일반 지역 변수로 할당
- 처리 속도를 증가시키려는 변수에 이용, 반복문의 횟수를 제어하는 제어변수에 효과적
- 일반 지역 변수와 같이 초기값이 저장되지 않으면 쓰레기값이 저장

2) 정적 변수

[키워드 static]

- 키워드 static을 자료형 앞에 넣어 정적 변수를 선언
- 초기 생성된 이후 메모리에서 제거되지 않으므로 지속적으로 저장값을 유지하거나 수정 가능
- 프로그램 시작되면 메모리에 할당되고 종료되면 메모리에서 제거, 초기화는 한 번만 수행
- 초기값 지정하지 않으면 자동으로 자료형에 따라 0, '\0', NULL 값이 저장
- 초기화된 정적 변수는 프로그램 실행 중간에 더 이상 초기화 X, 초기화는 상수로만 가능

전역 변수 선언 시 static을 가장 앞에 붙이면 정적 전역 변수
→ 참조 범위는 선언된 파일에만 한정, 변수의 할당과 제거는 전역 변수의 특징을 가짐

지역 변수 선언 시 static을 가장 앞에 붙이면 정적 지역 변수
→ 참조 범위는 지역 변수, 변수의 할당과 제거는 전역 변수의 특징을 가짐

[정적 지역 변수]
- 함수나 블록에서 정적으로 선언되는 변수
- 유효범위 : 선언된 블록 내부에서만 참조 가능 (지역 변수 특성)
- 함수나 블록을 종료해도 메모리에서 제거되지 않고 계속 유지 관리 (전역 변수 특성)
- 함수에서 이전에 호출되어 저장된 값을 유지하여 이번 호출에 사용 가능

[정적 전역 변수]
- 함수나 외부에서 정적으로 선언되는 변수
- 유효범위 : 선언된 파일 내부에서만 참조 가능 (extern으로 다른 파일에서 참조 불가능)
- 가급적 전역 변수 사용 자제, 파일에서만 전역 변수로 이용할 수 있는 정적 전역 변수 이용할 것

3. 메모리 영역과 변수 이용

1) 메모리 영역

[데이터, 스택, 힙 영역]

메인 메모리의 영역은 프로그램 실행 과정에서 데이터, 힙, 스택 영역 세 부분으로 나뉘고 변수의 유효범위와 생존 기간에 결정적인 역할

데이터 영역

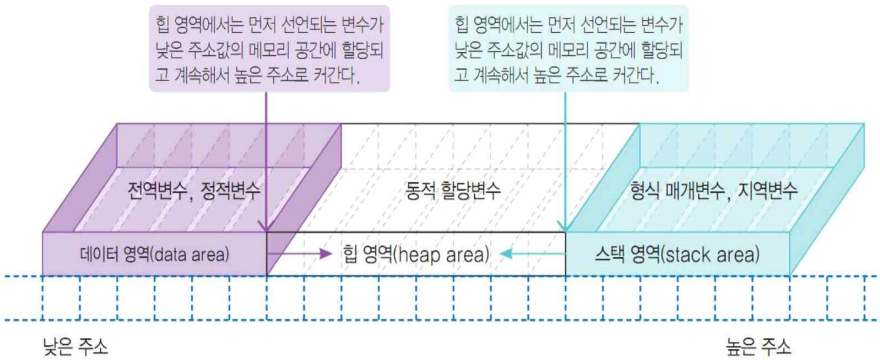
- 전역 변수와 정적 변수가 할당되는 저장 공간
- 메모리 주소가 낮은 값 → 높은 값으로 저장 장소가 할당
- 프로그램이 시작되는 시점에 정해진 크기대로 고정된 메모리 영역이 확보

힙 영역

- 동적 할당되는 변수가 할당되는 저장 공간
- 데이터 영역과 스택 영역 사이에 위치

스택 영역

- 함수 호출에 의한 형식 매개변수 그리고 함수 내부의 지역 변수가 할당되는 저장 공간
- 힙 영역과 스택 영역은 프로그램이 실행되면서 영역 크기가 계속적으로 변화
- 메모리 주소가 높은 값 → 낮은 값으로 저장 장소가 할당
- 함수 호출과 종료에 따라 메모리가 할당 되었다가 다시 제거되는 작업이 반복



2) 변수의 이용

[이용 기준]

- 전역 변수의 사용을 자제하고 지역 변수를 주로 이용
- 레지스터 변수 : 실행 속도 개선하고자 하는 경우
- 정적 지역 변수 : 함수나 블록 내부에서 종료되더라도 계속적으로 값 저장
- 정적 전역 변수 : 해당 파일 내부에서만 변수를 공유하고자 하는 경우
- 전역 변수 : 프로그램의 모든 영역에서 값을 공유하고자 하는 경우

변수 할당 메모리 영역에 따라 변수의 할당과 제거 시기가 결정

<데이터 영역> 전역 변수와 지역 변수 → 프로그램 시작 시 할당, 종료 시 제거

<스택 영역과 레지스터> 자동 지역 변수와 레지스터 변수 → 함수|블록 시작 시 할당, 종료 시 제거

<table>
<tr><th>선언 위치</th><th>상세 종류</th><th colspan="2">키워드</th><th>유효범위</th><th>기억 장소</th><th>생존기간</th></tr>
<tr><td rowspan="2">전역</td><td>전역 변수</td><td>참조선언</td><td>extern</td><td>프로그램 전역</td><td rowspan="2">메모리
(데이터 영역)</td><td rowspan="2">프로그램
실행 시간</td></tr>
<tr><td>정적 전역 변수</td><td colspan="2">static</td><td>파일 내부</td></tr>
<tr><td rowspan="4">지역</td><td>정적 지역 변수</td><td colspan="2">static</td><td rowspan="3">함수나 블록 내부</td><td rowspan="2">레지스터</td><td rowspan="3">함수 | 블록
실행 시간</td></tr>
<tr><td>레지스터 변수</td><td colspan="2">register</td></tr>
<tr><td>자동 지역 변수</td><td colspan="2">auto(생략가능</td><td>메모리
(스택 영역)</td></tr>
</table>

변수의 유효범위

구분	종류	메모리 할당	동일 파일 외부 함수에서의 이용	다른 파일 외부 함수에서의 이용	메모리 제거
전역	전역 변수	프로그램 시작	O	O	프로그램 종료
	정적 전역 변수	프로그램 시작	O	X	프로그램 종료
지역	정적 지역 변수	프로그램 시작	X	X	프로그램 종료
	레지스터 변수	함수(블록) 시작	X	X	함수(블록) 종료
	자동 지역 변수	함수(블록) 시작	X	X	함수(블록) 종료

변수의 초기값

<table>
<tr><th>지역, 전역</th><th>종류</th><th>자동 저장되는 기본 초기값</th><th>초기값 저장</th></tr>
<tr><td rowspan="2">전역</td><td>전역 변수</td><td rowspan="2">자료형에 따라
0, '\0', NULL 값 저장</td><td rowspan="2">프로그램 시작 시</td></tr>
<tr><td>정적 전역 변수</td></tr>
<tr><td rowspan="3">지역</td><td>정적 지역 변수</td></tr>
<tr><td>레지스터 변수</td><td rowspan="2">쓰레기값 저장</td><td rowspan="2">함수(블록) 실행될 때마다</td></tr>
<tr><td>자동 지역 변수</td></tr>
</table>

3 구조체와 공용체

1. 구조체와 공용체

1) 구조체 개념과 정의

2) 구조체 변수 선언과 초기화

3) 구조체 활용

4) 공용체 활용

2. 자료형 재정의

1) 자료형 재정의 typedef

2) 구조체 자료형 재정의

3. 구조체와 공용체의 포인터와 배열

1) 구조체 포인터

2) 구조체 배열

1. 구조체와 공용체

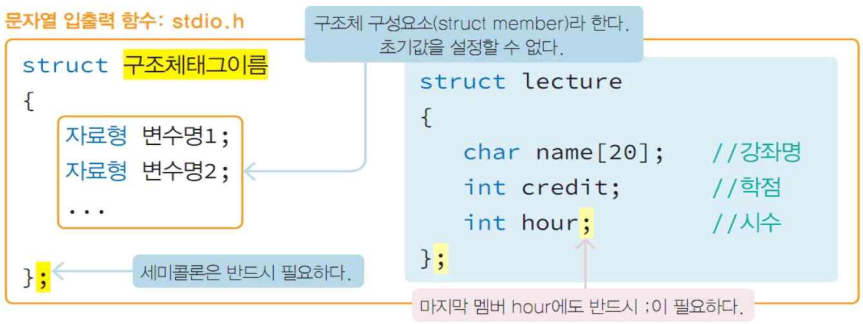
1) 구조체 개념과 정의

[구조체 개념]

- 정수, 문자, 실수나 포인터 그리고 이들의 배열 등을 묶어 하나의 자료형으로 이용하는 것
- 연관성이 있는 서로 다른 개별적인 자료형의 변수들을 하나의 단위로 묶은 새로운 자료형
- 연관된 멤버로 구성되는 통합 자료형으로 대표적인 [1]유도 자료형

[구조체 정의]

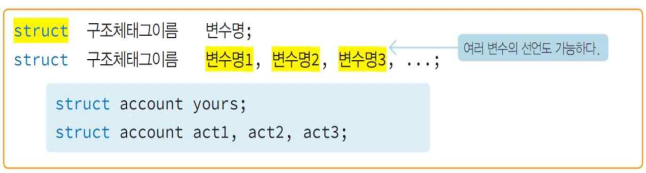
- 구조체를 자료형으로 사용하려면 먼저 만들 구조체 틀을 정의
- 구조체 정의는 변수의 선언과는 다른 것으로 변수 선언에서 이용될 새로운 구조체 자료형을 정의
- 모두 하나의 문장이므로 반드시 세미콜론으로 종료, 각 구조체 멤버의 초기값 대입 불가능
- 모든 멤버 선언에 반드시 세미콜론 삽입, 구조체 멤버의 이름은 모두 유일해야 함
- 같은 자료형이 연속적일 경우 int credit; int hour; → int credit, hour; 가능
- 구조체 멤버로는 일반 변수, 포인터 변수, 배열, 다른 구조체 변수 및 구조체 포인터도 허용



2) 구조체 변수 선언과 초기화

[구조체 변수 선언]

- 구조체 정의 후 구조체 struct account가 새로운 자료유형으로 사용 가능
- struct account형 변수 mine을 선언 → struct account mine;
- 기존의 일반 변수 선언과 같이 동시에 여러 개의 변수 선언도 가능



1) 유도 자료형 : 기존 자료형으로 새로이 만들어진 자료형

※ TIP 이름 없는 구조체
- 이 구조체와 동일한 자료형의 변수를 더 이상 선언 불가능
- 단 한 번 이 구조체형으로 변수를 선언하는 경우에만 이용
- 태그이름이 없는 구조체 정의에서는 바로 변수가 나오지 않는다면 아무 의미 없는 문장

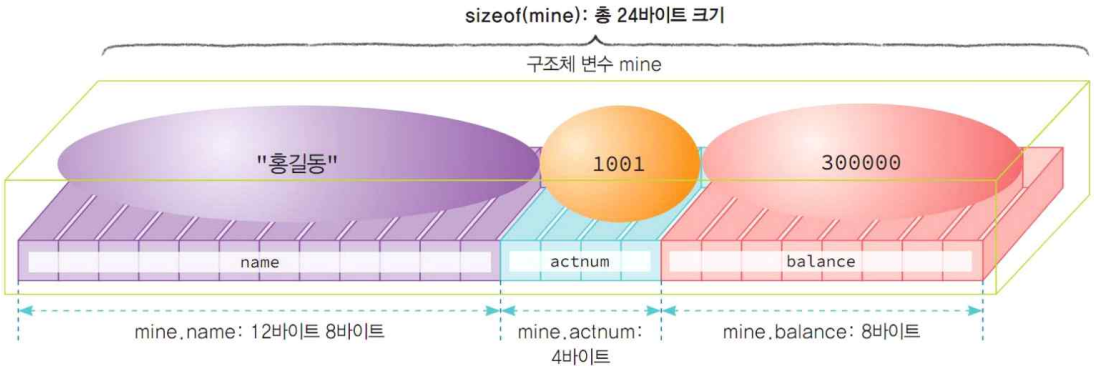
```
struct account
{
   char name[12];       //계좌주이름
   int actnum;          //계좌번호
   double balance;      //잔고
} youraccount;
```
변수 youraccount이후에 이와 동일한 구조체의 변수선언은 불가능하다.

[구조체 변수의 초기화]
struct 구조체태그이름 변수명 = {초기값1, 초기값2, 초기값3, ...};
: 중괄호 내부에서 각 멤버 정의 순서대로 초기값을 쉼표로 구분하여 기술
: 기술되지 않은 멤버값은 자료형에 따라 0, 0.0, '\0'으로 저장

[구조체의 멤버 접근 연산자 .와 변수 크기]
- 선언된 구조체형 변수는 접근연산자(참조연산자) .을 사용하여 멤버 참조 가능
- mine.actnum = 1002; → 구조체 변수 mine의 멤버 actnum에 1002를 저장

```
구조체변수이름.멤버
mine.actnum = 1002;    mine.balance = 300000;
```

- 구조체 struct account의 변수 mine은 다음 구조로 메모리에 할당
- 변수 mine의 크기는 sizeof(mine)으로 확인 가능
- 실제 구조체의 크기는 멤버 크기의 합보다 크거나 같음



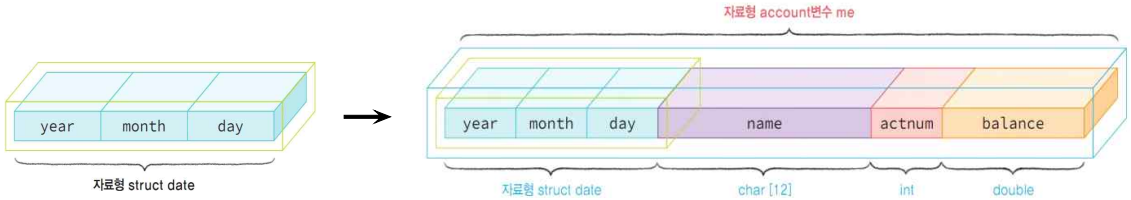
3) 구조체 활용

[구조체 멤버로 사용되는 구조체]
멤버로 이미 정의된 다른 구조체형 변수와 자기 자신을 포함한 구조체 포인터 변수 사용 가능

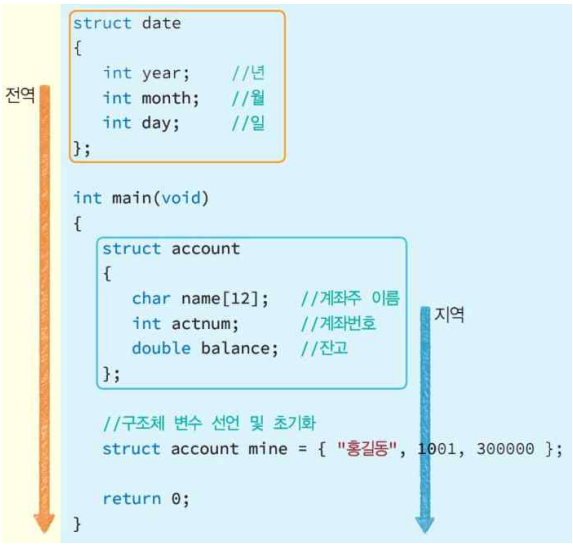
```
struct date
{
   int year;      //년
   int month;     //월
   int day;       //일
};
```

→

```
struct account
{
   struct date open;    //계좌 개설일자
   char name[12];       //계좌주 이름
   int actnum;          //계좌번호
   double balance;      //잔고
};
   struct account me = {{2012, 3, 9}, "홍길동", 1001, 300000 };
```



※ TIP 구조체 정의의 위치



구조체 정의 위치에 따라 구조체 유효범위 결정
전역 : main() 함수 외부 상단에서 정의
지역 : main() 함수, 다른 함수 내부에서 정의

[구조체 변수의 대입과 동등 비교]
동일한 구조체형의 변수는 멤버마다 모두 대입할 필요X, 변수 대입으로 한번에 모든 멤버 대입 가능

```
struct student
{
   int snum;    //학번
   char *dept;    //학과 이름
   char name[12];    //학생 이름
};
struct student hong = { 201800001, "컴퓨터정보공학과", "홍길동" };
struct student one;

one = hong;
```

- struct student 형의 변수 hong과 one에서 (one == hong) 동등 비교는 불가능
- 구조체를 비교하려면 구조체 멤버, 하나 하나를 비교

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

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

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

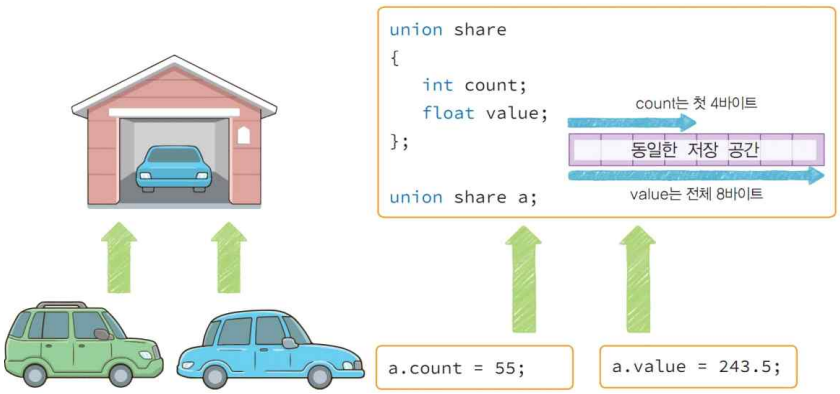
※ TIP 문자열을 처리하기 위한 포인터 char *와 배열 char[]

char 포인터	char 배열
char *dept; //학과 이름	char name[12]; //학생 이름
char *dept = "컴퓨터정보공학과";	char name[12] = "나한국";
변수 dept는 포인터로 단순히 문자열 상수를 다루는 경우 효과적	변수 name은 배열로 12byte 공간을 가지며 문자열을 저장하고 수정 등이 필요한 경우 효과적
dept = "컴퓨터정보공학과";	name = "나한국"; //오류
단지 문자열 상수의 첫 주소를 저장하므로 문자열 자체를 저장하거나 수정하는 것은 불가능하므로 다음 구문은 사용 불가능	문자열 자체를 저장하는 배열이므로 문자열의 저장 및 수정이 가능하고 문자열 자체를 저장하는 다음 구문 사용 가능
strcpy(dept, "컴퓨터정보공학과"); //오류	strcpy(name, "배상문");
scanf("%s", dept); //오류	scanf("%s", name);

4) 공용체 활용

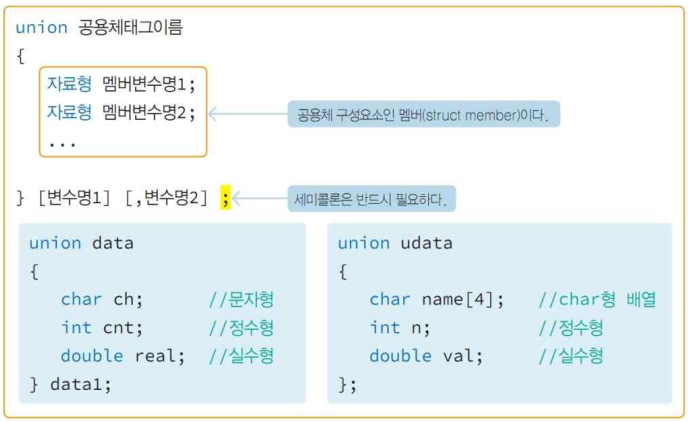
[공용체 개념]

- 동일한 저장 장소에 여러 자료형을 저장하는 방법
- 공용체를 구성하는 멤버에 한 번에 한 종류만 저장하고 참조 가능



[union을 사용한 공용체 정의 및 변수 선언]

- 서로 다른 자료형의 값을 동일한 저장 공간에 저장하는 자료형
- union을 struct로 사용하는 것을 제외하면 구조체 선언 방법과 동일



공용체 변수의 크기
- 멤버 중 가장 큰 자료형의 크기로 정해짐
- 여러 멤버의 값을 동시에 저장하여 이용X, 마지막에 저장된 단 하나의 멤버 자료값만 저장
- 공용체도 구조체와 같이 typedef를 이용하여 새로운 자료형으로 정의 가능

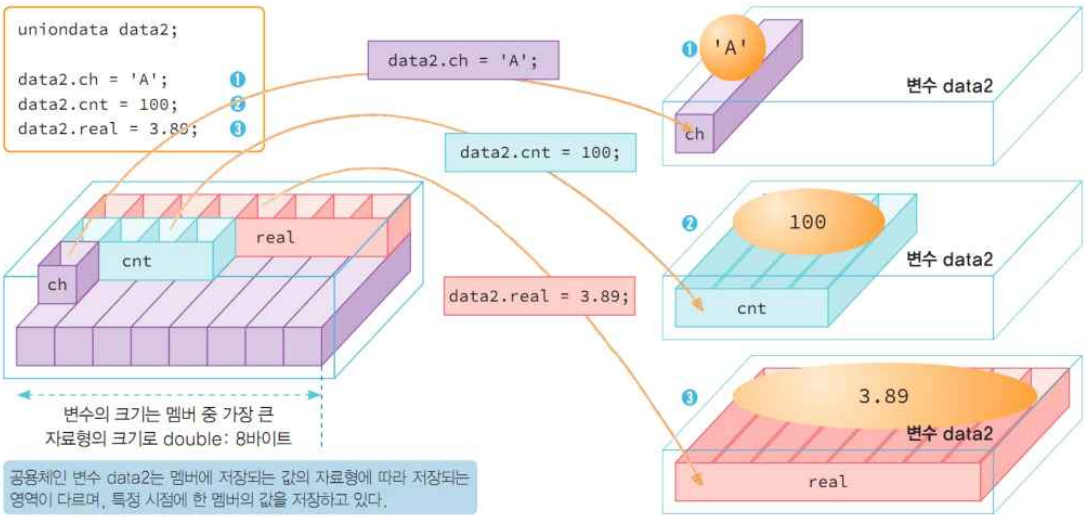
공용체의 초기화
- 공용체 정의시 처음 선언한 멤버의 초기값으로만 저장 가능
- 다른 멤버로 초기값을 지정하면 컴파일 시 경고 발생
- 초기값으로 동일한 유형의 다른 변수의 대입도 가능

```
typedef union data uniondata;

   uniondata data2 = {'A'};       //첫 멤버인 char형으로만 초기화 가능
   //uniondata data2 = {10.3};    //컴파일 시 경고 발생
   warning C4244: '초기화중' : 'double'에서'char'(으)로 변환하면서 데이터가 손실될 수 있습니다.

   uniondata data3 = data2;       //다른 변수로 초기화 가능
```

[공용체 멤버 접근]
- 구조체와 같이 접근연산자 .을 사용
- data.ch = 'A'; → 공용체 변수 data2의 멤버 ch에 문자 'A'를 저장
- 항상 마지막에 저장한 멤버로 접근해야 원하는 값을 얻을 수 있음

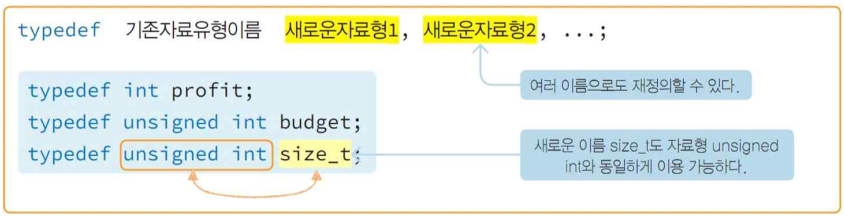


2. 자료형 재정의

1) 자료형 재정의 typedef

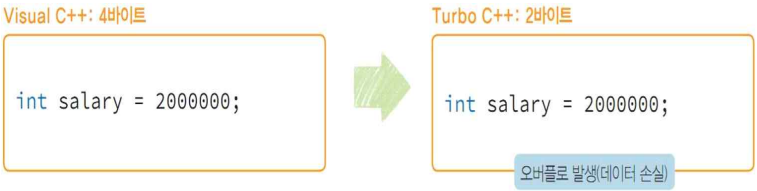
[typedef 구문]

이미 사용되는 자료 유형을 다른 새로운 자료형 이름으로 재정의할 수 있도록 하는 키워드



재정의하는 이유 : 프로그램의 시스템 간 호환성과 편의성을 위해 필요

- 터보 C++ 컴파일러에서 자료 유형 int는 저장 공간이 2byte / Visual C++에서는 4byte
- Visual C++에서 작성한 프로그램이 터보 C++에서 문제 발생 (2byte로는 2000000 저장X)



↓해결방안↓

- Visual C++에서 int를 myint로 재정의하고 모든 int형을 myint형으로 선언하여 이용
- 이 소스를 터보 C++에서 컴파일한다면 typedef 문장에서 int를 long으로 수정
- 다른 소스는 수정 없이 그대로 이용 가능

하나의 자료형을 여러 자료형으로 재정의 가능

```
typedef int integer, word;

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

자료형 int를 새로운 자료형 이름 integer와 word로 재정의

함수 내부에서 재정의 : 선언된 이후 그 함수에서만 이용

함수 외부에서 재정의 : 재정의된 이후 그 파일에서 이용

2) 구조체 자료형 재정의

[struct를 생략한 새로운 자료형]

① typedef 사용하여 구조체 struct date를 date로 재정의하면 항상 struct를 쓰는 불편함 감소

```
struct date
{
   int year;     //년
   int month;    //월
   int day;      //일
};

typedef struct date date;
```

자료유형인 date는 struct date와 함께 동일한 자료유형으로 이용이 가능하다.

→ date가 아닌 다른 이름으로도 재정의 가능

② typedef 구문에서 새로운 자료형으로 software형이 정의

```
typedef struct
{
   char title[30];     //제목
   char company[30];   //제작회사
   char kinds[30];     //종류
   date release;       //출시일
} software;
```

software는 변수가 아니라 새로운 자료형이다.

→ 구문 이후 software를 구조체 자료형으로 변수 선언에 사용
→ 구조체 태그 이름은 생략 가능
→ 구조체 software형은 멤버로 구조체 date형 변수 release

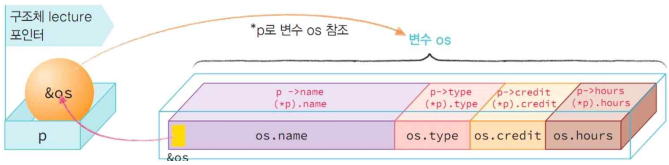
3. 구조체와 공용체의 포인터와 배열

1) 구조체 포인터

[포인터 변수 선언]

구조체 포인터는 구조체의 주소값을 저장할 수 있는 변수 (일반 포인터 변수 선언과 동일하게 선언)

```
lecture os = {"운영체제", 2, 3, 3};
lecture *p = &os;
```



[포인터 변수의 구조체 멤버 접근 연산자 ->]

포인터 p가 가리키는 구조체 변수의 멤버 name을 접근하는 연산식 (p -> name과 같이 사용)

- p->type, p->credit, p->hours는 각각 os.type, os.credit, os.hours를 참조
- -> 에서 - 와 > 사이에 공백 X
- 연산식 p->name은 접근연산자(.)와 간접연산자(*)를 사용한 (*p).name으로도 사용 가능
  → (*p).name은 *p.name과는 다르다는 것에 주의
- 연산식 *p.name은 접근연산자(.)가 간접연산자(*)보다 우선순위가 빨라 *(p.name)과 같은 연산식
  → p가 포인터 이므로 p.name은 문법 오류가 발생
- 접근연산자 -> 와 .은 간접연산자 *를 포함한 다른 어떠한 연산자 우선순위보다 높음
- 연산자 -> 와 .은 우선순위 1위, 결합성은 좌→우 / 연산자 *은 우선순위 2위, 결합성은 우→좌

구조체 변수와 구조체 포인터 변수를 이용한 멤버의 참조

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 문법 오류가 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)을 의미하며, 포인터 p이 가리키는 구조체의 멤버 name이 가리키는 변수로 이 경우는 구조체 포인터 p이 가리키는 구조체의 멤버 강좌명의 첫 문자임, 마찬가지로 한글인 경우에는 실행 오류

[공용체 포인터]

공용체 변수도 포인터 사용 가능, 공용체 포인터 변수로 멤버를 접근하려면 접근연산자 -> 이용

```
union data
{
   char ch;
   int cnt;
   double real;
} value, *p ;
   변수 value는 union data형이며 p는 union data 포인터 형으로 선언
p = &value;      //포인터 p에 value의 주소값을 저장
p->ch = 'a';     //value.ch = 'a';와 동일한 문장
```

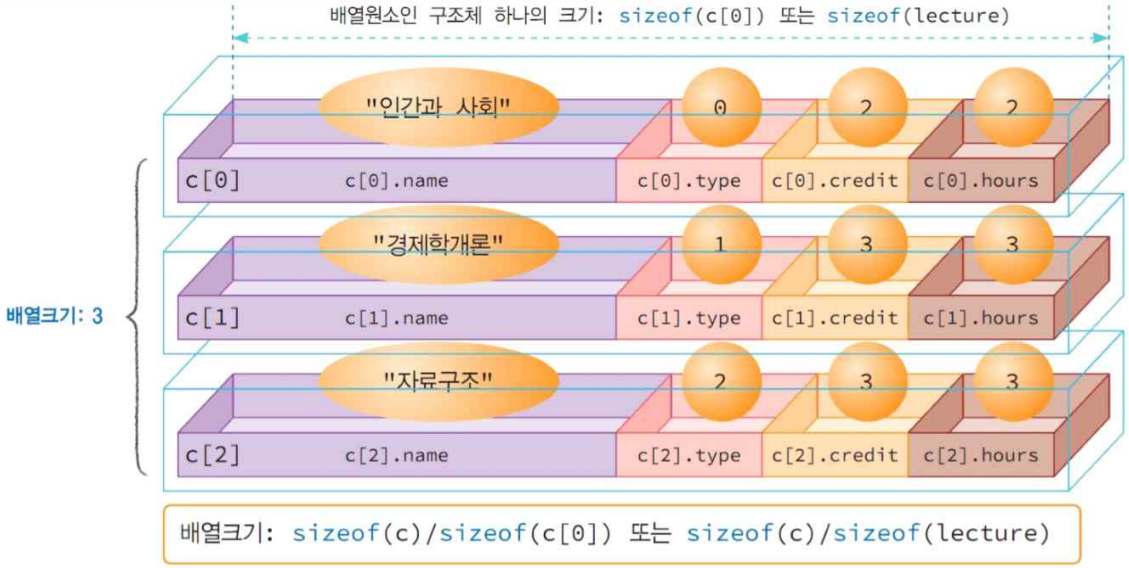
2) 구조체 배열

[구조체 배열 변수 선언]

동일한 구조체 변수가 여러 개 필요하면 구조체 배열을 선언

- 구조체 배열의 초기값 지정 구문에서는 중괄호가 중첩되게 표시
- 외부 중괄호는 배열 초기화의 중괄호, 내부 중괄호는 배열 원소인 구조체 초기화를 위한 중괄호

```
lecture c[] = { {"인간과사회", 0, 2, 2},
                {"경제학개론", 1, 3, 3},
                {"자료구조", 2, 3, 3}     };
```



4 함수와 포인터 활용

1. 함수의 인자 전달 방식

1) 값에 의한 호출과 참조에 의한 호출

2) 배열의 전달

3) 가변 인자

2. 포인터 전달과 반환

1) 매개변수와 반환으로 포인터 사용

2) 상수를 위한 const 사용

3) 함수의 구조체 전달과 반환

3. 함수 포인터와 void 포인터

1) 함수 포인터

2) 함수 포인터 배열

3) void 포인터

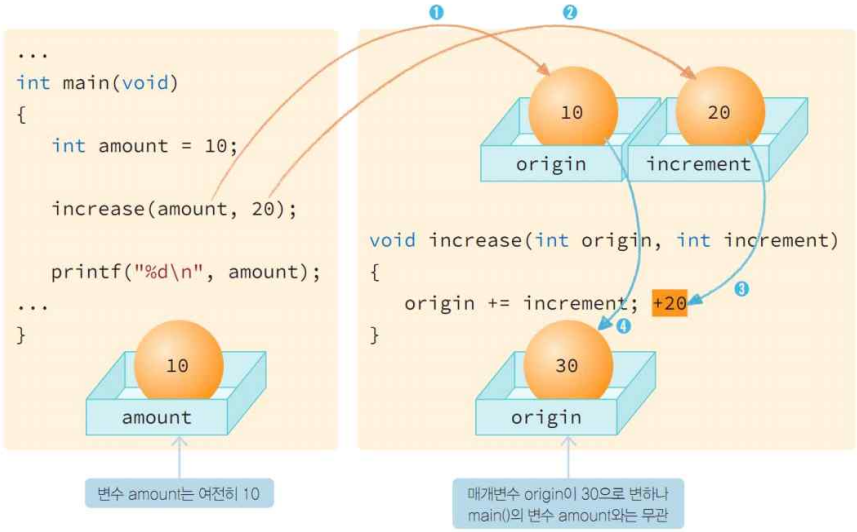
1. 함수의 인자 전달 방식

1) 값에 의한 호출과 참조에 의한 호출

[함수에서 값의 전달]

C언어는 함수의 인자 전달 방식이 기본적으로 값에 의한 호출 방식[1)]

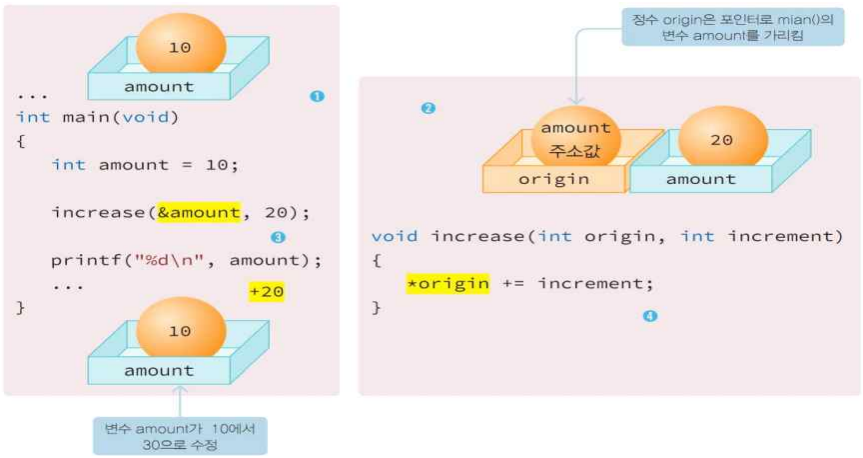
- 함수 호출 시 변수 amount의 값 10이 매개변수인 origin에 복사, 20이 매개변수인 increase에 복사되는 방식으로 함수 외부의 변수를 함수 내부에서 수정할 수 없는 특징



[함수에서 주소의 전달]

함수에서 주소의 호출을 참조에 의한 호출[2)]

- 첫 번째 매개변수를 int*로 수정, 함수 구현도 *orgin+=increment;로 수정하여 구현
- 함수 호출 시 첫 번째 인자가 &amount이므로 변수 amount의 주소값이 매개변수인 orgin에 복사, 20이 매개변수인 increment에 복사
- 함수 increase() 내부 실행에서 *orgin은 변수 amount 자체를 의미, *orgin ↑ → amount ↑



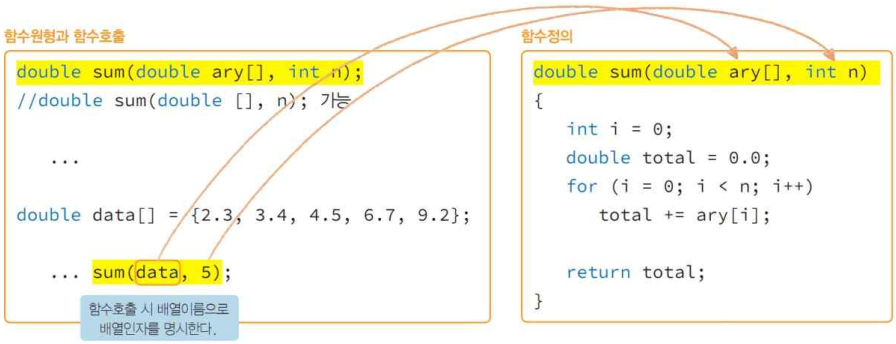
1) 값에 의한 호출 : 함수 호출 시 실인자의 값이 형식 인자에 복사되어 저장된다는 의미
2) 참조에 의한 호출 : 포인터를 매개변수로 사용하면 함수로 전달된 실인자의 주소를 이용하여 그 변수를 참조 가능

2) 배열의 전달

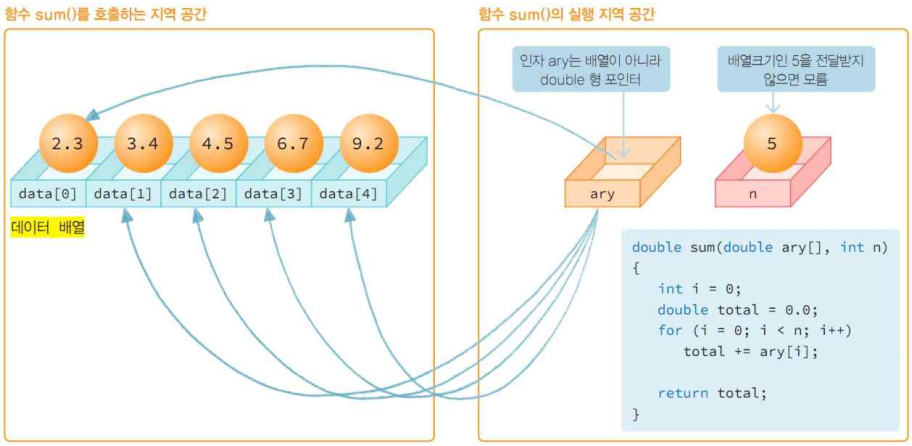
[배열 이름으로 전달]

함수의 매개변수로 배열을 전달하는 것은 배열의 첫 원소를 참조 매개변수로 전달하는 것과 동일

- 첫 번째 형식 매개변수에서 배열 자체에 배열 크기를 기술하는 것은 아무의미가 없음
- double ary[5]보다는 double ary[]라고 기술하는 것을 권장
- 실제로 함수 내부에서 실인자로 전달된 배열의 배열 크기를 알 수 없음
- 매개변수를 double ary[]처럼 기술해도 단순히 double *ary처럼 포인터 변수로 인식
- 배열 크기를 두 번째 인자로 사용



배열 크기를 인자로 사용하지 않으면 정해진 상수를 함수 정의 내부에서 사용해야 하기 때문에 배열 크기가 변하면 소스를 수정해야 하므로 비효율적 → 배열 크기도 하나의 인자로 사용



- 함수 원형에서 매개변수는 배열 이름 생략하고 double[]와 같이 기술 가능
- 함수 호출에서 배열 인자에는 반드시 배열 이름으로 sum(data, 5)로 기술

[다양한 배열 원소 참조 방법]

- 배열 point에서 간접 연산자를 사용한 배열 원소의 접근 방법은 *(point + I)
- 배열의 합을 구하려면 sum += *(point + i); 문장을 반복
- 문장 int *address = point;로 배열 point를 가리키는 포인터 변수 address를 선언하여 point를 저장 → 문장 sum += *(address++)으로도 배열의 합 가능
- 배열 이름 point는 주소 상수이기 때문에 sum += *(point++)는 사용 불가능
  → 증가 연산식 point++의 피연산자로 상수인 point를 사용할 수 없기 때문

간접 연산자 *를 사용한 배열 원소의 참조 방법

```
int i, sum = 0;
int point[] = {95, 88, 76, 54, 85, 33, 65, 78, 99, 82};
int *address = point;
int aryLength = sizeof (point) / sizeof (int);
```

가능
```
for (i=0; i<aryLength; i++)
  sum += *(point+i);
```

가능
```
for (i=0; i<aryLength; i++)
  sum += *(address++);
```

오류
```
for (i=0; i<aryLength; i++)
  sum += *(point++);
```

함수 헤더에 int ary[]로 기술하는 것은 int *ary로도 대체 가능

같은 의미로 모두 사용할 수 있다

```
int sumary(int ary[], int SIZE)
{
   ...
}
```

```
int sumaryf(int *ary, int SIZE)
{
   ...
}
```

```
for (i = 0; i < SIZE; i++)
{
   sum += ary[i];
}
```

```
for (i = 0; i < SIZE; i++)
{
   sum += *(ary + i);
}
```

```
for (i = 0; i < SIZE; i++)
{
   sum += *ary++;
}
```

```
for (i = 0; i < SIZE; i++)
{
   sum += *(ary++);
}
```

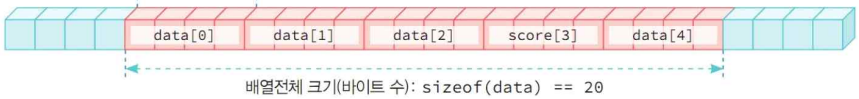
- 변수 ary는 포인터 변수로서 주소값을 저장하는 변수이므로 증가 연산자의 이용이 가능
- 연산식 *ary++는 *(ary++)와 같은 의미 (후위 증가연산자 (ary++)의 우선순위가 가장 높기 때문에)

[배열 크기 계산 방법]
- 배열이 함수 인자인 경우, 대부분 배열 크기도 함수 인자로 하는 경우가 일반적
- 배열 크기 : (sizeof(배열이름) / sizeof(배열원소))

```
int data[] = {12, 23, 17, 32, 55};
```

```
배열크기(배열원소의 수) = sizeof(배열이름) / sizeof(배열원소)

int  arraysize = sizeof(data) / sizeof(data[0]);
```



[다차원 배열 전달]

다차원 배열을 인자로 이용하는 경우 첫번째 대괄호 내부의 크기를 제외한 다른 모든 크기는 반드시 기술되어야 하고 이차원 배열의 행의 수를 인자로 이용하면 보다 일반화된 함수 구현 가능

함수원형과 함수호출

```
...
//이차원 배열값을 모두 더하는 함수원형
double sum(double data[][3], int, int);
//이차원 배열값을 모두 출력하는 함수원형
void printarray(double data[][3], int, int);

...
double x[][3] = { {1, 2, 3}, {7, 8, 9},
 {4, 5, 6}, {10, 11, 12} };

int rowsize = sizeof(x) / sizeof(x[0]);
int colsize = sizeof(x[0]) / sizeof(x[0][0]);

printarray(x, rowsize, colsize);

... sum(x, rowsize, colsize) ...

...
```

함수정의

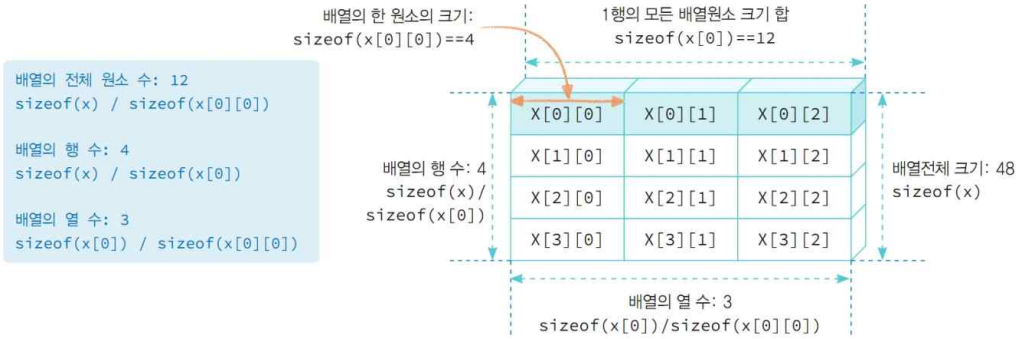
```
//이차원 배열값을 모두 출력하는 함수
void printarray(double data[][3], int rowsize, int colsize)
{

...

}

//이차원 배열값을 모두 더하는 함수
double sum(double data[][3], int rowsize, int colsize)
{
...
   for (i = 0; i < rowsize; i++)
      for (j = 0; j < colsize; j++)
         total += data[i][j];
   return total;
}
```

- 이차원 배열의 행의 수 : (sizeof(x) / sizeof(x[0]))
- 이차원 배열의 열의 수 : (sizeof(x[0]) / sizeof(x[0][0]))
- sizeof(x)는 배열 전체의 바이트 수, sizeof(x[0])는 1행의 바이트 수
- sizeof(x[0][0])은 첫 번째 원소의 바이트 수



3) 가변 인자

[가변 인자가 있는 함수 머리]

가변 인자 : 함수에서 인자의 수와 자료형이 결정되지 않은 함수 인자 방식

```
//함수 printf()의 함수원형
int printf(const char *_Format, ...); //...이 무엇일까?

//함수 사용 예
printf("%d%d%f", 3, 4, 3.678); //인자가 총 4개
printf("%d%d%f%f%f", 7, 9, 2.45, 3.678, 8.98); //인자가 총 5개
```

- 처음 또는 앞부분의 매개변수는 정해졌으나 이후 매개변수 수와 각각의 자료형이 고정적X, 변화
- 매개변수에서 중간 이후부터 마지막에 위치한 가변 인자만 가능
- 함수 정의 시 가변 인자의 매개변수는 …으로 기술
- 가변 인자 … 시작 전 첫 고정 매개변수는 이후의 가변 인자 처리에 필요한 정보 지정에 사용

```
int vatest(int n, ...);
double vasum(char *type, int n, ...);
double vafun1(char *type, ..., int n); //오류, 마지막이 고정적일 수 없음
double vafun2(...); //오류, 처음부터 고정적일 수 없음
```

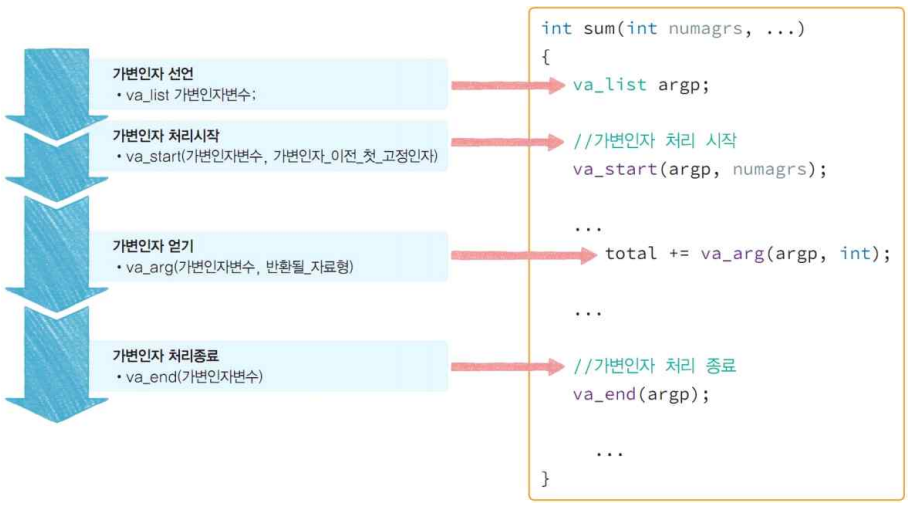
[가변 인자가 있는 함수 구현]

함수에서 가변 인자 구현 과정

- 필요 매크로 함수와 자료형을 위해 헤더파일 stdarg.h가 필요
- 가변 인자 선언 : 마치 변수 선언처럼 가변 인자로 처리할 변수를 하나 만드는 일
- 가변 인자 처리 시작 : 선언된 변수에서 마지막 고정 인자를 지정하여 가변 인자의 시작 위치를 알리는 방법
- 가변 인자 얻기 : 가변 인자 각각의 자료형을 지정하여 가변 인자를 반환 받는 절차
  매크로 함수 va_arg()의 호출로 반환된 인자로 원하는 연산을 처리
- 가변 인자 처리 종료 : 가변 인자에 대한 처리를 끝내는 단계

구분	처리 절차	설명
va_list argp;	① 가변 인자 선언	va_list로 변수 argp을 선언
va_start(va_list argp, prevarg)	② 가변 인자 처리 시작	va_start()는 첫 번째 인자로 va_list로 선언된 변수 이름 argp와 두 번째 인자는 가변 인자 앞의 고정 인자 prevarg를 지정하여 가변 인자 처리 시작
type va_arg(va_list argp, type)	③ 가변 인자 얻기	va_arg()는 첫 번째 인자로 va_start()로 초기화한 va_list 변수 argp를 받으며, 두 번째 인자로는 가변 인자로 전달된 값의 type을 기술
va_end(va_list argp)	④ 가변 인자 처리 종료	va_list로 선언된 변수 이름 argp의 가변 인자 처리 종료

가변 인자가 있는 함수 구현
- 가변 인자 앞의 첫 고정인자인 numargs는 가변 인자의 수
- int형인 가변 인자를 처리하여 그 결과를 반환하는 함수
- 가변 인자 ... 시작 전 첫 고정 매개변수는 이후의 가변 인자 처리에 필요한 정보 지정에 사용



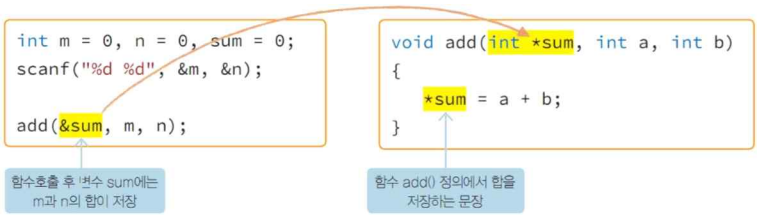
2. 포인터 전달과 반환

1) 매개변수와 반환으로 포인터 사용

[주소연산자 &]

함수에서 매개변수를 포인터로 이용하면 결국 참조에 의한 호출

- 함수 원형 void add(int*, int, int);에서 첫 매개변수가 포인터인 int*
- 함수 add()는 두 번째와 세 번째 인자를 합해 첫 번째 인자가 가리키는 변수에 저장 함수
- 변수인 sum을 선언하여 주소값인 &sum을 인자로 호출



[주소값 반환]

함수의 결과를 포인터로 반환

지역 변수 주소값의 반환은 문제를 발생시킬 수 있어 반환하지 않는 것이 바람직

```
int * add(int *, int, int);

   int m = 0, n = 0, sum = 0;
   ...
   scanf("%d %d", &m, &n);

   printf("두 정수 합: %d\n", *add(&sum, m, n));
```

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

2) 상수를 위한 const 사용

[키워드 const]

수정을 원하지 않는 함수의 인자 앞에 키워드 const를 삽입하여 참조되는 변수가 수정 X

- 키워드 const는 인자인 포인터 변수가 가리키는 내용을 수정 불가능
- constdouble *a와 constdouble *b로 기술, *a와 *b를 대입 연산자의 l-value로 사용 불가능
  → *a와 *b를 이용하여 그 내용 수정 불가능
- 상수 키워드 const의 위치는 자료형 앞이나 포인터 변수 *a 앞에도 가능
  → const double *a와 double const *a는 동일한 표현

```
// 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;
}
```

3) 함수의 구조체 전달과 반환

[복소수를 위한 구조체]

구조체 complex : 실수부와 허수부를 나타내는 real과 img를 멤버로 구성

```
struct complex
{
   double real; //실수
   double img;  //허수
};
typedef struct complex complex;
```

자료형 struct complex　　자료형 complex

자료형 struct complex와 complex 모두 복소수 자료형으로 사용 가능

※ TIP 복소수가 뭐예요?

- 실수의 개념을 확장한 수로 a+bi로 표현
- a와 b는 실수, I는 허수 단위로 i2 = -1을 만족하며 a는 실수부, b는 허수부
- 복소수에서의 사칙 연산

복소수의 합: (a+bi)+(c+di) = (a+b)+(c+d)i
복소수의 곱: (a+bi)*(c+di) = (ac-db)+(ad+bc)i
(a+bi)의 켤레 복소수: (a-bi)
(a-bi)의 켤레 복소수: (a+bi)

[인자와 반환형으로 구조체 사용]

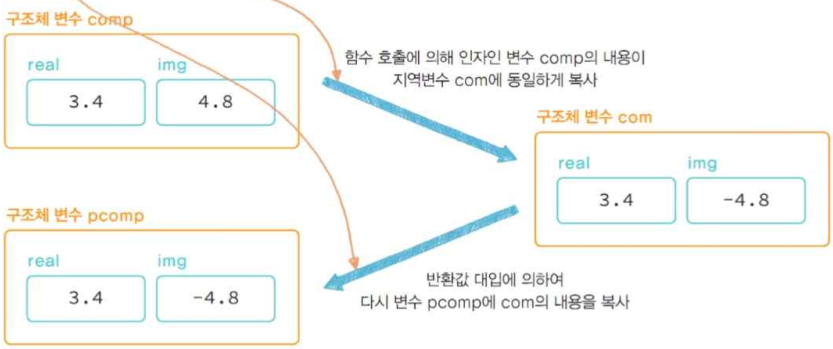
함수 paircomplex1()

- 인자인 복소수의 켤레 복소수 pair complex number를 구하여 반환하는 함수
- 구조체는 함수의 인자와 반환값으로 이용이 가능, 구조체 인자를 값에 의한 호출 방식으로 이용
- 함수에서 구조체 지역 변수 com을 하나 만들어 실인자의 구조체 값을 모두 복사하는 방식으로 구조체 값을 전달 받음

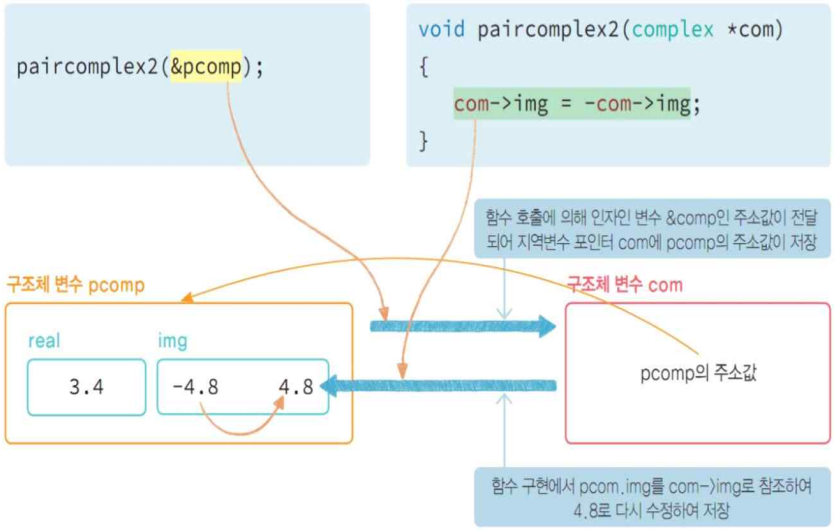
```
complex comp = { 3.4, 4.8 };
complex pcomp;

pcomp = paircomplex1(comp);
```

```
complex paircomplex1(complex com)
{
   com.img = -com.img;
   return com;
}
```



- 값에 의한 호출 방식 → 참조에 의한 호출 방식으로 수정
- paircomplex2()는 인자를 주소값으로 저장, 실인자의 변수 comp의 값을 직접 수정하는 방식
- 이 함수를 호출하기 위해서는 &pcomp처럼 주소값을 이용해 호출



구조체를 함수의 인자로 사용하는 방식은 값에 의한 호출, 참조에 의한 호출 둘 다 사용 가능

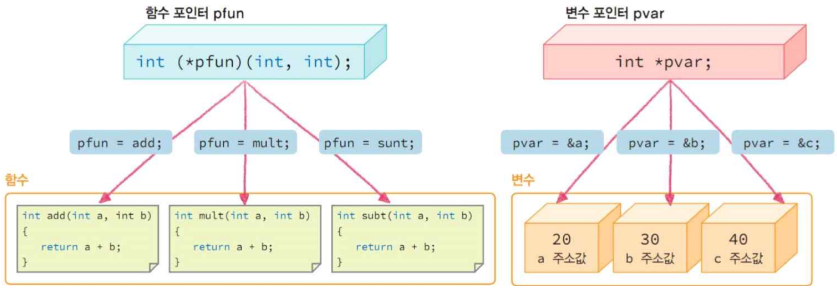
- 구조체 크기가 매우 큰 구조체를 값에 의한 호출의 인자로 사용한다면 매개변수의 메모리 할당과 값의 복사에 많은 시간이 소요
- 장점 : 주소값을 사용하는 참조에 의한 호출 방식은 메모리 할당과 값의 복사에 드는 시간 X

3. 함수 포인터와 void 포인터

1) 함수 포인터

[함수 주소 저장 변수]

- 포인터의 장점은 다른 변수를 참조하여 읽거나 쓰는 것도 가능
- 하나의 함수 이름으로 필요에 따라 여러 함수를 사용하면 편리
- 함수 포인터 pfun은 함수 add()와 mult() 그리고 subt()로도 사용 가능



함수 포인터 : 함수의 주소값을 저장하는 포인터 변수

- 반환형, 인자 목록의 수와 각각의 자료형이 일치하는 함수의 주소를 저장할 수 있는 변수
- 함수 포인터 선언 : 함수 원형에서 함수 이름을 제외한 반환형과 인자 목록의 정보가 필요

함수 포인터 변수 선언

```
반환자료형  (*함수 포인터변수이름)( 자료형1 매개변수이름1, 자료형2 매개변수이름2, ...);

또는

반환자료형  (*함수 포인터변수이름)( 자료형1, 자료형2, ...);
```

```
void add(double*, double, double);
void subtract(double*, double, double);
   ...
   void (*pf1)(double *z, double x, double y) = add;
   void (*pf2)(double *z, double x, double y) = subtract;
   pf2 = add;
```

- 함수 포인터 pf는 함수 add()의 주소 저장 가능
- 함수 원형이 void add(double*, double, double);인 함수의 주소를 저장
- 함수 원형에서 반환형인 void와 인자 목록인 (double *, double, double) 정보 필요

```
//잘못된 함수 포인터 선언
void *pf(double*, double, double);   //함수원형이 되어 pf는 원래 함수이름

   void (*pf)(double*, double, double); //함수 포인터
   pf = add; //변수 pf에 함수 add의 주소값을 대입 가능
```

주의할 점

- (*pf)와 같이 변수 이름인 pf 앞에는 *이 있어야 하며 반드시 괄호를 사용
- 만일 괄호가 없으면 pf는 함수 포인터 변수가 아니라 void *를 반환하는 함수 이름이 되고, 이 문장은 함수 원형이 되어 버림

- 함수 add()만을 가리키는 게 아니라 add()와 반환형과 인자 목록이 같은 함수는 가리킬 수 있음
- subtract()의 반환형과 인자 목록이 add()와 동일하다면 pf는 함수 subtract()도 가리킬 수 있음
- 문장 pf = subtract;와 같이 함수 포인터에는 괄호가 없이 함수 이름만으로 대입
- 함수 이름 add나 subtract는 주소연산자를 함께 사용하여 &add나 &subtract로도 사용 가능
- subtract()와 add()와 같이 함수 호출로 대입해서는 오류 발생

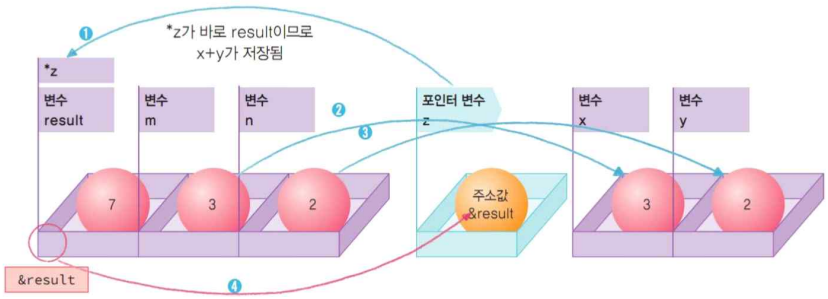
```
void (*pf2)(double *z, double x, double y) = add();      //오류발생
pf2 = subtract();                                        //오류발생
pf2 = add;                                               //가능
pf2 = &add;                                              //가능
pf2 = subtract;                                          //가능
pf2 = &subtract;                                         //가능
```

[함수 포인터를 이용한 함수 호출]

함수 인자를 포인터 변수로 사용하면 함수 내부에서 수정한 값이 그대로 실인자로 반영

```
double m, n, result = 0;
void (*pf)(double*, double, double);
....
pf = add;
pf(&result, m, n);  //add(&result, m, n);
//(*pf)(&result, m, n); //이것도 사용 가능
```

```
void add(double *z, double x, double y)
{
   *z = x + y;
}
```

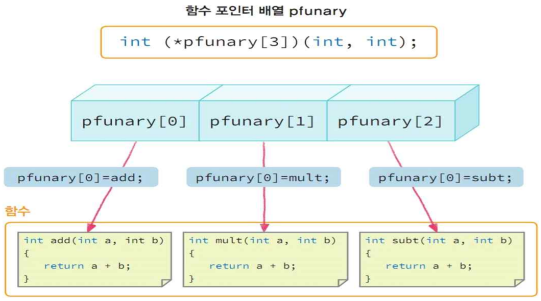


- 문장 pf = add;로 함수 포인터 변수인 pf에 함수 add()의 주소값이 저장, 변수 pf를 이용하여 add() 함수 호출 가능
- 포인터 변수 pf를 이용한 함수 add()의 호출 방법은 add() 호출과 동일
  → pf(&result, m, n);로 add(&result, m, n) 호출을 대체
- 함수 add()에서 m + n이 반영된 변수 result를 사용
- pf(&result, m, n)은 (*pf)(&result, m, n)로도 가능

2) 함수 포인터 배열

[함수 포인터 배열 개념]

함수 포인터 배열 : 원소로 여러 개의 함수 포인터를 선언 (함수 포인터가 원소인 배열)



[함수 포인터 배열 선언]

- 배열 fpary의 각 원소가 가리키는 함수는 반환값이 void, 인자목록은 (double*, double, double)

```
반환자료형  (*배열이름[배열크기])( 자료형1 매개변수이름1, 자료형2 매개변수이름2, ...);

또는

반환자료형  (*배열이름[배열크기])( 자료형1, 자료형2, ...);
```

```
void add(double*, double, double);
void subtract(double*, double, double);
void multiply(double*, double, double);
void devide(double*, double, double);
   ...
   void (*fpary[4])(double*, double, double);
```

- 배열 fpary을 선언한 이후에 함수 4개를 각각의 배열 원소에 저장
- 배열 fpary을 선언하면서 함수 4개의 주소값을 초기화하는 문장

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

배열의 선언과 초기화 문장으로 간단히 처리

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

3) void 포인터

[void 포인터 개념]

void 포인터 : 자료형을 무시하고 주소값[3]만을 다루는 포인터

- 대상에 상관없이 모든 자료형의 주소를 저장할 수 있는 만능 포인터로 사용 가능
- 일반 포인터는 물론 배열과 구조체, 함수 주소도 저장 가능

```
char ch = 'A';
int data = 5;
double value = 34.76;

void *vp;           //void 포인터 변수 vp 선언

vp = &ch;           //ch의 주소 만을 저장
vp = &data;         //data의 주소 만을 저장
vp = &value;        //value의 주소 만을 저장
```

[void 포인터 활용]

- 모든 주소를 저장 가능, 가리키는 변수를 참조하거나 수정은 불가능
- void 포인터는 자료형 정보는 없이 임시로 주소만을 저장하는 포인터
  → 실제 void 포인터로 변수를 참조하기 위해서는 자료형 변환이 필요

```
int m = 10;     double x = 3.98;

void *p = &m;
int n = *(int *)p; //int * 로 변환
n = *p; //오류
          오류: "void" 형식의 값을 "int" 형식의 엔터티에 할당할 수 없습니다.
p = &x;
int y= *(double *)p; //double * 로 변환
y = *p; //오류
          오류: "void" 형식의 값을 "int" 형식의 엔터티에 할당할 수 없습니다.
```

3) 주소값 : 참조를 시작하려는 주소, 자료형을 알아야 참조할 범위와 내용을 해석할 방법을 알 수 있음

5 파일 처리

1. 파일 기초

1) 텍스트 파일과 이진 파일

2) 파일 스트림 열기

2. 텍스트 파일 입출력

1) 파일에 서식화된 문자열 입출력

2) 파일 문자열 입출력

3) 파일 문자 입출력

3. 이진 파일 입출력

1) 텍스트와 이진 파일 입력과 출력

2) 구조체의 파일 입출력

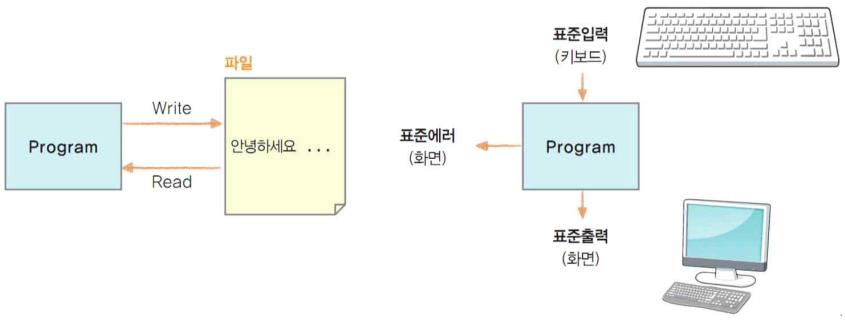
1. 파일 기초

1) 텍스트 파일과 이진 파일

[파일의 필요성]

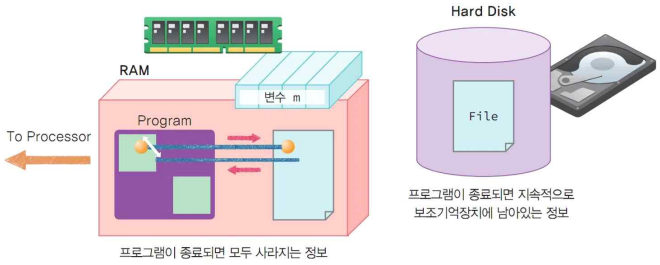
프로그램에서 출력을 파일에 한다면 → 파일 생성

키보드에서 표준 입력하던 입력을 파일에서 입력하면 → 파일 입력



파일과 메모리

- 프로그램이 종료되면 주기억 장치의 메모리 공간은 모두 사라짐
- 보조기억장치인 디스크에 저장되는 파일 직접 삭제하지 않는 한 프로그램 종료 후에도 계속 저장
- 종료 후에도 계속 사용하고 싶다면 프로그램에서 파일에 그 내용을 저장



[텍스트 파일과 이진 파일]

파일

- 보조기억장치의 정보 저장 단위로 자료의 집합 (텍스트 파일과 이진 파일 두 가지 유형)

텍스트 파일

- 메모장 같은 편집기로 작성된 파일
- 내용이 아스키코드와 같은 문자 코드값으로 저장
- 메모리에 저장된 실수와 정수와 같은 내용도 문자 형식으로 변환되어 저장
- 텍스트 편집기를 통하여 그 내용을 볼 수 있고 수정 가능

이진파일

- 목적에 알맞은 자료가 이진 형태로 저장되는 파일 (실행파일과 그림파일, 음악파일, 동영상파일 등)
- 컴퓨터 내부 형식으로 저장되는 파일, 입출력 속도도 텍스트 파일보다 빠름
- 자료는 메모리 자료 내용에서 어떤 변환도 거치지 않고 그대로 파일에 기록
- 텍스트 편집기(메모장)로는 그 내용을 볼 수 없고 그 내용을 이미 알고 있는 특정한 프로그램에 의해 인지될 때 의미가 있음

[입출력 스트림]
자료의 입력과 출력은 자료의 이동, 자료가 이동하려면 이동 경로가 필요
- 입출력 스트림 : 입출력 시 이동 통로
- 표준입력 스트림 : 키보드에서 프로그램으로 자료가 이동 경로
- 함수 scanf() : 표준입력 스트림에서 자료를 읽을 수 있는 함수
- 표준출력 스트림 : 프로그램에서 모니터의 콘솔로 자료가 이동 경로
- 함수 printf() : 표준출력 스트림으로 자료를 보낼 수 있는 함수

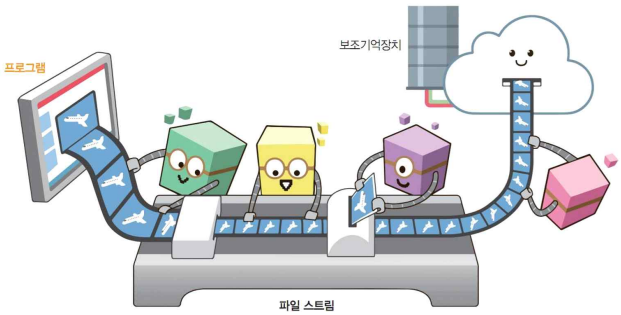
입력 스트림 : 다른 곳에서 프로그램으로 들어오는 경로
- 자료가 떠나는 시작 부분이 자료 원천부

원천부	입력
키보드	표준입력
파일	파일 입력 : 파일로부터 자료 읽음
터치스크린	스크린 입력 : 스크린에서 터치 정보 읽음
네트워크	네트워크 입력 : 다른 곳에서 프로그램으로 네트워크를 통해 자료 전달

출력 스트림 : 프로그램에서 다른 곳으로 나가는 경로
- 자료가 도착 장소가 자료 목적부

목적부	출력
콘솔	표준출력
파일	파일 출력 : 파일에 원하는 값 저장
프린터	프린터 출력 : 프린터에서 출력물
네트워크	네트워크 출력 : 네트워크 출력이 되어 다른 곳으로 자료 이동

[파일 스트림 이해]
파일 스트림
- 보조기억장치의 파일과 프로그램을 연결하는 전송 경로
- 파일 입력 스트림 : 파일에서 프로그램으로 자료의 입력을 위한 스트림
- 파일 출력 스트림 : 프로그램에서 파일로 출력을 위한 스트림
- 파일 스트림을 만들기 위해서는 특정한 파일 이름과 1)파일모드가 필요



1) 파일모드 : 입력 또는 출력과 같은 스트림의 특성

2) 파일 스트림 열기

[함수 fopen()으로 파일 스트림 열기]

프로그램에서 특정한 파일과 파일 스트림을 연결하기 위해서는 fopen() 또는 fopen_s()를 이용

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

- 함수 fopen()은 파일명 _Filename의 파일 스트림을 모드 _Mode로 연결하는 함수이며, 스트림 연결에 성공하면 파일 포인터를 반환하며, 실패하면 NULL을 반환한다.
- 함수 fopen_s()는 스트림 연결에 성공하면 첫 번째 인자인 _File에 파일 포인터가 저장되고 정수 0을 반환하며, 실패하면 양수를 반환한다. 현재 Visual C++에서는 함수 fopen_s()의 사용을 권장하고 있다.

- FILE : 헤더 파일 stdio.h에 정의되어 있는 구조체 유형
- 함수 fopen()의 반환값 유형 FILE *은 구조체 FILE의 포인터 유형
- 함수 fopen()은 인자가 파일 이름과 파일 열기 모드
- 파일 스트림 연결에 성공하면 파일 포인터를 반환, 실패하면 NULL을 반환

파일 스트림 연결에 성공하면 정수 0을 반환, 만일 스트림 연결에 실패하면 양수를 반환

- 첫 번째 인자는 파일 포인터의 주소값
- 두 번째 인자인 문자열은 처리하려는 파일 이름
- 세 번째 문자열은 파일 열기 종류인 모드

  읽기 모드 r : 읽기가 가능한 모드, 쓰기는 불가능

  쓰기 모드 w : 파일 어디에든 쓰기가 가능한 모드, 읽기는 불가능

  추가 모드 a : 파일 중간에 쓸 수 없으며 파일 마지막에 추가적으로 쓰는 것만 가능한 모드

  읽기는 불가능, 파일에 쓰는 내용은 무조건 파일 마지막에 추가

```
if ( fopen_s(&f, "basic.txt", "w") != 0 )
//if ( (f = fopen(fname, "w")) == NULL )
{
   printf( "파일이 열리지 않습니다.\n" );
   exit(1);
};
...
fclose(f);
```

[함수 fclose()로 파일 스트림 닫기]

- fopen()으로 연결한 파일 스트림을 닫는 기능을 수행 (성공하면 0을 실패하면 EOF을 반환)
- 파일 스트림을 연결한 후 파일 처리가 모두 끝났으면 파일 포인터 f를 인자로 함수 fclose()를 호출하여 반드시 파일을 닫도록
- 내부적으로 파일 스트림 연결에 할당된 자원을 반납, 파일과 메모리 사이에 있던 버퍼의 내용을 모두 지우는 역할을 수행

```
int fclose(FILE * _File);
```

함수 fclose()는 파일 스트림 f를 닫는 함수로서, 성공하면 0을 실패하면 EOF을 반환한다.

```
fclose(f);
```

2. 텍스트 파일 입출력

1) 파일에 서식화된 문자열 입출력

[함수 fprintf()와 fscanf()]

텍스트 파일에 자료를 쓰거나 읽기 위하여 fprintf()와 fscanf() 또는 fscanf_s()를 이용

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

위 함수에서 _File은 서식화된 입출력 스트림의 목적지인 파일이며, _Format은 입출력 제어 문자열이며, 이후 기술되는 인자는 여러 개의 출력될 변수 또는 상수이다.

- 첫 번째 인자는 입출력에 이용될 파일
- 두 번째 인자는 입출력에 이용되는 제어 문자열
- 다음 인자들은 입출력될 변수 또는 상수 목록
- 함수 fprintf()와 fscanf() 또는 fscanf_s()의 첫 번째 인자에 각각 stdin 또는 stdout를 이용하면 표준입력, 표준출력으로 이용이 가능

표준 파일	키워드	장치(device)
표준입력	stdin	키보드
표준출력	stdout	모니터 화면
표준에러	stderr	모니터 화면

[표준입력 자료를 파일에 쓰기]

- 파일에 내용 쓰기와 반대로 파일로부터 내용을 읽는 프로그램을 작성
- 쓰기 모드로 파일을 열어 표준입력으로 받은 학생 이름과 중간점수, 기말점수를 파일에 기록 후 닫기
- 다시 읽기 모드로 그 파일을 열어 기록된 내용을 읽어와 표준출력으로 출력하는 프로그램
- 함수 scanf_s()에서 표준입력을 빈칸으로 구분하므로 빈칸이 없는 이름을 입력
- 학생 이름, 중간점수, 기말점수를 함수 fprintf()를 이용하여 파일 f에 출력
- 변수 cnt는 이름 앞에 번호를 붙이려는 목적으로 이용

```
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);
```

- 파일 f에서 fscanf_s()를 이용하여 구조체의 정보를 입력하려면 파일의 내부 자료의 저장 형태 필요
- 학생 이름, 중간, 기말이므로 다음과 같이 문장으로 fscanf_s()를 사용

```
                  문자열이 저장되는 name과 그 크기를 지정해야 한다.
                  ↓
//파일"grade.txt"에서 읽기
fscanf_s(f, "%d %s %d %d\n", &cnt, name, 30, &point1, &point2);
```

2) 파일 문자열 입출력

[함수 fgets()와 fputs()]

함수 fgets() : 파일로부터 한 행의 문자열을 입력 받는 함수

- 파일로부터 문자열을 개행 문자 (\n)까지 읽어 마지막 개행 문자를 '\0' 문자로 바꾸어 입력 버퍼 문자열에 저장
- 첫 번째 인자 : 문자열이 저장될 문자 포인터
- 두 번째 인자 : 입력할 문자의 최대 수
- 세 번째 인자 : 입력 문자열이 저장될 파일

함수 fputs() : 파일로 한 행의 문자열을 출력하는 함수

- 문자열을 한 행에 출력
- 첫 번째 인자 : 출력될 문자열이 저장된 문자 포인터
- 두 번째 인자 : 문자열이 출력되는 파일

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

- 함수 fgets()는 _File로부터 한 행의 문자열을 _MaxCount 수의 _Buf 문자열에 입력 수행
- 함수 fputs()는 _Buf 문자열을 _File에 출력 수행

```
char names[80];
FILE *f;

fgets(names, 80, f);
fputs(names, f);
```

[함수 feof()와 ferror()]

함수 feof() : 파일 스트림의 EOF(End Of File) 표시를 검사하는 함수

- 읽기 작업이 파일의 이전 부분을 읽으면 0을 반환, 그렇지 않으면 0이 아닌 값을 반환
- 파일 스트림의 EOF은 이전 읽기 작업에서 EOF 표시에 도달하면 0이 아닌 값으로 지정
- 단순히 파일 지시자가 파일의 끝에 있더라도 feof()의 결과는 0

함수 ferror() : 파일 처리에서 오류가 발생 했는지 검사하는 함수

- 이전 파일처리에서 오류가 발생하면 0이 아닌 값을 반환, 오류가 발생하지 않으면 0을 반환

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

- 함수 feof()은 _File의 EOF를 검사
- 함수 ferror()는 _File 에서 오류발생 유무를 검사

```
while ( !feof(stdin) )
{
   ...
   fgets(names, 80, stdin);    //표준입력
}
```

3) 파일 문자 입출력

[함수 fgetc()와 fputc()]

함수 fgetc()와 getc()

- 파일로부터 문자 하나를 입력 받는 함수

함수 fputc()와 putc()

- 문자 하나를 파일로 출력하는 함수
- 함수들은 문자 하나의 입출력의 대상인 파일 포인터를 인자로 이용

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

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

- 함수 fgetc()와 getc()는 _File에서 문자 하나를 입력받는 함수
- 함수 fputc()와 putc()문자 _Ch를 파일 _File 에 출력하는 함수

getchar()와 putchar()

- getc()와 putc()를 이용한 매크로로 정의
- getchar()와 putchar()는 함수로도 구현

```
#define  getchar()        getc(stdin)
#define  putchar(_c)      putc((_c),stdout)

int  getchar(void);
int  putchar(int _Ch);
```

[파일 내용을 표준출력으로 그대로 출력]

- 도스 명령어 type와 같이 파일의 내용을 그대로 콘솔에 출력하는 프로그램
- 파일 filename의 내용을 표준출력하는 프로그램
- 명령행 인자에서 두 번째 인자가 파일 이름에 해당
- 파일 내용의 출력은 한 줄마다 맨 앞에 줄 번호를 출력

```
D:\Creative C Sources\Ch15\Debug>list ../Prj05/listpart.c
 1: // file: listpart.c
 2:
 3: #include <stdio.h>
 4: #include <stdlib.h>
 5:
 6: int main(int argc, char *argv[])
 7: {
 8:    FILE *f;
 9:    int ch, cnt = 0;
10:    ...
11:    return 0;
12: }
```

두 번째 인자인 '../list/listpart.c'를 파일 이름으로 이 파일의 내용을 출력한다.

줄 번호와 함께 파일 내용을 그대로 출력한다.

3. 이진 파일 입출력

1) 텍스트와 이진 파일 입력과 출력

[함수 fprintf()와 fscanf_s()]

함수 fprintf()와 fscanf(), fscanf_s()는 자료의 입출력을 텍스트 모드로 처리

- 출력된 텍스트 파일은 텍스트 편집기로 그 내용을 볼 수 있음
- 텍스트 파일의 내용은 모두 지정된 아스키 코드와 같은 문자 코드 값
- 그 내용을 확인할 수 있을 뿐만 아니라 인쇄 가능

```
int cnt = 10;

fprintf(f, "%d", cnt);
```

- 실제로 파일에 저장되는 자료는 정수값 10에 해당하는 각 문자의 아스키값
- 각각의 문자 '1'과 '0'의 아스키 코드값이 저장

[함수 fwrite()와 fread()]

- 이진 파일은 C 언어의 자료형을 모두 유지하면서 바이트 단위로 저장되는 파일
- 이진 모드로 블록 단위 입출력을 처리

```
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);
```

- 함수 fwrite()는 ptr이 가리키는 메모리에서 size만큼 n개를 파일 f에 쓰는(저장) 함수
- fread()는 반대로 파일 f에서 elmtsize의 n개만큼 메모리 dstbuf에 읽어오는 함수, 반환값은 성공적으로 입출력을 수행한 항목의 수

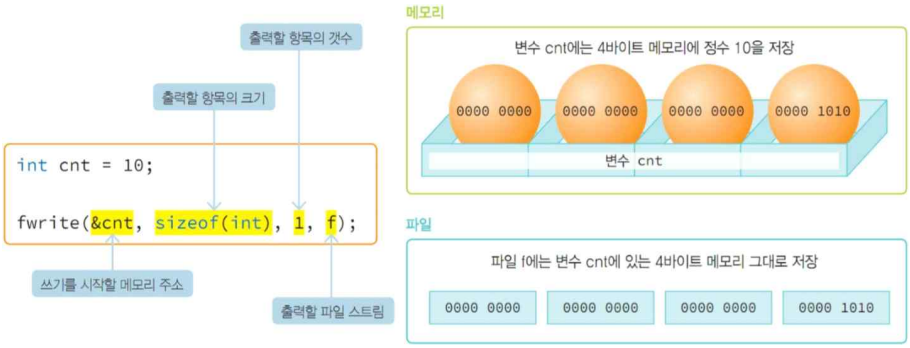
```
int cnt = 10;
fwrite(&cnt, sizeof(int), 1, f);
fread(&cnt, sizeof(int), 1, f);
```

함수 fwrite()

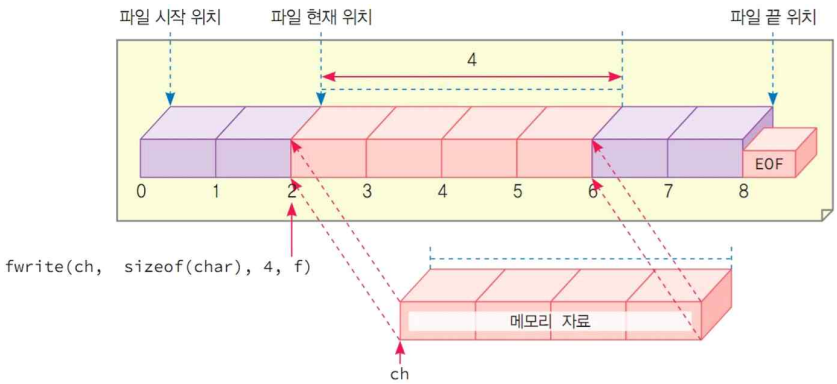
- 첫 번째 인자 ptr은 출력될 자료의 주소값, 두 번째 인자 size는 출력될 자료 항목의 바이트 크기
- 세 번째 인자는 출력될 항목의 개수, 마지막 인자는 출력될 파일 포인터
- 파일 f에 ptr에서 시작해서 size*n 바이트 만큼의 자료를 출력, 반환값은 출력된 항목의 개수

함수 fread()

- 이진 파일에 저장되어 있는 자료를 입력
- 함수 fwrite()와 인자는 동일



- 함수 fwrite()는 바이트 단위로 원하는 블록을 파일에 출력하기 위한 함수
- fwrite()에서 출력된 자료는 함수 fread()로 입력해야 그 자료 유형 유지 가능



- 세 번째 항목인 출력 항목 수를 4로 지정한 경우의 출력

이진 파일을 위한 열기 모드는 문자 'b'를 추가 ('b'가 없으면 기본 텍스트 파일 의미)

모드	의미
rb	이진 파일의 읽기(read) 모드로 파일을 연다.
wb	이진 파일의 쓰기(write) 모드로 파일을 연다.
ab	이진 파일의 추가(append) 모드로 파일을 연다.

2) 구조체의 파일 입출력

[학생 성적 구조체 파일 쓰기]

- 학생의 성적 정보를 구조체로 표현
- 번호, 이름, 중간, 기말, 퀴즈 점수를 멤버로 구성

```
struct personscore
{
   int number;          //번호
   char name[40];       //이름
   int mid;             //중간성적
   int final;           //기말성적
   int quiz;            //퀴즈성적
};
typedef struct personscore pscore;
```

- 표준입력으로 여러 명의 자료를 입력받은 구조체 자료형을 파일 "score.bin"에 저장하는 프로그램
- 표준입력은 사람마다 한 행씩 입력받도록 입력된 학생 수로 학생 번호를 입력
- 행마다 fgets()를 이용하여 하나의 문자열로 받고 입력된 문자열에서 각 구조체의 멤버 자료를 추출
- 문자열에서 자료를 추출하기 위하여 함수 sscanf()를 이용

[파일에서 학생 성적 구조체 읽기]
- 예제에서 만든 이진 파일 score.bin의 내용을 읽어 표준출력하는 프로그램을 작성
- 함수 feof()을 이용하여 파일의 마지막까지 구조체 자료를 읽어 적당한 출력 형태가 되도록 함수 fprintf()와 prinf()를 이용하여 출력
- 예제를 실행하려면 예제에서 생성된 파일 score.bin을 반드시 이 프로젝트 하부 폴더에 복사