

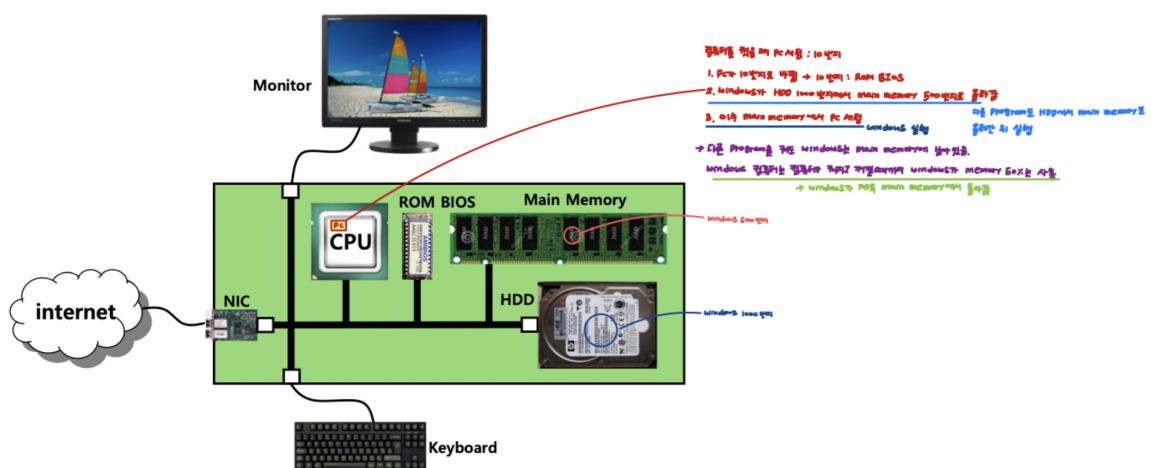
# Introduction to System Programming (CSE2018)

▼ 09/05 목 - Pdf 1

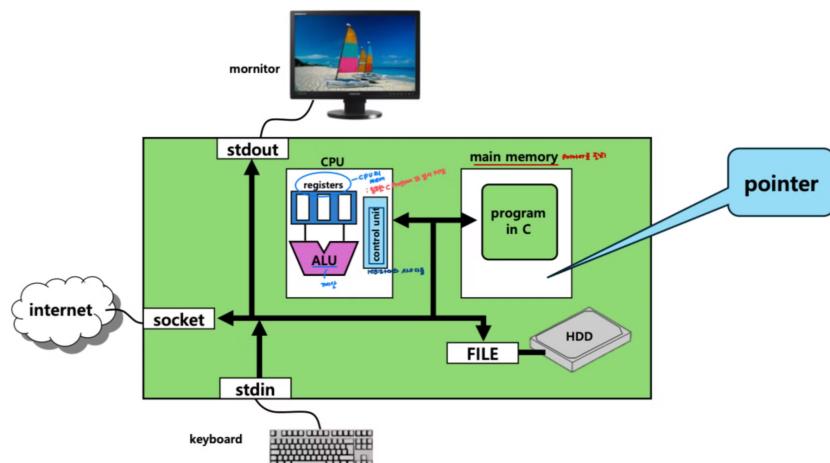
## C언어를 위한 컴퓨터 구조와 코드 생성 및 실행과정

### 컴퓨터 구조

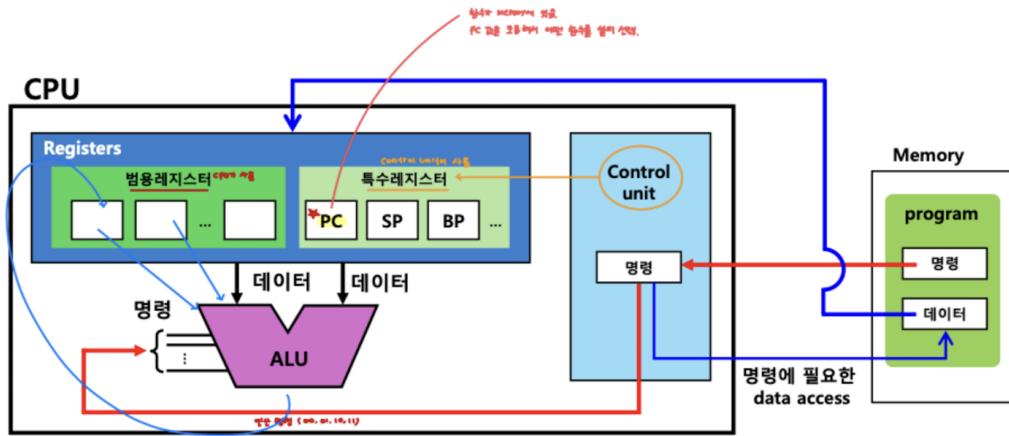
- 컴퓨터 구조



- C언어에서 바라본 컴퓨터 구조

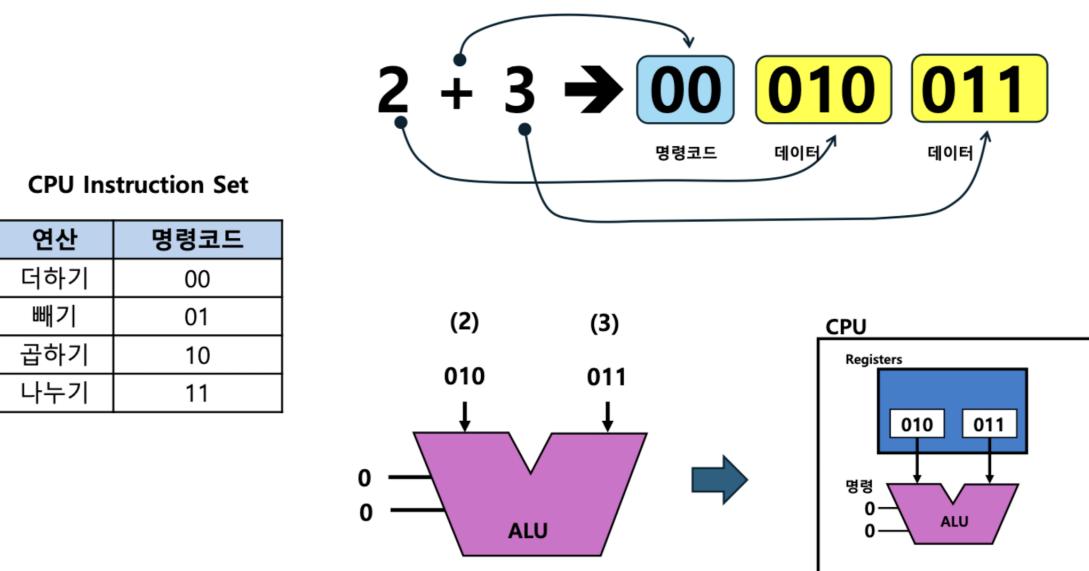


- CPU 구조



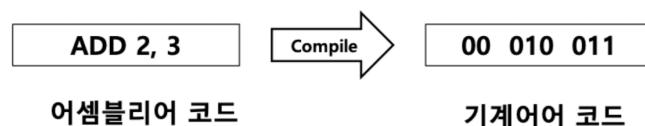
## 기계어, 어셈블리어 그리고 고급언어

- 기계어 코드



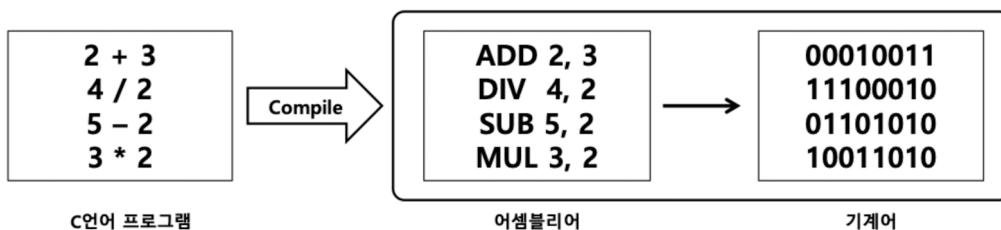
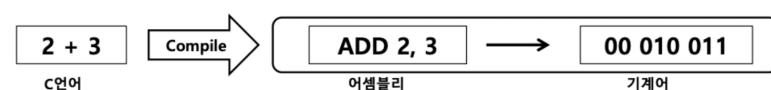
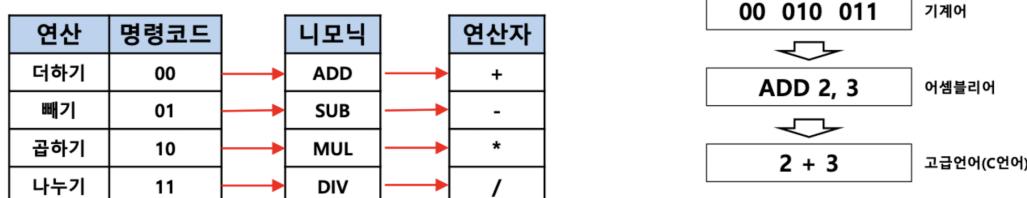
- 어셈블리어 코드

### Instruction Set



- 고급언어(C언어) 코드

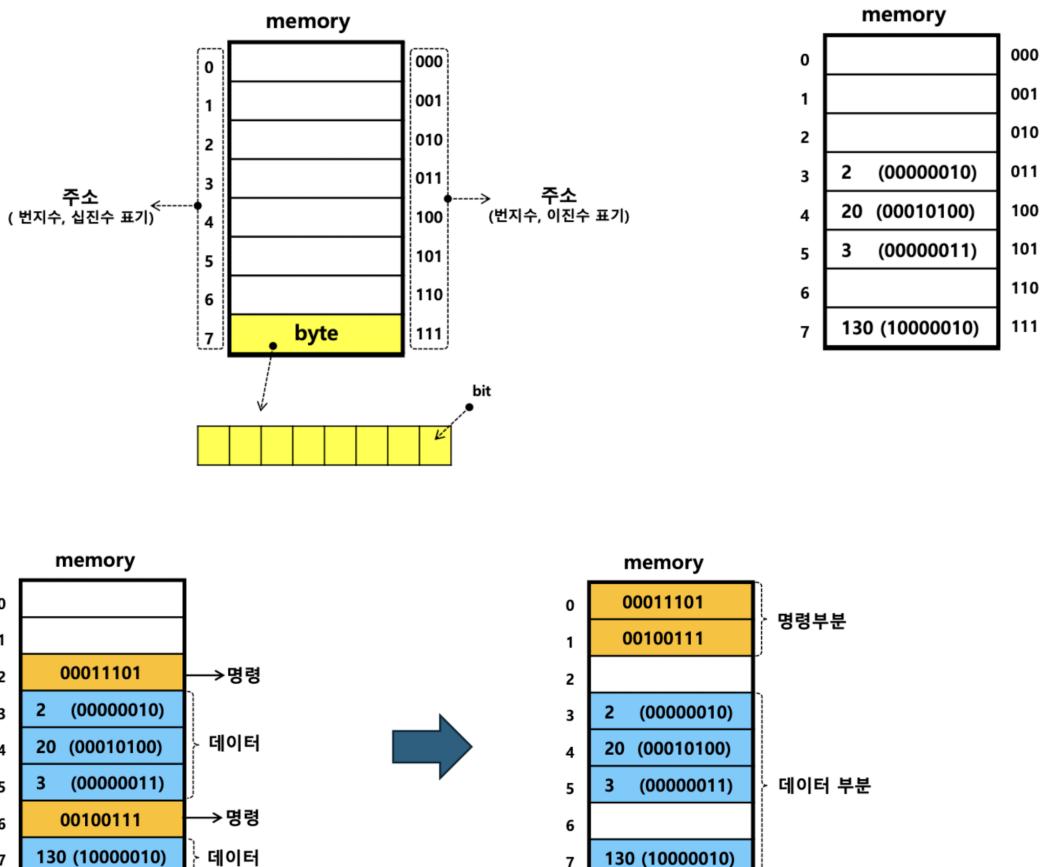
### Instruction Set



## 메모리 구조

- 메모리 구조

## 8 바이트 메모리

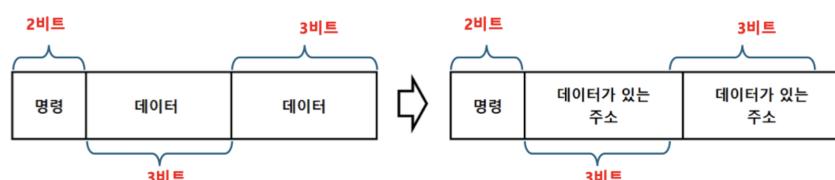


**명령과 데이터 혼재**  
 → 명령이 불규칙적으로 배치  
 → CPU가 차례차례로 명령을 가져와 실행할 수 없다!!!

**명령과 데이터를 분리**  
 → 메모리를 명령이 들어가는 부분과 데이터가 들어가는 부분으로 분리  
 → CPU가 차례차례로 명령을 가져와 실행할 수 있다!!!

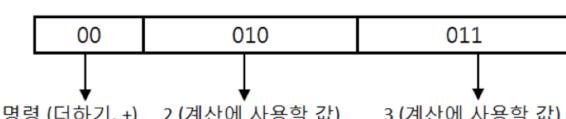
- 메모리를 효율적으로 사용하기 위한 명령어 구조

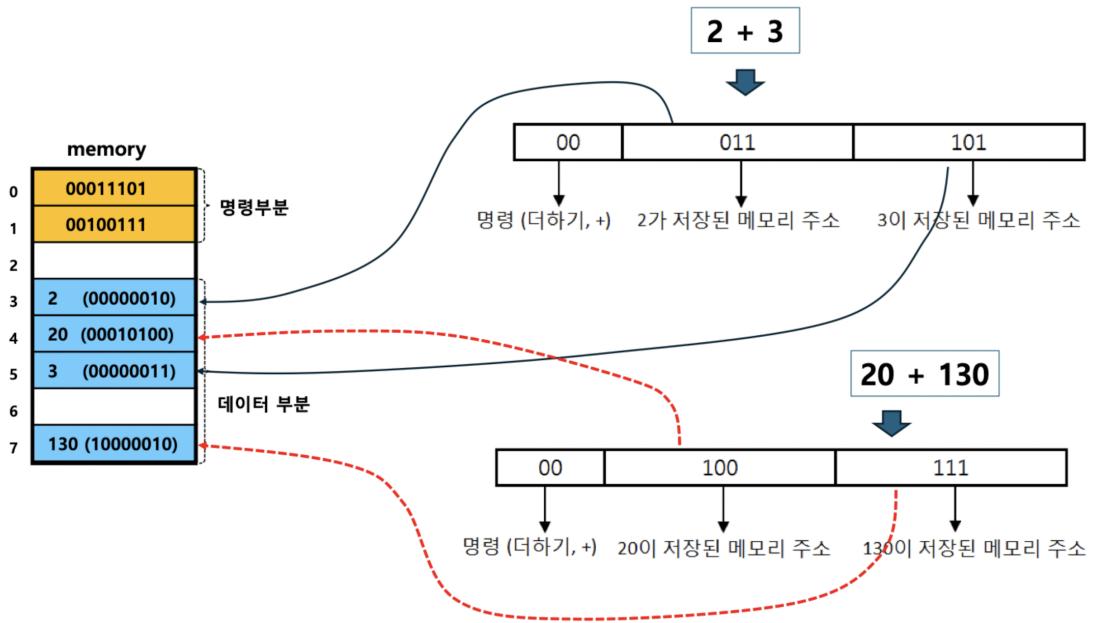
명령어의 길이가 1 바이트(8 비트) 라고 하면



3 비트로 표현할 수 있는 데이터 값은 0부터 7까지  
 → 사용할 수 있는 수의 범위에 제약이 생김  
 → 2 + 10을 기계어로 표현할 수 없음

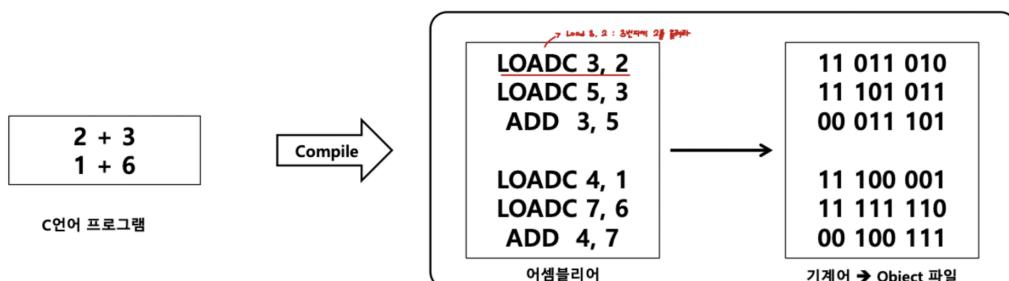
3 비트로 표현할 수 있는 주소는 0부터 7번지까지  
 → 한 번지는 8비트이므로 0~ 255까지 데이터 값을 사용 가능





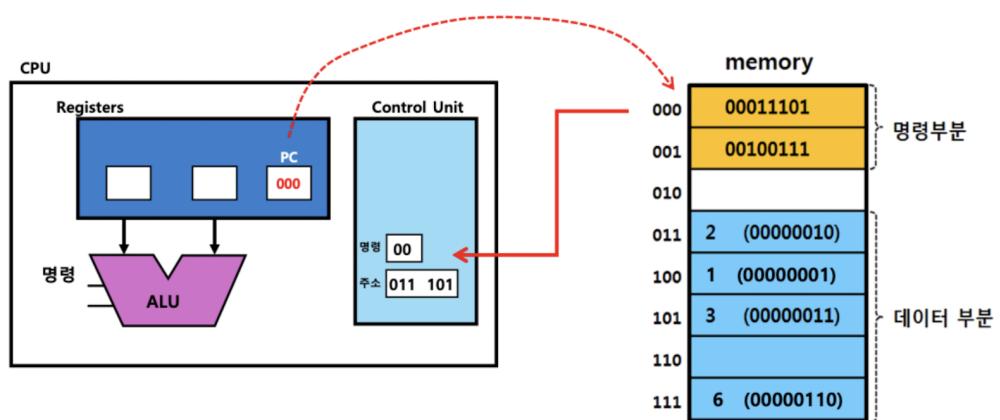
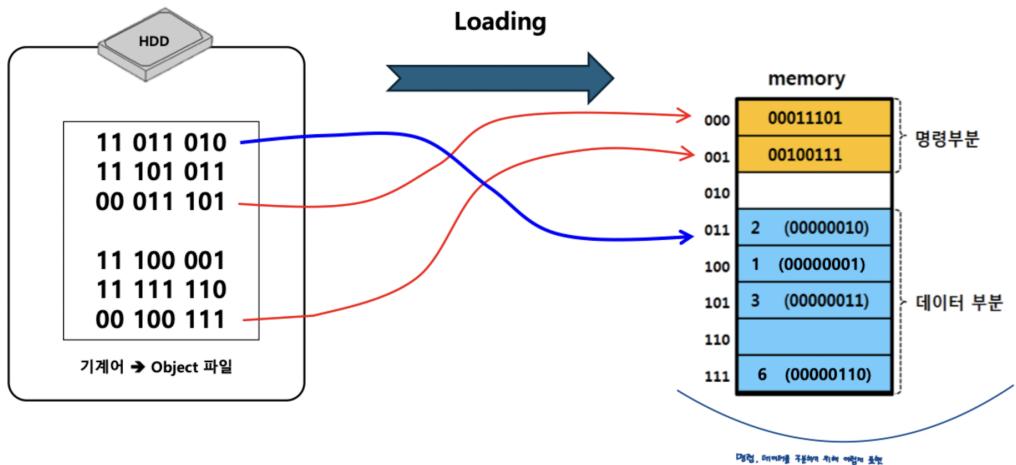
## 소스코드 작성과 실행과정

- 소스코드 작성과 컴파일



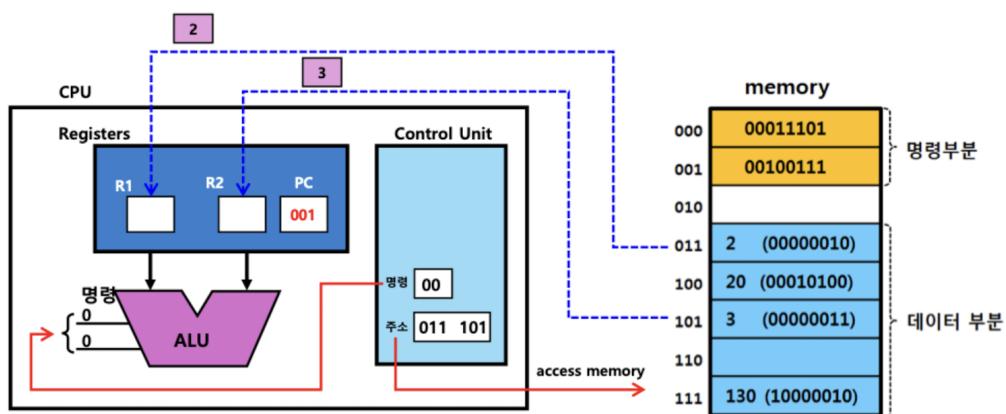
- LOADC `addr, value` → `addr` 주소(번지)에 `value` 저장  
LOADC의 기계어 코드 → 11이라고 가정
- ADD `addr1, addr2` → `addr1` 번지와 `addr2` 번지에 있는 값을 더하기  
ADD의 기계어 코드 → 00이라고 가정

- 프로그램 실행



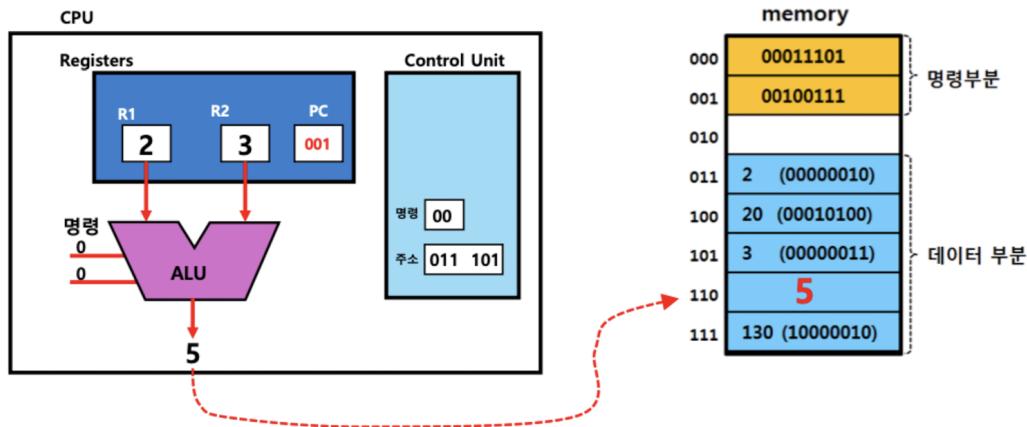
1. PC 레지스터에 있는 주소에서 명령가져오기

2. PC 값 증가 (다음 명령을 가져오기 위함)



1. CU이 명령해석  $\Rightarrow$  ALU에 명령어 셋팅

2. 메모리에서 데이터 가져와 범용레지스터 R1과 R2에 저장



1. 명령실행

2. 계산 결과를 메모리에 저장

## 실행파일 생성 과정 (라이브러리와 링킹)

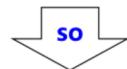
앞의 예의  $2+3$ 의 계산 결과 5를 모니터 화면에 출력하려면 어떻게 해야할까?

→ 메모리의 내용을 비디오 카드로 보내서 모니터 화면에 출력하도록 프로그램을 작성해야 한다.

→ 이 작업은 실제 우리가 사용하는 컴퓨터 하드웨어를 모두 알아야 작성할 수 있다.

→ 우리는 하드웨어에 대해서 잘 모르니까 화면에 결과를 출력할 수 없다.

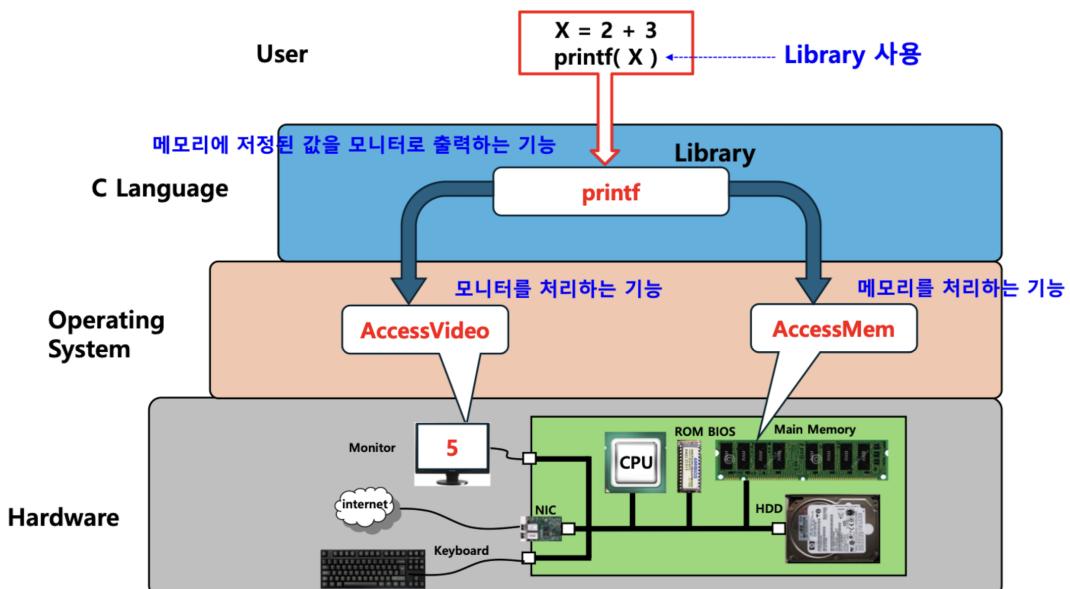
그럼 어떻게 해야하는가?

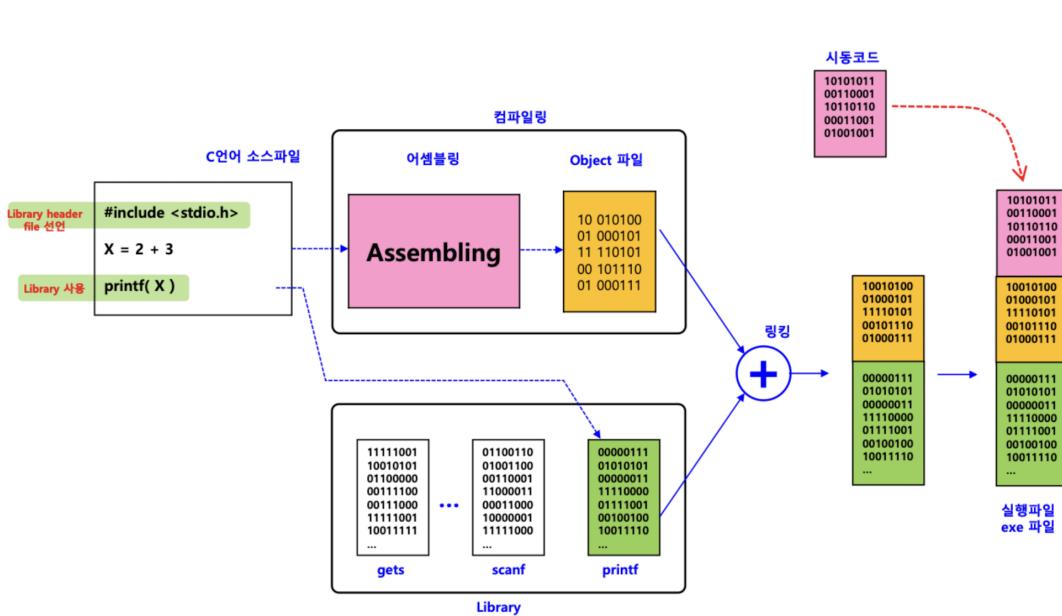
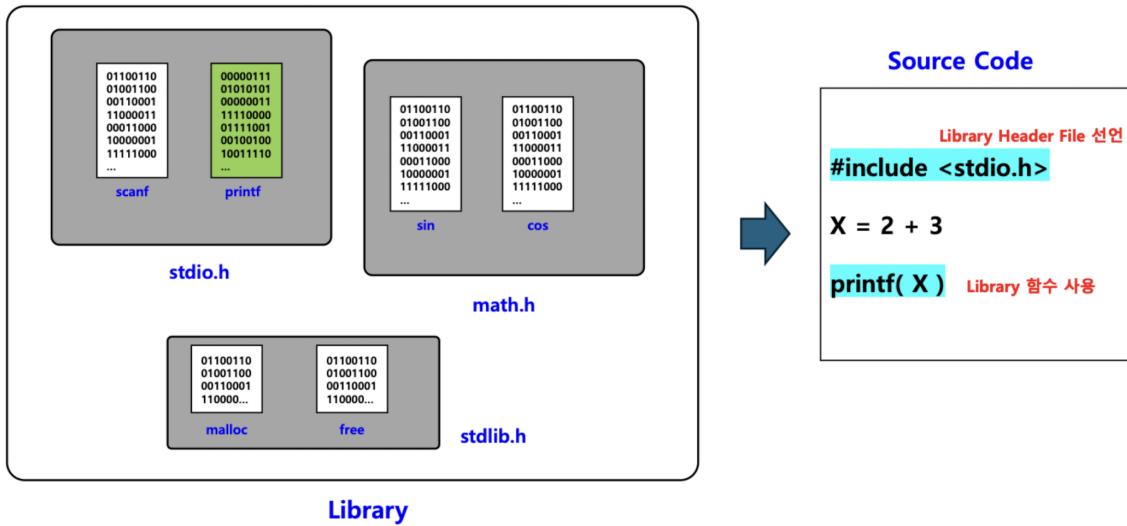


메모리에 저장된 내용을 화면에 출력하는 다른 프로그램을 [연결\(link\)](#)해서 이용할 수 있다.

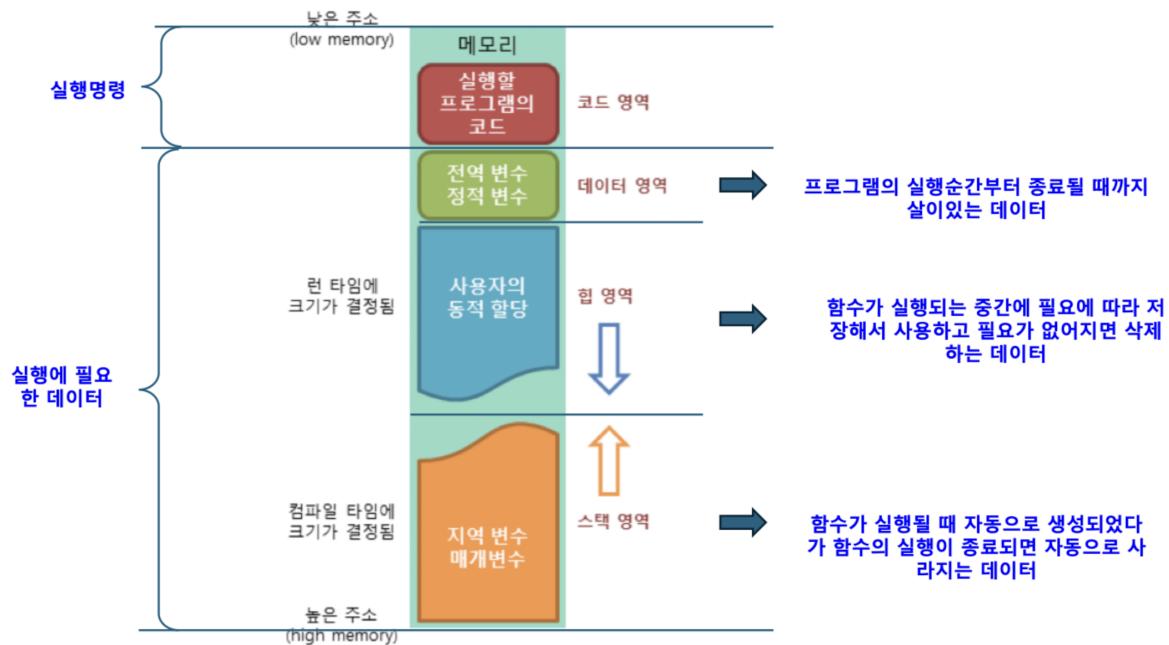
→ 이럴 때 사용하는 다른 사람이 작성해 놓은 프로그램을 [라이브러리\(library\)](#)라고 한다.

→ 실생활에서 도서관에 가면 다른 사람이 써놓은 책이 있고 그것을 우리가 읽고 지식을 얻는 것과 같은 원리이다.





## C언어 메모리 모델 (C언어가 메모리를 사용하는 방법)



▼ 09/12 목 - Pdf 2, 1장

## 정수와 실수 (보수와 Overflow, 부동소수점 실수)

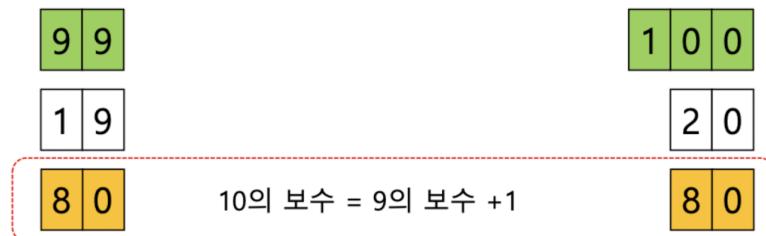
### 정수

- 정수 리터럴
  - 정수를 표현하는 방법
  - 소수점 또는 10의 거듭제곱 없이 숫자 문자로만 표현된 숫자
  - 즉, 소수 부분이 없는 숫자 (예: 134)
- 보수 : 특정한 수를 만들기 위한 보충해야하는 수

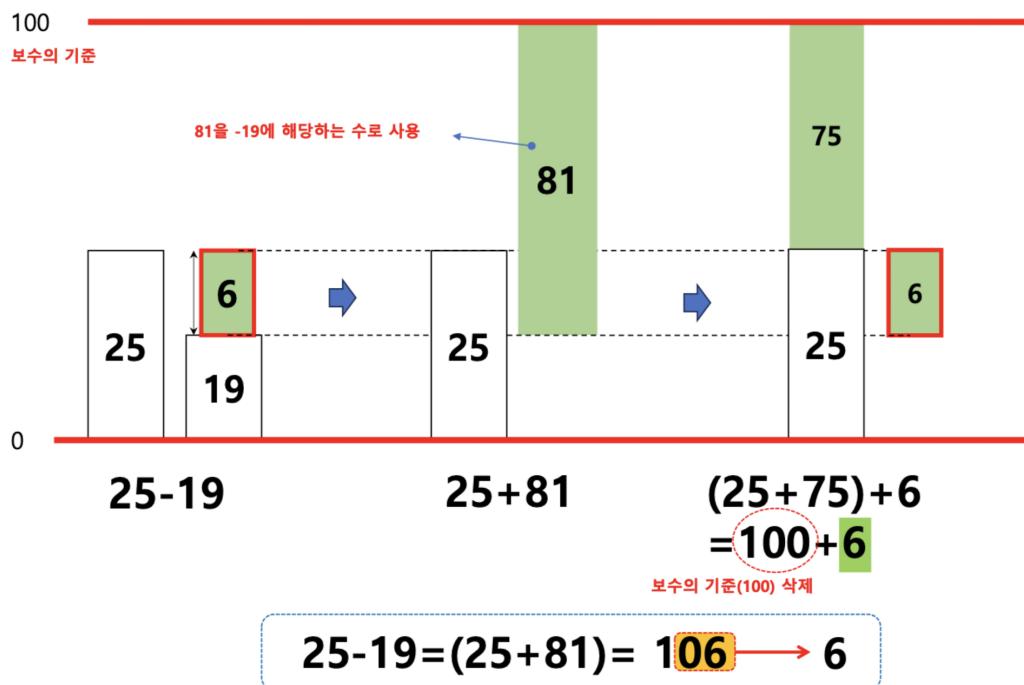
## [ 10 진수 ]

19에 대한 9의 보수 (19에 대한 99의 보수)  
 ➔ 99를 만들기 위해 19에 보충해주어야 하는 수  
 ➔ 80

19에 대한 10의 보수 (19에 대한 100의 보수)  
 ➔ 100을 만들기 위해 19에 보충해주어야 하는 수  
 ➔ 81



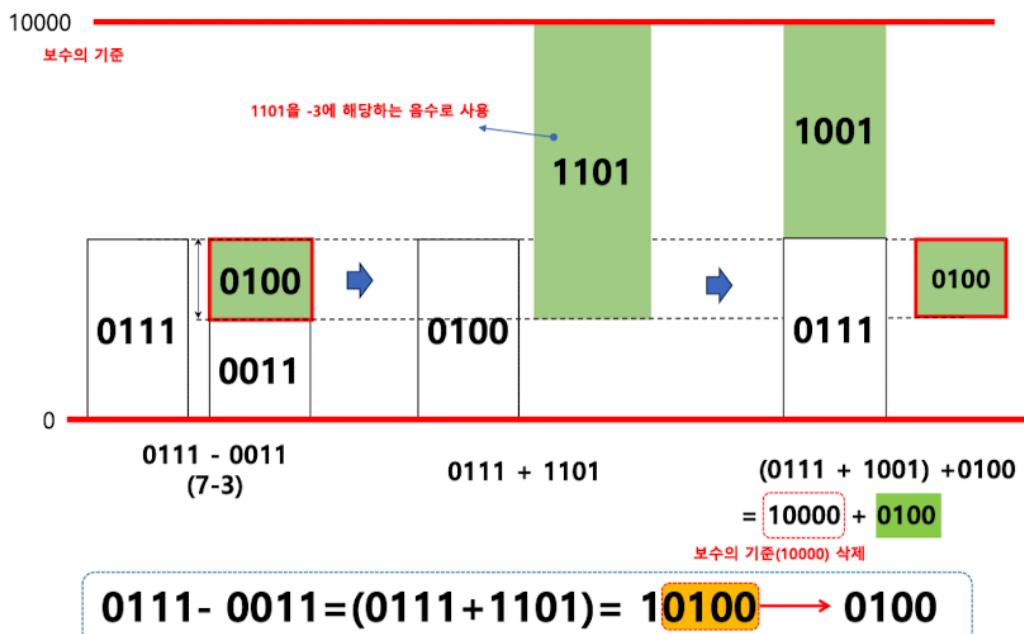
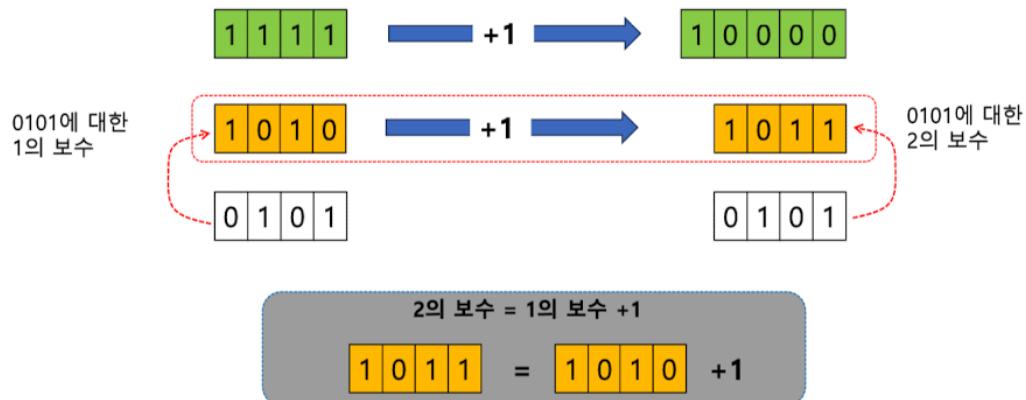
## 빼기의 결과를 더하기를 이용하여 얻기 위해 보수 사용

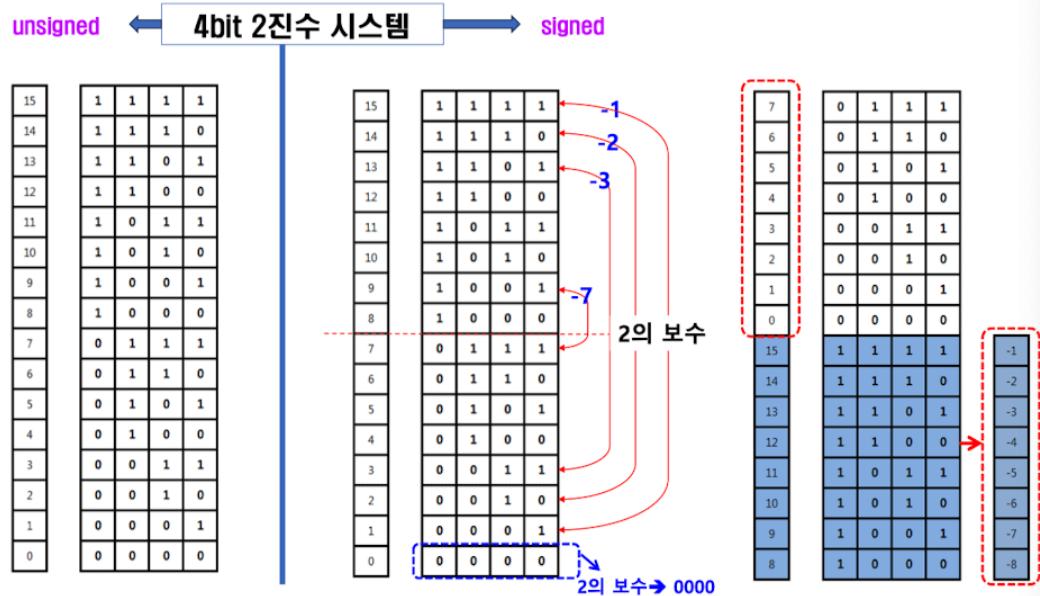


## [ 2 진수 ]

0101에 대한 1의 보수  
(0101에 대한 1111의 보수)  
→ 1111을 만들기 위해 0101에 보충해주어야 하는 수  
→ 1010

0101에 대한 2의 보수  
(0101에 대한 10000의 보수)  
→ 10000을 만들기 위해 0101에 보충해주어야 하는 수  
→ 1011





2의 보수 표현 (4bit)

7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
-1	1	1	1	1
-2	1	1	1	0
-3	1	1	0	1
-4	1	1	0	0
-5	1	0	1	1
-6	1	0	1	0
-7	1	0	0	1
-8	1	0	0	0

2의 보수

$2 - 4$   
 $= 2 + (-4)$   
 $= 0010 + 1100$   
 $= 1110$   
 $= -2$

- 2의 보수 : 2의 보수는 컴퓨터 과학에서 매우 중요한 개념으로, 특히 음수를 표현하는 데 사용된다. 2진법에서 주어진 숫자의 2의 보수를 구하는 방법은 다음과 같다 :

- 모든 비트를 반전시킨다. (0을 1로, 1을 0으로).
- 그 결과에 2를 더한다.

예를 들어, 4비트 숫자 5의 2의 보수를 구하면 :

- 5는 4비트로 0101
- 모든 비트를 반전하면 1010
- 1010에 1을 더하면 1011, 즉 -5가 된다.

2의 보수는 컴퓨터에서 덧셈과 뺄셈을 처리할 때 편리하게 사용할 수 있으며, 이를 통해 음수를 쉽게 표현할 수 있다.

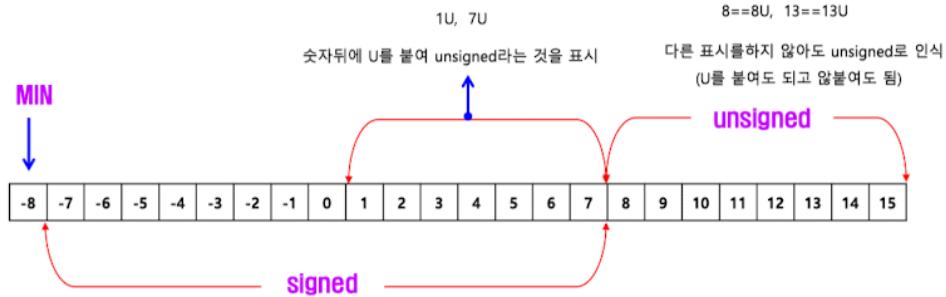
- 1의 보수 : 1의 보수는 모든 비트를 단순히 반전시킨 값이다. 2의 보수와 달리 추가로 1을 더하는 과정이 없으며, 1의 보수 체계에서는 음수를 나타낼 때 가장 큰 비트(부호 비트)를 사용한다.

예를 들어, 4비트 숫자 5의 1의 보수는 :

- 5는 4비트로 0101
- 모든 비트를 반전하면 1010, 이는 -5를 나타낸다.

## 4bit 2진수 시스템

unsigned에는 2의 보수를 반환하는 단항연산자 – 를 사용할 수 없다!!!



- Overflow

- 오버플로우(Overflow)란?

오버플로우는 컴퓨터에서 특정 자료형(예: 4, 8, 16, 32비트)이 표현할 수 있는 값의 범위를 초과할 때 발생하는 현상이다. 일반적으로 정수 연산에서 많이 발생하며, 이때 계산 결과는 원래 의도했던 값과 전혀 다른 값이 될 수 있다.

컴퓨터에서는 숫자를 지정할 수 있는 비트가 제한되어 있다. 예를 들어, 8비트 숫자는 -128에서 127까지의 정수를 표현할 수 있다. 만약 127에 1을 더하면 128이 되어야 하지만, 8비트로는 128을 표현할 수 없기 때문에 오버플로우가 발생하고 값이 -128로 변한다. 이는 값이 최대값을 넘어 다시 최소값으로 순환되는 효과이다.

- 오버플로우의 예시

- 양수 오버플로우 :

- 8비트 시스템에서 가장 큰 양수는 127이다. 여기서 1을 더하면 오버플로우가 발생하여 -128이 된다.
- 예 :  $127 + 1 = -128$  (오버플로우 발생)

- 음수 오버플로우 :

- 반대로, 가장 작은 음수는 -128이다. 여기서 1을 빼면 다시 양수로 변하는 오버플로우가 발생 한다.
- 예 :  $-128 - 1 = 127$  (오버플로우 발생)

- 오버플로우 확인 방법

오버플로우는 보통 덧셈이나 뺄셈 연산에서 발생하며, 두 가지 중요한 개념이 있다 :

- 올림수 (Carry) : 연산에서 비트가 넘쳐 다음 자리로 이동하는 값이다.
- 부호 비트 (Sign Bit) : 음수와 양수를 구분하는 비트. 가장 왼쪽 (최상위) 비트를 의미한다.

오버플로우를 감지하는 방법 중 하나는 덧셈이나 뺄셈에서 두 수의 부호와 결과의 부호를 비교하는 것이다. 예를 들어 :

- 두 양수를 더했을 때 결과가 음수이면 오버플로우가 발생한 것이다.
- 두 음수를 더했을 때 결과가 양수이면 오버플로우가 발생한 것이다.

마지막 자리 올림수와 마지막 자리에서 다음 자리로의 올림수 비교 :

이 부분은 덧셈 연산에서 오버플로우를 감지하는 방법 중 하나이다. 구체적으로 말하자면 :

1. 마지막 자리에서 발생하는 올림수 (Carry)는 하위 비트에서 상위 비트로의 올림이다.
2. 최상위 비트에서 발생하는 올림수 (Carry-out)는 마지막 자리(최상위 비트)에서 더 상위의 비트로 이동하는 올림이다.

두 올림수가 같으면, 계산된 값이 올바르게 표현된 것이므로 오버플로우가 발생하지 않는다. 반대로, 두 올림수가 다르면 오버플로우가 발생한다는 의미이다.

구체적인 예시로 설명 :

### No Overflow !!



위에 carry(올림수)가 110이라고 표시된 것은 덧셈 연산에서 각 자리에서 발생한 올림수를 계산한 결과입니다. 이진 덧셈에서는 각 자리의 비트 덧셈에서 올림(carry)이 발생할 수 있으며, 이 올림수가 다음 자리 덧셈에 영향을 줍니다.

이것을 확인하는 방법은 아래와 같은 아진 덧셈 규칙을 따르는 것입니다:

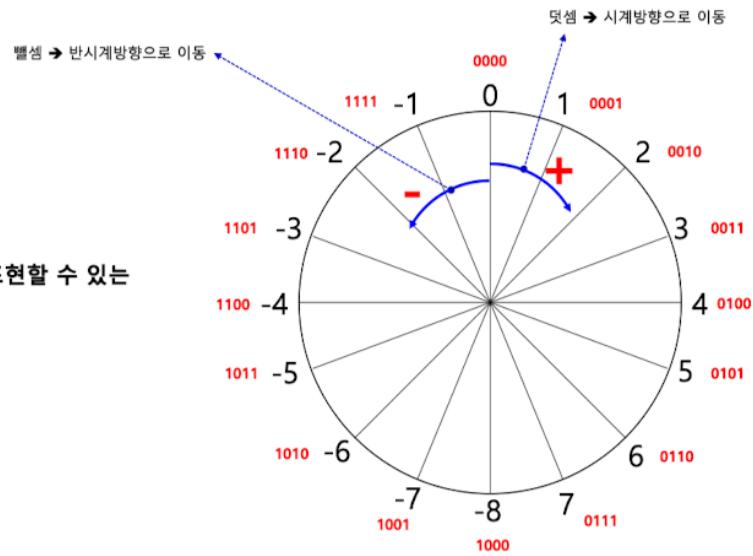
#### 덧셈 규칙

- $0 + 0 = 0$  (올림수 없음)
- $0 + 1 = 1$  (올림수 없음)
- $1 + 0 = 1$  (올림수 없음)
- $1 + 1 = 10$  (즉, 결과는 0이고 올림수가 1 발생)

## 4bit 2진수 시스템에서 덧셈과 뺄셈

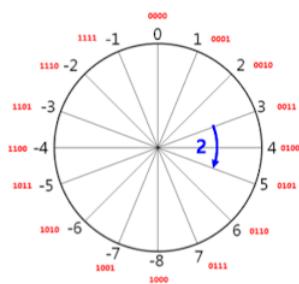
### Overflow

→ 계산 결과가 4비트로 표현할 수 있는  
숫자 범위를 벗어나는 것



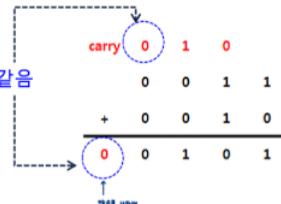
$$3+2 = 5$$

더한 결과 5가 4비트로 표현할 수 있는 숫자 범위(-8~+7)안에 있음  
→ No Overflow



Overflow 확인 방법  
→ 마지막 자리로 올라오는 올림수(carry)와 마지막 자리에서 다음 자리로 옮겨주는 올림수가 같으면 Overflow가 아니다!!!

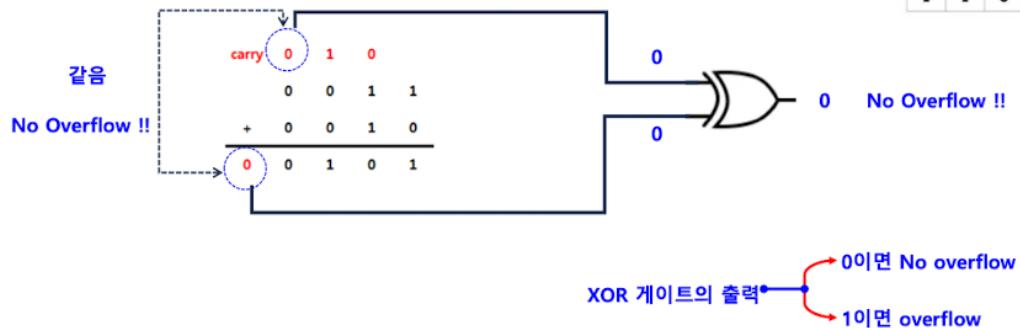
No Overflow !!



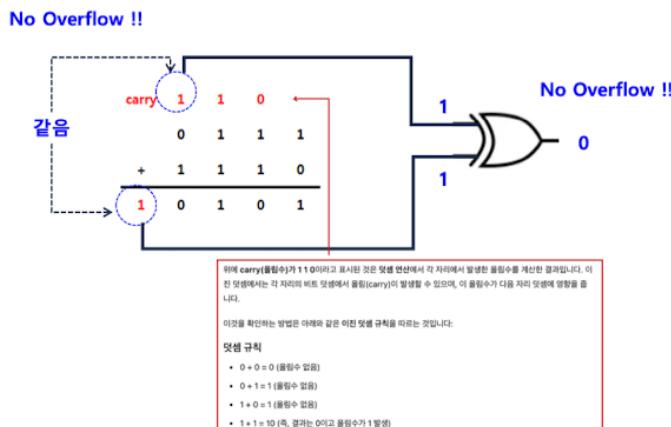
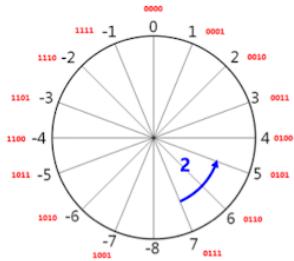
## Overflow Check with XOR Gate

Overflow 확인은 논리회로의 XOR 게이트를 이용해서 할 수 있다.

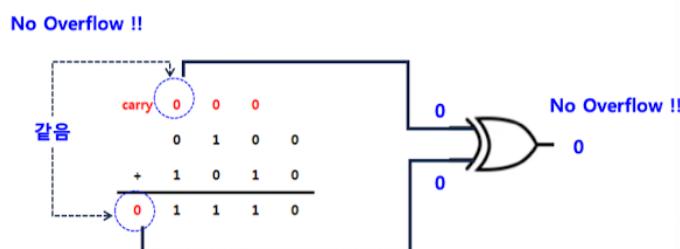
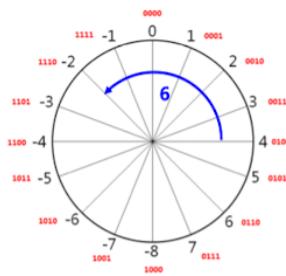
2 input XOR gate		
A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0



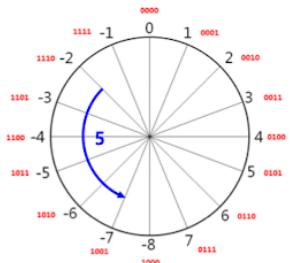
$$7-2=5$$



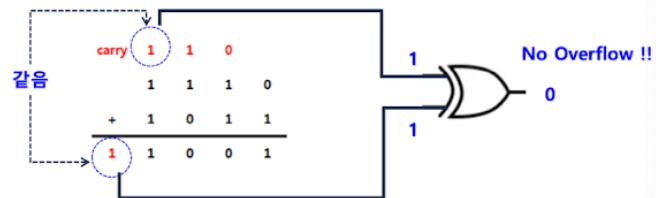
$$4-6=-2$$



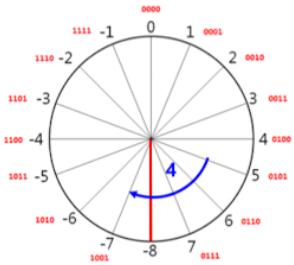
$$-2 - 5 = -7$$



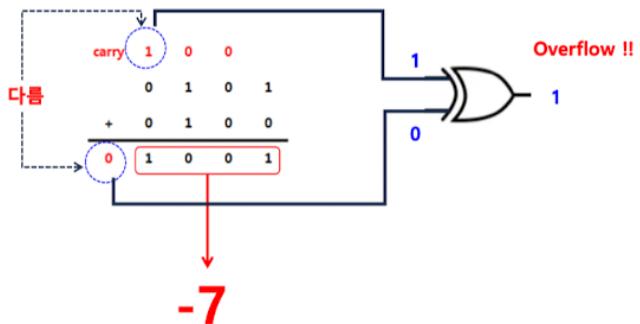
No Overflow !!



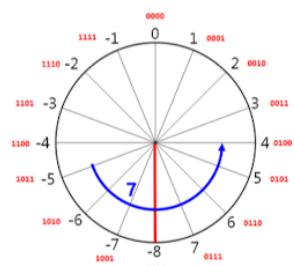
$$5 + 4 = -7$$



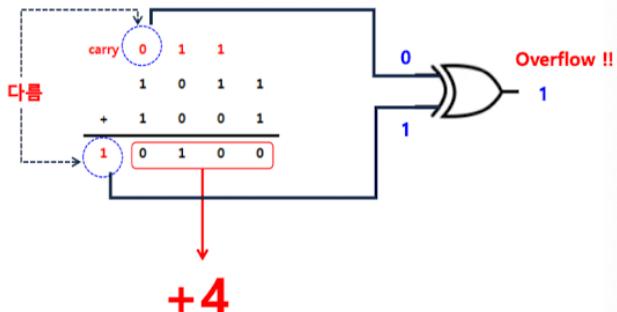
Overflow !!



$$-5 - 7 = +4$$



Overflow !!

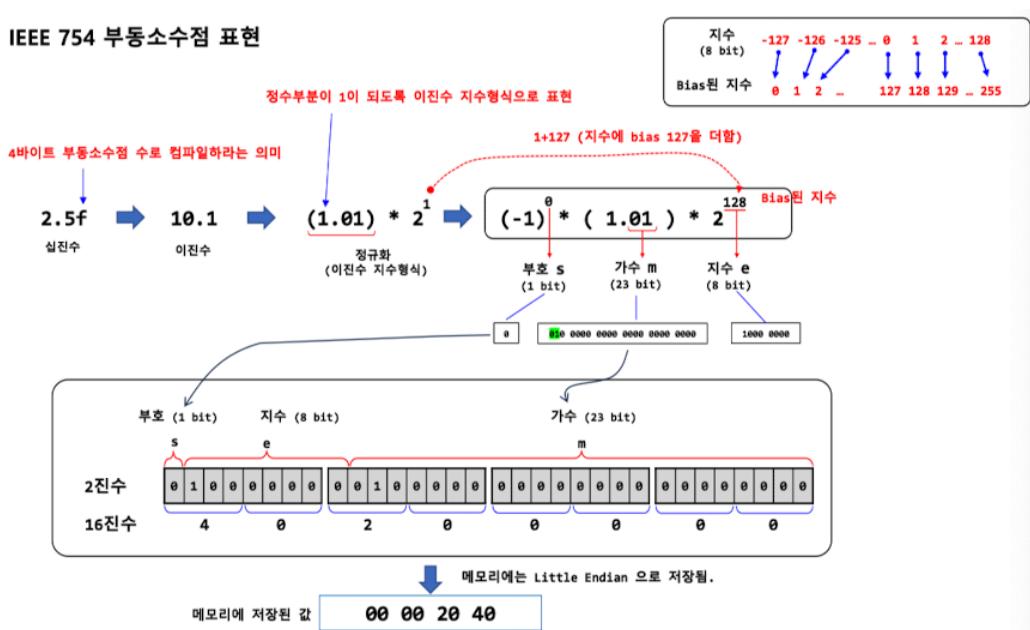


양수+양수=양수 (3+2=5)	큰 수-작은 수=양수 (7-2=7+(-2)=5)	작은 수 - 큰 수= 음수 (4-6=4+(-6)=-2)
$  \begin{array}{r}  \text{carry} \quad 0 \quad 1 \quad 0 \\  & 0 \quad 0 \quad 1 \quad 1 \\  + & 0 \quad 0 \quad 1 \quad 0 \\  \hline  0 \quad 0 \quad 1 \quad 0 \quad 1  \end{array}  $	$  \begin{array}{r}  \text{carry} \quad 1 \quad 1 \quad 0 \\  & 0 \quad 1 \quad 1 \quad 1 \\  + & 1 \quad 1 \quad 1 \quad 0 \\  \hline  1 \quad 0 \quad 1 \quad 0 \quad 1  \end{array}  $	$  \begin{array}{r}  \text{carry} \quad 0 \quad 0 \quad 0 \\  & 0 \quad 1 \quad 0 \quad 0 \\  + & 1 \quad 0 \quad 1 \quad 0 \\  \hline  0 \quad 1 \quad 1 \quad 1 \quad 0  \end{array}  $
음수 + 음수= 음수 (-2-5=(-2)+(-5)=-7)	큰 양수 + 큰 양수 = 음수 (5+4=-7)	큰 음수 + 큰 음수 = 양수 (-5-7=+4)
$  \begin{array}{r}  \text{carry} \quad 1 \quad 1 \quad 0 \\  & 1 \quad 1 \quad 1 \quad 0 \\  + & 1 \quad 0 \quad 1 \quad 1 \\  \hline  1 \quad 1 \quad 0 \quad 0 \quad 1  \end{array}  $	$  \begin{array}{r}  \text{carry} \quad 1 \quad 0 \quad 0 \\  & 0 \quad 1 \quad 0 \quad 1 \\  + & 0 \quad 1 \quad 0 \quad 0 \\  \hline  0 \quad 1 \quad 0 \quad 0 \quad 1  \end{array}  $	$  \begin{array}{r}  \text{carry} \quad 0 \quad 1 \quad 1 \\  & 1 \quad 0 \quad 1 \quad 1 \\  + & 1 \quad 0 \quad 0 \quad 1 \\  \hline  1 \quad 0 \quad 1 \quad 0 \quad 0  \end{array}  $

overflow

## 부동소수점 실수

- 실수 리터럴
  - 실수를 표현하는 방법
  - 소수점으로 표현된 숫자 (예 : 13.4 / 1. / .5)
  - 10의 거듭제곱으로 표기하는 과학적 표기법이 적용된 숫자  
(예 : 123E-1 / 1e+3) 알파벳 e 또는 E, 대소문자 구분없음
- IEEE 754 부동소수점 표현



<p>이 이미지는 IEEE 754 부동소수점 표준에 따라 2.5f라는 값을 어떻게 컴퓨터 메모리에 저장하는지 설명한 것입니다. IEEE 754는 부동소수점 수를 컴퓨터에서 효율적으로 표현하기 위한 국제 표준입니다. 여기에서 부동소수점 수는 세 부분으로 나뉩니다: 부호(S), 지수(E), *가수(M)**입니다.</p> <p>이미지에서 주요 내용을 단계별로 설명하겠습니다:</p> <ol style="list-style-type: none"> <li><b>1. 십진수 → 이진수 변환</b> <ul style="list-style-type: none"> <li>먼저, 부동소수점 수 2.5f는 십진수입니다.</li> <li>이것을 이진수로 변환하면 10.1이 됩니다. (2는 10이고, 0.5는 0.1로 변환)</li> </ul> </li> <li><b>2. 정규화</b> <ul style="list-style-type: none"> <li>부동소수점 수는 정규화된 형태로 표현됩니다. 즉, 소수점 왼쪽에 항상 1이 오도록 수식을 변경합니다.</li> <li>10.1은 정규화된 형태로 표현하면 1.01 * 2^1이 됩니다.</li> <li>여기서 1.01은 가수(mantissa)이고, 2^1은 지수(exponent)입니다.</li> <li>정규화는 계산 효율성을 높이고, 부동소수점 수의 구조를 표준화하기 위해 필요합니다.</li> </ul> </li> <li><b>3. 부호, 지수, 가수로 나누기</b> <ul style="list-style-type: none"> <li>부동소수점 수는 부호(S), 지수(E), 가수(M)로 나눌 수 있습니다.</li> <li>부호(S): 이 값은 양수이므로 0입니다. (음수는 1)</li> <li>지수(E): 지수는 1이지만, IEEE 754에서 지수는 Bias라는 개념을 사용하여 저장합니다. Bias는 127이 기본값이므로, 1 + 127 = 128로 저장됩니다.</li> <li>가수(M): 정규화된 가수의 1.01에서 1은 생략되고, 소수점 이하 부분만 저장됩니다. 즉, 01000000000000000000000000000000이 가수로 저장됩니다.</li> </ul> </li> <li><b>4. Bias 지수</b> <ul style="list-style-type: none"> <li>지수는 Bias(127)를 기준으로 하여 변환됩니다. 원래 지수는 1이었지만, Bias가 127이므로 1 + 127 = 128 이 최종적으로 저장됩니다.</li> <li>이미지 오른쪽 상단에 있는 설명은 이 과정을 나타냅니다.</li> </ul> </li> </ol>	<p><b>5. 메모리에 저장되는 값</b></p> <ul style="list-style-type: none"> <li>이 수는 32비트 부동소수점 수이므로, 4바이트(32비트)로 메모리에 저장됩니다.</li> <li>이 값은 다음과 같은 32비트 형식으로 저장됩니다:       <ul style="list-style-type: none"> <li>부호(S): 0</li> <li>지수(E): 10000000 (128)</li> <li>가수(M): 01000000000000000000000000000000</li> </ul> </li> <li>이 값을 리틀 엔디안 형식으로 메모리에 저장하면: 00 00 20 40이 됩니다.</li> <li>리틀 엔디안 방식에서는 메모리 상에서 가장 하위 바이트(00)가 먼저 저장됩니다. 최종적으로 메모리에 저장되는 값은 리틀 엔디안 형식으로 00 00 20 40입니다.</li> </ul> <p><b>최종 요약:</b></p> <p>이 이미지에서는 부동소수점 수 2.5f가 IEEE 754 표준에 따라 32비트(4바이트)로 변환되는 과정을 설명하고 있습니다. 부호(S), 지수(E), 가수(M)로 나뉘어 저장되며, Bias를 적용하여 지수를 변환하고, 가수는 정규화되어 저장됩니다. 최종적으로 메모리에 저장되는 값은 리틀 엔디안 형식으로 00 00 20 40입니다.</p>
--	--

## 1장. C언어와 컴파일러

### C언어 소개

#### 고급 언어와 저급 언어

- 기계어
- 어셈블리 언어
- C 언어

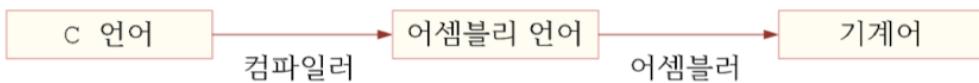
10011111 00000011

mov %ax, \$3

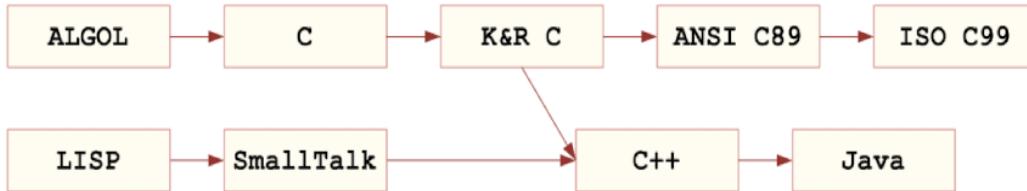
a = 3;

C 언어	어셈블리 언어	기계어
count = 1;	mov %ax, \$1	10011111 00000001
while (count <= 10)	loop: add \$2, %ax	00101101 00000010
count = count + 2;	cmp \$10, %ax	00110001 00001010
	jle loop	11010001 10110001

- 고급 언어와 저급 언어



## 고급 언어의 진화



- C by Dennis Ritchie and Kenneth Thompson, 1972
- K&R C: Kernighan and Ritchie C
- ANSI C89가 사실상의 표준 (de facto standard)
- 절차적 언어: 절차에 주안점
  - 이렇게 하라. 그 다음에는 저렇게 하라.
- 객체지향 언어: 객체 (Object, 대상) 와 객체의 임무에 주안점
  - 객체 A는 이러이러한 작업을 담당하라.
  - 객체 B는 이러이러한 작업을 담당하라.

## C 언어의 장점

- 어셈블리 언어
  - 고속성: 레지스터나 메모리를 직접 다룰 수 있는 구체적 수준의 언어
- C 언어
  - 추상적 수준의 언어: 어셈블리 언어로 번역하는 것은 컴파일러에게 일임. 추상적 수준에서 프로그램을 여러 모듈로 분할하는 구조적 프로그래밍이 가능.
  - 간결성: 실행 파일 크기가 작음.
  - 이식성: 하드웨어나 운영체제가 바뀌어도 호환성이 높음.
  - 고속성: 다른 고급 언어에 비해 실행 속도가 빨라 고급 어셈블리 언어라 부름. 운영체제, 컴파일러, 데이터베이스, 임베디드 프로세서 작성에 활용.
  - 오랫동안 쌓인 풍부한 함수 라이브러리

Hello, world.

## 샘플 프로그램 해설



## 주석(Comments)을 다는 두 가지 방법

```
/* This program prints "Hello, world." on the screen. */
```

- /\* (Slash Asterisk) 와 \*/ (Asterisk Slash) 사이

```
printf("Hello world. \n"); // prints hello message
```

- // (Double Slash) 가 시작하는 곳부터 그 줄이 끝날 때까지

## 지시어(Directives)

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

- **지시어는 #으로 시작**
  - 컴파일러가 번역해야 할 **명령어**가 아님. (세미콜론도 없음)
  - **전 처리기 (Pre-processor)**가 처리
  - “여기에 stdio.h라는 파일의 내용을 포함시켜 주세요.”
- stdio.h (standard input/output header)
  - 표준 입출력을 위한 헤더 파일
  - 표준 입력 장치 = 키보드
  - 표준 출력 장치 = 화면
  - printf나 scanf 등 입출력 함수가 선언되어 있음.

- 전처리기가 처리 후 라이브러리 함수는 object 코드 형태로 제공된다. (컴파일 진행 X, 컴파일 시간의 부담을 줄이기 위해서) → 이후 프로그래머가 작성한 소스코드만 따로 컴파일한 후에 링커를 통해 곧바로 오브젝트 라이브러리와 링크함으로써 실행 파일을 만든다.

## main 함수

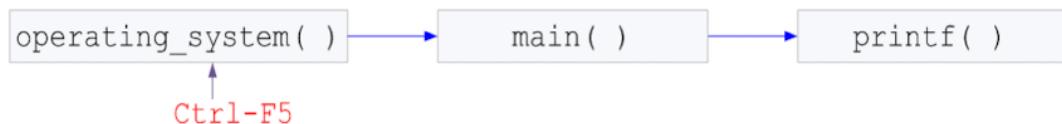
```
int main( ){
    printf("Hello, world.\n");
    return 0;
}
```

- `int`는 `integer`(정수)를 의미
  - 자신을 호출한 함수에게 돌려주는 리턴 값의 타입
  - 0을 리턴하고 있으므로 타입이 매칭
  - `return 1.0;` 이라고 했으면 타입 미스매치
- `main`은 함수명(진입점 함수)
- 중괄호로 둘러싸인 부분이 **함수 본체 (Function Body)** 또는 **함수 블록 (Function Block)**

## 함수의 역할

```
icebar get_icebar(money 1000) {
    take 500 as service charge;
    go to market;
    pay 500 and buy a melona;
    return melona;
}
```

- 함수는 심부름꾼
  - 건네받은 것이 있을 수도 있고 없을 수도 있음.
  - 돌려주는 것이 있을 수도 있고 없을 수도 있음.



- 연쇄적으로 함수를 호출해야 할 때도 있음.

## 컴파일 오류

### Example 1-1 실습 및 해설

```
//include <stdio.h>
int main() {
    printf("Hello, Wordl.\n");
    return 0;
}
```

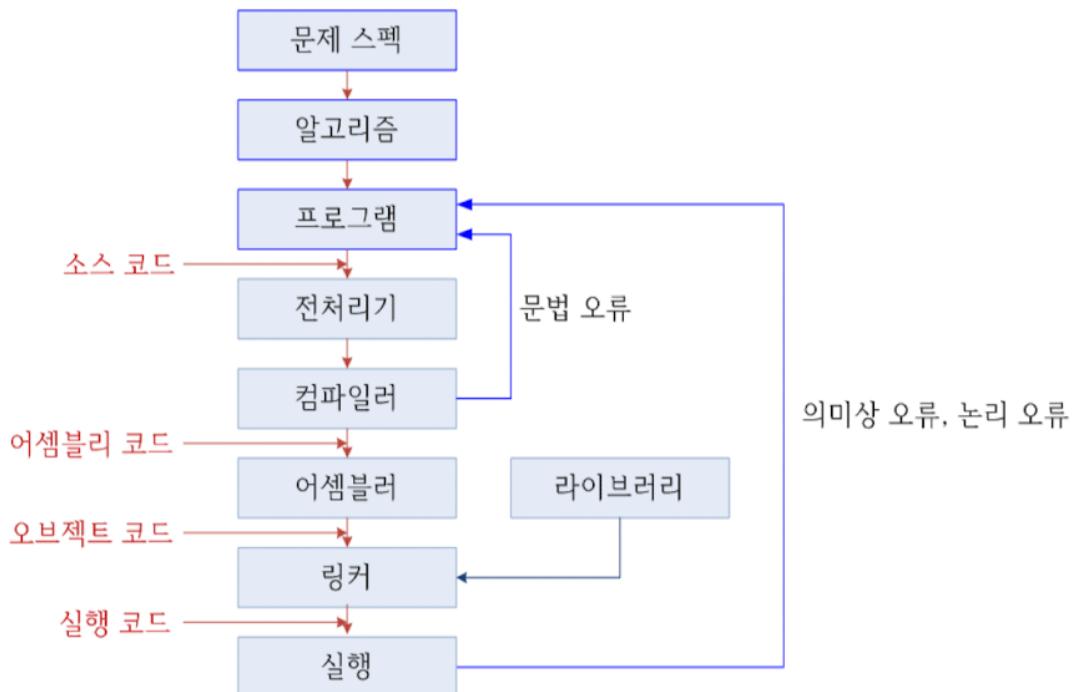
## 프로그램 작성 및 실행

## 알고리즘

1. 냄비에 적당량의 물을 담아 가스 불 위에 올려놓는다.
2. 물이 다 끓을 때 쯤, 라면 봉지를 뜯어 라면을 넣는다.
3. 대파를 길쭉하게 썰어서 준비한다.
4. 면발이 풀릴 때 쯤, 준비해 놓은 대파와 계란을 넣는다.
5. 라면 봉지에 있던 수프를 넣는다.

- 알고리즘 (Algorithms)
  - 문제 해결 방법을 단계적으로 기술한 것.
  - 의사코드 (擬似, Pseudo Code)

## 프로그램 작성 및 실행 절차

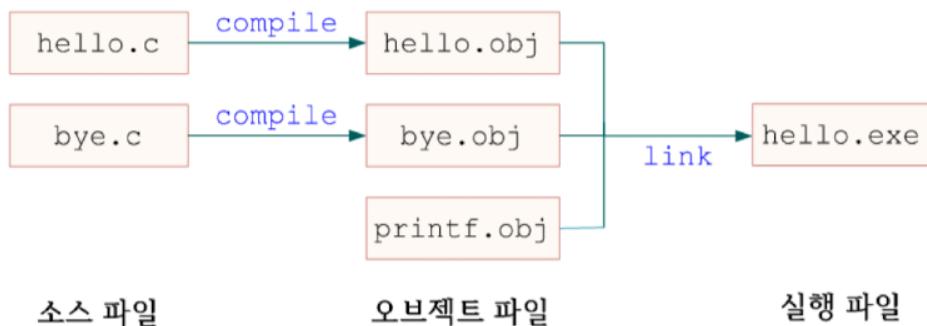


## 세 가지 오류

- 문법 오류, 의미상 오류, 논리 오류 (알고리즘 오류)
- 논리 오류의 예:
  - 소득이라는 단어가 들어간 문장은 몇 개인가?
  - 알고리즘: 소득이라는 단어 이후에 마침표이면 한 문장.
  - 그러나, 아래 입력에 대해 이 알고리즘은 오류

소득이 부진했던 지난 해 경제 성장률이 외환위기 이후 최저인 3.1%에 불과한데, 국민 소득이 10.1%가 늘어나게 된 이유는 국민 소득의 개념 및 계산 방법의 차이에 있다.

## 코드 형태 변화



- 소스 코드 → 오브젝트 코드 → 실행 코드
  - 오브젝트 파일 및 실행 파일은 2진 파일
  - 표준 헤더 파일에 선언된 함수는 **오브젝트 코드 형태**로 저장되어 있음. (WHY?)
- C언어가 제공하는 라이브러리 함수는 이미 컴파일이 끝난 오브젝트 코드 형태로 제공된다. 이른바 오브젝트 라이브러리다. 라이브러리를 소스코드 형태로 제공할 경우 컴파일 시간이 부담이 된다. 내 소스코드에 오류가 발생해서 그것을 수정하여 다시 컴파일을 할 때마다 소스코드가 호출한 라이브러리 함수까지도 같이 컴파일을 해야 하기 때문이다. 아무런 오류도 없는 라이브러리 함수까지 같이 컴파일을 할 필요는 없다. 따라서 그림에서 보듯 프로그래머가 작성한 소스코드만 따로 컴파일한 후에 곧바로 오브젝트 라이브러리와 링크함으로써 실행 파일을 만드는 데 걸리는 시간을 줄일 수 있다. 실제로 #include <stdio.h>도 소스 코드 형태가 아니라 오브젝트 코드 형태의 stdio 라이브러리를 포함하라는 지시어다.

## 링커의 역할

함수 명	주소	오브젝트 코드
printf( )	219	11010001
	...	...
	200	00110001
main( )	199	00101101
	...	
	164	01000001
	160	00101001 = printf( )
	100	10011111

- 연결 정보를 오브젝트 코드에 삽입하여 실행 파일을 생성.
- 링커는 함수 사이를 연결하기 위한 소프트웨어이다. 그림에서 보듯 main 함수가 메모리 100~199번지를 차지하고 라이브러리 함수인 printf 함수가 200~219번지를 차지한다고 가정해 보자. 그 경우 만약 메모리 160번지의 명령어가 printf 함수를 호출하는 명령어라면 프로그램이 실행해야 할 다음 명령어는 164번지 명령어가 아니라 printf 함수가 시작하는 200번지에 있는 명령어다. 그러자면 프로그램 실행 도중에 현재

위치인 160번지로부터 40만큼 더한 위치에서 printf 함수가 시작한다는 정보를 미리 알고 있어야 그리로 건너뛸 수 있다. 시스템 소프트웨어의 일종인 링커는 그러한 연결 정보를 오브젝트 코드에 삽입함으로써 실행 파일을 만드는 역할을 한다. Visual C는 컴파일, 어셈블, 링크 작업을 모두 묶어서 빌드라는 말로 부른다.

▼ 09/19 목 - 2장

## 2장. 변수와 산술 연산

### 2-1. 숫자 표현

#### Binary Number System

Decimal: 0, 1, ..., 9, **10**, 11, ...

Binary: 0, 1, **10**, 11, 100, 101, 110, ...

1 비트

0
1

8 비트

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1

2 비트

0	0
0	1
1	0
1	1

...

1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1

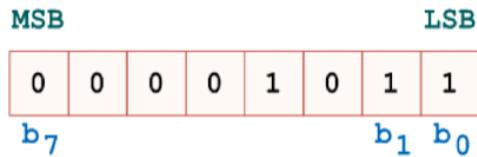
- 0과 1로 구성된 수 체계가 2진법
- N 비트로 표현할 수 있는 경우의 수는?

## 진법 변환

Binary → Decimal:  $1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$   
Decimal → Binary:

$$\begin{array}{r} 2 ) \underline{11} \\ 2 ) \underline{5} \quad \text{---} \quad 1 \\ 2 ) \underline{2} \quad \text{---} \quad 1 \\ \text{---} \quad \text{---} \quad 0 \end{array}$$

- 2진수에서 10진수로, 10진수에서 2진수로



- 1 바이트 표현이면 상위 4비트를 **제로 패딩 (Zero Padding)**
- Most Significant Bit, Least Significant Bit

- 2진수를 10진수로 변환하려면 위의 예처럼 자릿수별로 상응하는 2의승수를 곱해서 더하면 된다.
- 역으로 10진수를 2진수로 변환하려면 그림에서 보듯이 연속해서 2로 나누되 몫을 아래쪽에, 나머지는 오른쪽에 쓴 후에 최종 몫과 나머지를 화살표 순으로 쓰면된다. 이 결과를 8비트로 표현하면 그림처럼 0000 1011이 된다. 상위 4비트를 0으로 채워야하기 때문이다. 이를 제로 패딩이라 부른다.
- 비트에 명칭을 부여하기도 한다. 최하위 비트  $b_0$ 을 LSB(Least Significant Bit)라고 부른다. 2의 0승 자리로서 값이 가장 작기 때문이다.
- 최상위 비트를 MSB(Most Significant Bit)라고 부른다.

## 2진수, 16진수

①  $10110100 = 1011\ 0100 = \text{B}4$

②  $1010101111 = 0010\ 1010\ 1111 = \text{2AF}$

③ 1011 0100 (① 증명)

$$\begin{aligned} &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= (1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \text{2}^4 + (0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0) \text{2}^0 \\ &= (1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \text{16}^1 + (0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0) \text{16}^0 \\ &= \text{B}4 \end{aligned}$$

④  $\text{2AF} = 0010\ 1010\ 1111$

1	0	1	0	1	0	1	1	1	1
0	0	1	0	1	0	1	0	1	1

$\xleftarrow[2]{A}\xrightarrow[F]{}$

- 2진수와 16진수 사이에는 **직접 변환**이 가능
- 4자리 단위로 끊음. 필요시 ②, ④처럼 제로 패딩

## 2의 보수(2's Complement)

+ 5: 00000101

- 5: 10000101

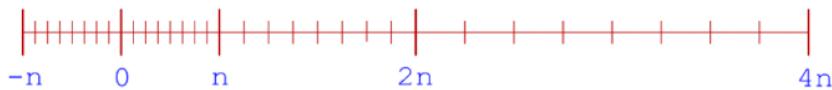
- 부호, 절대값 표현: +0과 -0.

+5	0 0 0 0 0 1 0 1	
	1 1 1 1 1 0 1 0	1의 보수
	1	+ 1
-5	1 1 1 1 1 0 1 1	2의 보수

- 2의 보수 표현: 0 표현이 Unique.

## 정수 저장, 부동소수 저장

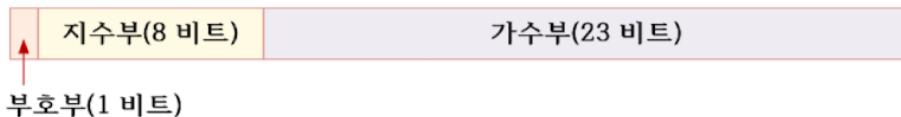
- 저장 가능한 정수의 범위
  - 8 비트이면 256가지
  - 0부터 255까지 또는 -128부터 127까지
  - $n$  비트이면  $-2^{n-1} \leq N \leq 2^{n-1}-1$
  - 그 외는 오버플로우 오류



- 실수는 연속. 부동소수 표현은 띄엄띄엄.

$$\begin{aligned}0.375 \times 2 &= 0.750 = 0 + 0.750 && \text{Hence } b_{-1} = 0. \\0.750 \times 2 &= 1.500 = 1 + 0.500 && \text{Hence } b_{-2} = 1. \\0.500 \times 2 &= 1.000 = 1 + 0.000 && \text{Hence } b_{-3} = 1.\end{aligned}$$

## 부동소수 표현

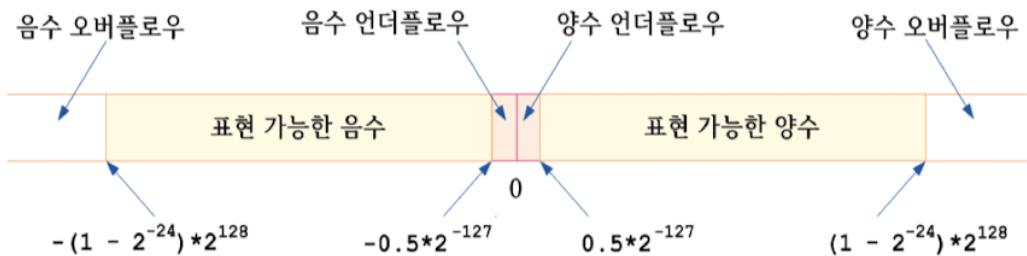


- IEEE 754 표준 32 비트 부동소수 저장 형식
- 단일 정밀도 (cf. 64비트 2중 정밀도)

$$\begin{aligned}45.625 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\&= 101101.101 \\&= 1.01101101 \times 2^5 \\0.0001101001110 &= 1.101001110 \times 2^{-4}\end{aligned}$$

- 45.625
  - 부호부 0 (= +) . 지수부 00000101 (=5) . 가수부 01101101

## Overflow and Underflow



- 언더플로우일 경우 Divide by Zero 오류에 주의

### Example 2-1 실습 및 해설

```
#include <stdio.h>

int main() {
    int i;
    float sum = 0.0F;

    for (i = 0; i < 100;x i++)
        sum = sum + 0.1F;

    printf("The sum is %f\n");
    return 0;
}
```

부동 소수 연산에는 항상 오차가 개입될 수 있다는 점에 유의해야 한다. 예는 0.1을 100번 더한 값을 출력하는 프로그램이다. 프로그램에 대한 구체적인 해설은 이후로 미루더라도 이 프로그램을 실행한 결과는 10.000000이 아니라 10.000002다. 부동 소수 표현에 오차가 있기 때문이다. 0.1을 2진수로 표현해 보면 그 이유를 알 수 있다.

## 2-2. 변수와 상수

## 변수는 그릇과 같다

밥그릇 종류 a;

int math;

↑  
타입 명      ↑  
                변수 명

- 변수 선언: 타입명과 변수명
- Type = Data Type = 자료형

a = 쌀밥;

- 변수에 값을 담는 명령문 = 대입문

### Example 2-2 실습 및 해설

```
#include <stdio.h>

int main(){
    int math;
    int science;
    int sum;

    math = 90;
    science = 100;
    sum = math+science;

    printf("sum of math and science is %d \n",sum);

    return 0;
}
```

## C 언어의 자료형

데이터 타입		일반 32 비트	인텔 IA-32	x86-64
문자	char	1	1	1
정수	short	2	2	2
	int	4	4	4
	long	4	4	8
	long long	8	8	8
부동소수	float	4	4	4
	double	8	8	8
	long double	16	8	8

- double은 2중 정밀도
- short < int < long < long long
- float < double < long double

## signed vs. unsigned

### Example 2-3 실습 및 해설

```
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main () {
    printf("sizeof(char) is %d.\n", sizeof(char));
    printf("sizeof(short) is %d.\n", sizeof(short));
    printf("sizeof(int) is %d.\n", sizeof(int));
    printf("sizeof(long) is %d.\n", sizeof(long));
    printf("sizeof(long long) is %d.\n", sizeof(long long));
    printf("sizeof(float) is %d.\n", sizeof(float));
    printf("sizeof(double) is %d.\n", sizeof(double));
    printf("sizeof(long double) is %d.\n", sizeof(long double));
    printf("int ranges from %d to %d.\n", INT_MIN, INT_MAX);
    printf("double ranges from %e to %e.\n", DBL_MIN, DBL_MAX);
    return 0;
}
```

```

int age;
unsigned int age;
• int = signed int
• unsigned int를 써서 더 큰 정수를 수용

```

컴퓨터에서 디폴트(Default)라는 말의 의미는 아무런 말을 하지 않으면 자동으로 먹히는 값을 의미한다. 예를 들어 우리가 '식사하셨어요?'라고 물으면 그것은 디폴트로 밥을 먹었느냐는 의미다. 물론 빵이 주식인 외국인이 이렇게 물어 보았다면 빵을 먹었느냐고 물어 보았을 가능성이 크다.

int age; 처럼 타입명 앞에 아무것도 붙이지 않으면 디폴트로 signed로 간주된다. 즉, int age;는 signed int age;와 동일한 선언으로서 age 변수에 양수나 음수를 저장하겠다는 의미다. 그러나 unsigned int age;처럼 unsigned를 붙이면 음수는 저장하지 않겠다는 의미다. 나이를 저장할 age 변수에 음수를 저장할 일은 거의 없기 때문이다. unsigned라는 말은 부호가 없다는 의미로서, 0 이상의 정수만 담겠다는 뜻이다. 음수를 저장하지 않으므로 MSB까지도 절댓값을 저장하는데 사용하면 담을 수 있는 최대 양수는 signed일 때 비해 거의 두 배로 늘어난다. 4바이트 signed int로는  $-2^{31}$ 부터  $2^{31}-1$ 까지의 정수를 담을 수 있지만 unsigned int로는 0부터  $2^{32}-1$ 까지의 정수를 담을 수 있다.

```

unsigned int age;
long sum; unsigned long total;
age = 24U;           // 우변 디폴트는 signed int
sum = 24L;
total = 2024UL;

```

상수에 접미사(Suffix)를 붙여야 할 때도 있다. age = 24U의 좌변 age 변수를 unsigned로 선언했으면 우변 상수에 U라는 접미사를 붙여 24를 unsigned 형으로 바꾸라고 요청해야 한다. 그래야 좌변과 우변의 자료형이 매칭되기 때문이다. 우변 상수에 아무런 접미사가 없으면 디폴트로 signed가 먹는다.

sum = 24L의 우변도 그냥 24라고 쓰면 디폴트로 int 형으로 간주된다. 따라서 24에 L을 붙여 long 형으로 바꾸라고 요청해야 한다. unsigned long이라면 total = 2024UL;처럼 UL을 붙여야 한다.

long long이라면 LL을, unsigned long long이라면 ULL을 붙여야 한다.

## 형 변환

### ● Example 2-4 실습 및 해설

### ● Example 2-5 실습 및 해설

- `double total;`
  - `total = 0.1234567F;` (**자동 형 변환**)
  - `total = (double)0.1234567;` (**강제 형 변환**)
- 정수는 디폴트로 `int`, 실수는 디폴트로 `double`로 먹힌다.
- `int`보다 작은 타입(`short, char`)
  - `int` 타입으로 변환된 후에 연산이 가해진다.

ex 2-4 :

```
#include <stdio.h>

int main()
{
    float area;
    double total;

    area = 0.1234567; // 1
    printf("Area is %f.\n", area);

    total = 0.1234567; // 2
    printf("Total is %lf.\n", total);

    total = 0.1234567f; // 3
    printf("Total is %lf.\n", total);

    return 0;
}
```

```
(base) minsung@minseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev $ ./example_2-4
Area is 0.123457.
Total is 0.123457.
Total is 0.123457.
```

컴파일러에 따라서는 //1에 대해 경고 메세지를 띄우기도 한다. 우변 실수는 디폴트로 `double`로 인식되지만 좌변 `area`가 `float` 형으로 선언되어 타입이 다르기 때문이다. `float`가 4바이트라면 `double`은 8바이트다. 따라서 이는 4바이트 크기의 그릇에 8바이트 크기의 값을 대입하라는 요구와도 같다.

//2는 우변과 좌변이 모두 double이기 때문에 오류가 아니다. printf 함수는 %f와 %lf 형식 모두가 디폴트로 소수 이하 6자리까지 찍는다.

C언어는 타입에 민감하지만 어느 정도 융통성을 발휘하기도 한다. //3의 우변 0.1234567f는 f라는 접미사에 의해 float로 인식된다. 그런데 이 float 값을 좌변 double 타입 변수에 대입할 때에는 타입이 다름에도 불구하고 아무런 경고를 하지 않는다. 작은 것을 그보다 큰 그릇에 넣으라고 했기 때문이다. 이 경우 좌변 타입에 맞추어 자동으로 우변에 형 변환이 가해진다. 즉, 우변 값을 float 형에서 double 형으로 변환한 후에 total에 대입한다.

ex 2-5 :

```
#include <stdio.h>

int main()
{
    short width;
    int height;
    float area;

    double total;
    int length;
    short num1, num2, sum;

    width = (short)10; // short width // 명시적 형변환 -> casting operator
// 1
    height = 5; // int height
    area = 100.0f; // float area

    // 암묵적 형변환, 자동 형변환
    total = area + width * height; // double total // 2
    printf("Total is %lf.\n", total);

    length = 3.14; // int length // 3
    printf("length is %d.\n", length);

    length = (int)3.14; // int length // 4
    printf("length is %d.\n", length);

    num1 = (short)10;
    num2 = (short)20;
    printf("sizeof(num1 + num2) is %d\n", sizeof(num1 + num2)); // 5
    printf("sizeof('a' - 'A') is %d\n", sizeof('a' - 'A'))); // 6

    sum = num1 + num2; // short sum // 7
    printf("sum is %d\n", sum);
    printf("sizeof(sum) is %d\n", sizeof(sum));

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev $ ./example_2-5
Total is 150.000000.
length is 3.
length is 3.
sizeof(num1 + num2) is 4
sizeof('a' - 'A') is 4
sum is 30
sizeof(sum) is 2
```

자동 형 변환의 예다. //2에서 \* 기호는 곱셈을 나타낸다. 여기서 자동 형 변환은 3번 일어난다.

첫째, width \* height를 계산하는 도중 일어난다. 일반적으로 자동 형 변환이 일어날 때에는 크기가 작은 자료형이 크기가 큰 자료형으로 바뀐다. 따라서 short 형인 width가 height와 같이 int 형으로 변환된 후에 곱해진다.

둘째, 곱셈 결과와 area를 더하는 도중 일어난다. 곱셈의 결과가 int 형인데 비해 area는 float 형이다. 따라서 int 형이 float 형으로 변환된 후에 area에 더해진다.

셋째, 우변을 좌변에 대입하는 과정에서 일어난다. 좌변이 double 형이므로 우변의 float를 double로 변환한 후에 대입한다.

그러나 무턱대고 자동 형 변환에 의존해서는 안된다. //3의 좌변 length는 int 형이고 우변 3.14는 double 형이다. 컴파일러에 따라서는 이 경우 아무런 경고도 없이 우변 3.14에서 소수부를 잘라 버리고 정수부인 3만 취하여 좌변에 대입한다. 물론 그것이 프로그래머가 의도하던 바라면 문제가 되지 않는다. 그러나 만약 타입 매칭에 신경을 쓰지 않는 바람에 실수로 그렇게 되었다면 잘못이다.

```
(int)3.14, (double)3, (long int)3, (unsigned)3
```

위와 같이 의도적으로 형 변환을 가하기도 한다. //4의 우변 3.14는 디폴트로 double로 인식된다. 그것을 int로 바꾸는 형 변환 연산자가 (int)다. 이를 강제 형 변환이라고도 부른다. 강제로 형을 변환하려면 예처럼 괄호 안에 원하는 자료형을 쓰면 된다. 강제 형 변환은 컴파일러가 제공하는 자동 형 변환에 의존하지 않고 프로그래머 자신이 원하는 대로 자료형을 변환하기 위한 것이다.

//1도 short 형으로 변환하는 연산자다. int를 short나 char로 변환하면 상위 바이트가 날아간다. 예를 들어, 4바이트 int 변수 F1F20102를 2바이트 short로 변환하면 상위 2바이트인 F1F2가 날아가고 하위 2바이트인 0102만 남는다. 이 경우 부호가 바뀔 수 있음에도 유의해야 한다. F1F20102는 MSB가 2진수 0인 양수이기 때문이다.

산술 연산 시에 int보다 작은 자료형은 모두 일단 int형으로 변환된 후에 연산이 가해진다. 따라서 //5에서 num1, num2는 short 형의 2바이트이지만 연산 도중 num1, num2는 물론 연산 결과인 num1 + num2도 4바이트 int형으로 저장된다. 이것이 //7의 좌변 sum에 대입될 때 다시 형 변환이 일어나 short로 바뀐다. short 형이 연산 도중에는 int 형으로 바뀌듯이 char 형도 연산 도중에는 int 형으로 바뀐다. 또, 연산의 결과인 //6의 'a'-'A'도 int 크기의 메모리에 저장된다.

//5에서 유의할 점은 short 형으로 선언한 num1, num2 변수 자체가 int 형으로 바뀌는 것은 아니라는 점이다. 연산을 위해서 num1, num2를 임시 저장 장소로 복사해야 하는데 그 장소의 크기가 int 크기라는 것이다. 연산이 int 형으로 진행된다면 아예 num1, num2를 int로 선언하는 것이 낫지 않느냐고 할지 모르지만 대개 short는 2바이트로 저장하기 때문에 그리 크지 않은 정수 변수는 short로 선언함으로써 메모리를 절약할 수 있다.

## 타입 별 상수 표현

- 정수형

- `short`    (`short`) 97,    (`short`) `0x61`
- `int`       97,    `0x61`
- `long`      97L,    `0x61L`,    (`long`) 97

- 부동 소수형

- `float`    97.0F,    9.70E1F
- `double`    97.0,    9.70E1
- `long double`    97.0L,    9.70E1L

- 문자형

- `char` 'a' = 97

위는 상수 97을 여러 자료형으로 표현한 모습이다.

## ASCII Codes

0 NULL (Null Character)	32 SPACE	64 @	96 `
1 SOH (start of heading)	33 !	65 A	97 a
2 STX (start of text)	34 "	66 B	98 b
3 ETX (end of text)	35 #	67 C	99 c
4 EOT (end of transmission)	36 \$	68 D	100 d
5 ENQ (enquiry)	37 %	69 E	101 e
6 ACK (acknowledge)	38 &	70 F	102 f
7 BEL (bell)	39 '	71 G	103 g
8 BS (backspace)	40 (	72 H	104 h
9 TAB (horizontal tab)	41 )	73 I	105 i
10 LF (line feed)	42 *	74 J	106 j
11 VT (vertical tab)	43 +	75 K	107 k
12 FF (form feed, new page)	44 ,	76 L	108 l
13 CR (carriage return)	45 -	77 M	109 m
14 SO (shift out)	46 .	78 N	110 n
15 SI (shift in)	47 /	79 O	111 o
16 DLE (data link escape)	48 0	80 P	112 p
17 DC1 (device control 1)	49 1	81 Q	113 q
18 DC2 (device control 2)	50 2	82 R	114 r
19 DC3 (device control 3)	51 3	83 S	115 s
20 DC4 (device control 4)	52 4	84 T	116 t
21 NAK (negative acknowledge)	53 5	85 U	117 u
22 SYN (synchronous idle)	54 6	86 V	118 v
23 ETB (end of trans. block)	55 7	87 W	119 w
24 CAN (cancel)	56 8	88 X	120 x
25 EM (end of medium)	57 9	89 Y	121 y
26 SUB (substitute)	58 :	90 Z	122 z
27 ESC (escape)	59 ;	91 [	123 {
28 FS (file separator)	60 <	92 \	124
29 GS (group separator)	61 =	93 ]	125 }
30 RS (record separator)	62 >	94 ^	126 ~
31 US (unit separator)	63 ?	95 _	127 DEL

## 특수 문자

특수 문자	아스키코드	의미
\0	NULL	널 문자
\r	CR	캐리지 리턴
\n	LF	라인 피드
\t	TAB	수평 탭
\v	VT	수직 탭
\a	BEL	알람벨 소리
\b	BS	백스페이스
\'	'	작은 따옴표
\"	"	큰 따옴표
\\	\	백 슬래쉬

```
printf("hello,\tworld.\a\a\a\n");
printf("\ \"%%\n");
```

아스키코드 0부터 31까지 문자를 특수 문자라 부른다. 그림은 프로그램에서 자주 사용되는 특수 문자를 보여준다.

## 문자와 문자열

### Example 2-6 실습 및 해설

- 문자 변수는 1 바이트 정수, 문자 상수는 4 바이트 정수

### Example 2-7 실습 및 해설

- char에는 0-127, unsigned char에는 0-255 저장 가능
- 문자 (Character)
  - 작은 따옴표로 둘러싼다.
- 문자열 (Character String)
  - 큰 따옴표로 둘러싼다.
  - 끝에 '\0' (Null 문자)를 붙인다.
  - 'a'와 "a"는 다르다.

ex 2-6 :

```
#include <stdio.h>

int main()
```

```

{
    char ch;

    ch = 'a'; // sizeof('a') ==> 4 // 1

    printf("sizeof(ch) is %d\n", sizeof(ch));
    printf("sizeof(a) is %d\n", sizeof('a'));

    printf("\'a\' in character format is %c.\n", ch);
    printf("\'a\' in decimal format is %d.\n", ch);

    ch = 'a' + 1;
    printf("\'a\' + 1 in character format is %c\n", ch);

    ch = 97;
    printf("97 in character format is %c\n", ch);

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev $ ./example_2-6
sizeof(ch) is 1
sizeof(a) is 4
'a' in character format is a.
'a' in decimal format is 97.
'a' + 1 in character format is b
97 in character format is a

```

ex 2-7 :

```

#include <stdio.h>

int main () {
    int num;
    char ch;

    num = 128;
    ch = num;
    printf("%d %d\n", num, ch);

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev $ ./example_2-7
128 -128

```

## 매크로 상수와 문자 상수

```
#define ADULT_AGE 19
          ↑           ↑
          기호상수      문자상수
          (Macro)       (Literal)
```

```
#define ADULT_AGE 19
#define LONG_MSG "This is a very long literal constant \
long enough to occupy two lines."
```

- 문자 상수를 매크로 상수로 치환할 수 있다.
- '\'는 문자 상수가 이어짐을 표시

### Example 2-8 실습 및 해설

리터럴 상수가 길어져 한 줄을 넘어갈 경우 Enter 키를 눌러 다음 줄로 이동하면 안된다. 예처럼 반드시 백 슬래시를 써서 문자열이 계속된다는 것을 표시해야 한다. 그렇지 않을 경우 Enter 키까지도 문자 상수에 포함되기 때문이다.

ex 2-8 :

```
#include <stdio.h>
#define PI 3.14
#define ERR_MSG "Error has occurred while opening file.\n"
#define MAX 32.0
#define MIN 10.0

int main () {
    double area, circum, diff;

    area = PI * 3.0 * 3.0;
    printf("The area is %lf.\n", area);
    circum = 2.0 * PI * 3.0;
    printf("The circumference is %lf.\n", circum);

    diff = MAX - MIN;
    printf("The difference is %lf.\n", diff);
    printf(ERR_MSG);

    return 0;
}
```

```
(base) minsung@minseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev $ ./example_2-8
The area is 28.260000.
The circumference is 18.840000.
The difference is 22.000000.
Error has occurred while opening file
```

## 변수 이름

- 의미를 부여

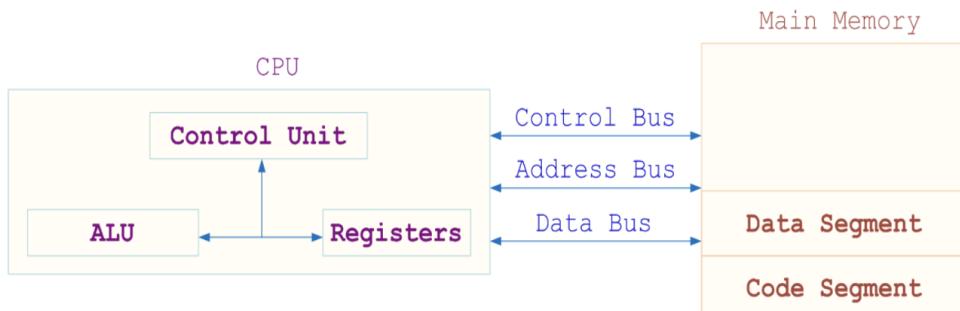
- 알파벳, 숫자, 밑줄 문자. 대소문자에 민감.
- 가독성이 중요: `int a;` 보다 `int age;`
- 밑줄 문자로 시작하면 시스템 라이브러리와 충돌 우려
- 조합 가능: `salary_sum`, `SalarySum`

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

- 예약어는 사용 불가

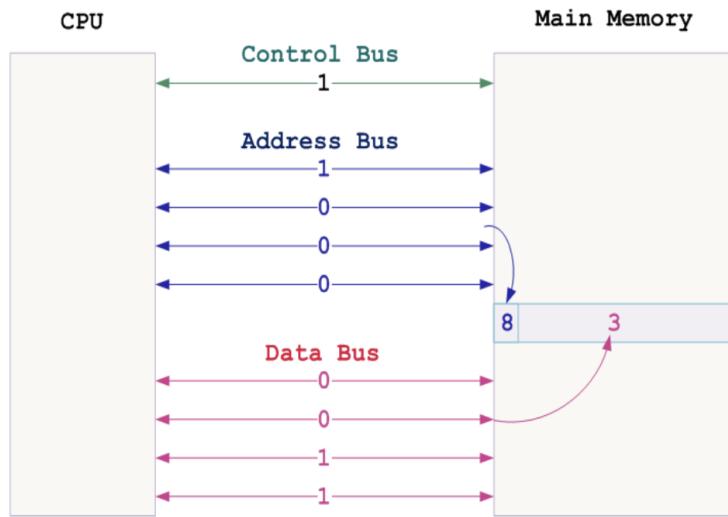
## 2-3. 변수와 메모리

### 컴퓨터 하드웨어



- 폰 노이만 구조
  - 데이터는 물론 프로그램까지도 메모리에 저장
  - 데이터 세그먼트와 코드 세그먼트
- CPU
  - CU (Control Unit): 명령어 해독 및 실행
  - ALU (Arithmetic Logic Unit): 산술 및 논리 연산
  - Registers: 연산 결과 및 명령어 저장

## 버스

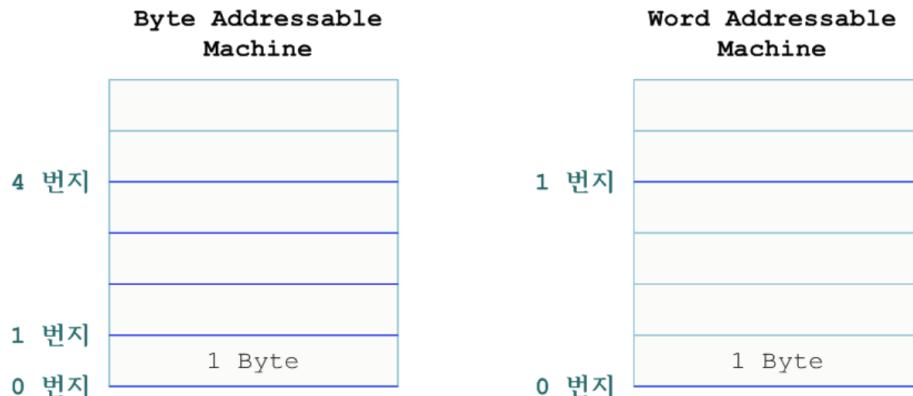


- Control Bus: 읽기/쓰기 제어
- Address Bus: 접근할 메모리 주소를 실어 나름.
- Data Bus: 읽고 쓸 데이터나 명령어를 실어 나름.

## 레지스터, 워드 크기

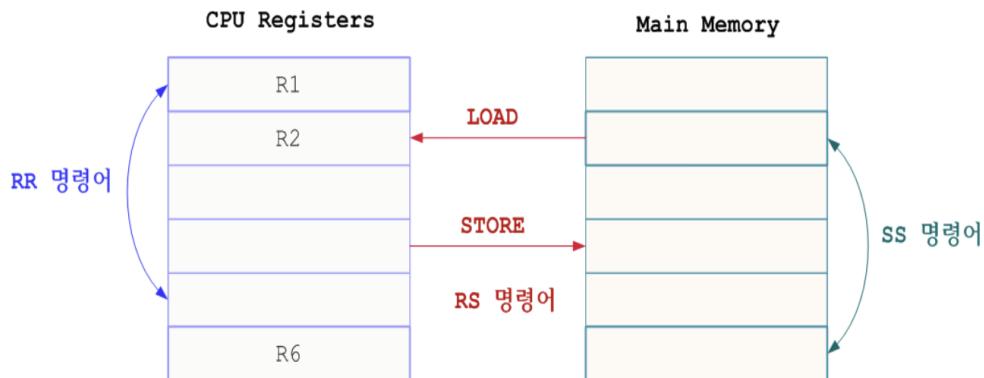
- 레지스터의 종류
  - MAR (Memory Address Register): 주소 저장
  - MBR (Memory Buffer Register): 데이터 저장
  - IBR (Instruction Buffer Register): 명령어 저장
  - PC (Program Counter): 다음 실행할 명령어 주소 저장
  - General Purpose Registers: 범용 레지스터
- 워드 (Word)
  - 데이터 버스의 크기.
  - 대개 어ドレス 버스 크기 및 정수형의 크기와 일치
- 32 비트 프로세서 (데이터 버스 크기가 32 비트)
  - 1 Word = 1 Full Word = 32 bit = 4 Bytes
  - Half Word = 2 Bytes, Double Word = 8 Bytes
  - cf. 64 비트 프로세서: 1 Word = 8 Bytes

## 메모리 주소



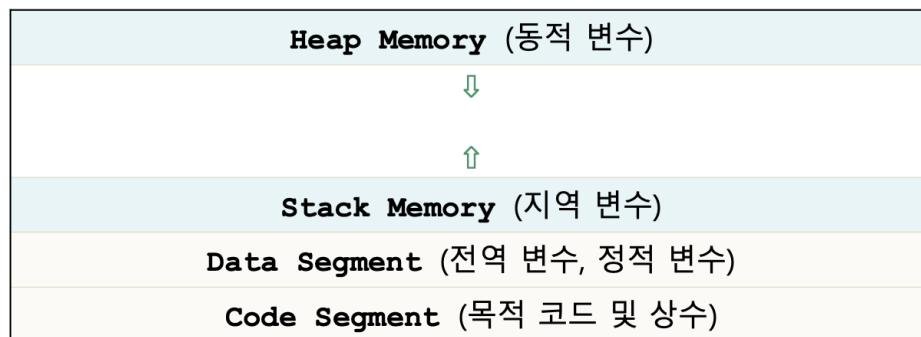
- 주소 지정 방식
  - 바이트 단위 주소 지정 vs. 워드 단위 주소 지정
- 바이트 단위 주소 지정
  - 32 비트 프로세서:  $2^{32}$  Bytes (4GB) 주소 공간
  - 64 비트 프로세서:  $2^{64}$  Bytes 주소 공간

## 어셈블리 명령어



- RR: Register to Register
- RS: Register to Storage or Storage to Register
  - **LOAD M(X)**: 메모리 x 번지 내용을 레지스터에 저장
  - **STORE M(X)**: 레지스터 내용을 메모리 x 번지에 저장
- SS: Storage to Storage

## 프로그램 실행 중 메인 메모리 구조



- 코드 세그먼트
  - 목적 코드 형태의 프로그램과 상수 데이터를 저장
  - 읽기 전용 (Read Only)
- 스택 메모리
  - 함수 내부에 선언된 지역 변수 저장
- 힙 메모리
  - 프로그램 실행 중 만들어진 동적 변수 저장

## 심볼 테이블

### ● Example 2-9 실습 및 해설

변수명	타입	크기	주소	주소	값
grade	char	1	1000	1000	0004 0001 (= 41 <sub>hex</sub> )
sum	Int	4	1004	1001	
				1002	
				1003	
				1004	0100 0000 (= 40 <sub>hex</sub> )
				1005	0011 0000 (= 30 <sub>hex</sub> )
				1006	0010 0000 (= 20 <sub>hex</sub> )
				1007	0001 0000 (= 10 <sub>hex</sub> )

- 심볼 테이블과 메인 메모리

ex 2-9 :

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

```

int main () {
    char grade;
    int sum;

    grade = 'A';
    sum = 0x40302010;
    printf("Grade starts at address %p.\n", &grade);
    printf("Sum starts as address %p.\n", &sum);

    return 0;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev $ ./example_2-9
Grade starts at address 0x16b65ef5b.
Sum starts as address 0x16b65ef54.

```

## 빅 엔디언과 리틀 엔디언

### Example 2-10 실습 및 해설

- 16진수 0x40302010
  - Big Endian: 40, 30, 20, 10 순으로 저장
  - Little Endian: 10, 20, 30, 40 순서로 저장

ex 2-10:

```

#include <stdio.h>

int main () {
    int i, sum;

    sum = 0x40302010;
    unsigned char * p = (unsigned char*)&sum;
    printf("Sum is %x.\n", sum); // 1
    for (i = 0; i < 4; i++)
        printf("%p\t%#x\n", p+i, p[i]); // 2
    printf("\n");

    return 0;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev $ ./example_2-10
Sum is 40302010.
0x16b846f44      10
0x16b846f45      20
0x16b846f46      30
0x16b846f47      40

```

변수가 1바이트보다 클 경우 어떤 순서로 값을 저장할 것인지도 관건이다. 16진수 0x40302010을 바이트 단위로 저장할 경우 낮은 주소에서 높은 주소인 40, 30, 20, 10의 순으로 차례로 저장하기도 한다. 이처럼 높은 자리 수부터 먼저 저장하는 방식을 빅 엔디언이라 부른다. 이와는 반대로 10, 20, 30, 40의 순서로 저장하기도 한다. 이처럼 낮은 주소에서 높은 주소로 가면서 낮은 자리 수부터 먼저 저장하는 방식을 리틀 엔디언이라 부른다.

자신의 컴퓨터가 리틀 엔디언과 빅 엔디언 중 어떤 방식을 쓰는지 알아보기 위한 프로그램이다. //1에 의해 sum 변수 값이 %x(hexadecimal), 즉 16진수 형식으로 찍힌다.

이어 //2에 의해 메모리 주소별로 저장된 값이 바이트 단위로 찍힌다.

현재로서는 이 프로그램을 이해할 필요가 없다. 다만, 이 프로그램을 통해 메모리 몇 번지에 어떤 값이 들어가 있는지 확인해 보면 된다. 이처럼 메모리에 있는 내용을 있는 그대로 화면에 뿌리는 작업을 메모리 덤프라고 부른다.

## 2-4. 표준 입출력 함수

### 입출력 함수

#### Example 2-11 실습 및 해설

```
printf("%d green %s are on the table.\n", 10, "apples");
```

- `printf` for 'formatted' print
  - 큰 따옴표 안에 제어 문자열
  - %는 형식 지정자
  - 인자가 여럿이면 순차적으로 들어감

ex 2-11 :

```
#include <stdio.h>

int main () {
    int sum, count;

    sum = 0x01020304;
    printf("Sum is %d in decimal.\n", sum);
    printf("%d green %s are on the table\n", 10, "apples");
    count = printf("Hello.\n");
    printf("printf returns %d.\n", count);
```

```
    return 0;  
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch2 $ ./example_2-11  
Sum is 16909060 in decimal.  
10 green apples are on the table  
Hello.  
printf returns 7.
```

printf 함수에도 리턴 값이 있다. 화면에 찍은 문자의 수를 돌려준다. 리턴 값을 count 변수에 대입하여 print 하고 있다. 마침표는 물론 \n도 문자 수에 포함된다.

## 형식 지정자

형식 지정자	의미
%d	decimal
%ld	long decimal
%lld	long long decimal
%u	unsigned int
%f	float
%lf	double
%x (X)	hexadecimal (capital)
%e (E)	exponent (capital)
%c	character
%s	character string
%p	pointer

- printf 문의 형식 지정자
  - 화면 입력은 모두가 문자
  - 형식 지정자 = **변환 코드** (읽으면서 변환)

printf 함수는 전달된 값을 형식 지정자에 맞게 변환하여 출력한다. 예를 들어 10진수 31을 %d로 출력하면 31 이 보이지만 %x로 출력하면 1f가 보인다. 그런 점에서 형식 지정자를 변환 코드라고 부르기도 한다. 위 그림은 printf 문에서 사용할 수 있는 형식 지정자이다. %x 대신 %라고 하면 A, B, ..., F 식으로 대문자로 찍는다.

## printf

### Example 2-12 실습 및 해설

ex 2-12 :

```
#include <stdio.h>  
  
int main () {  
    float f;
```

```

    double d;
    int i;

    f = 0.123456789123456789F; // 1
    d = 0.123456789123456789; // 2
    printf("float f is %f.\n", f); // 3
    printf("double d is %lf.\n", d); // 4
    printf("float in .20f is %.20f.\n", f); // 5
    printf("double in .20lf is %.20lf.\n", d); // 6
    printf("double in 25.4lf is \n%25.4lf.\n", d); // 7
    printf("double in -25.4lf is \n%-25.4lf.\n", d); // 8
    i = 365;
    printf("int in -8d is %-8d.\n", i); // 9
    printf("Exponent form of 123456.78 is %e.\n", 123456.78); // 10

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch2 $ ./example_2-12
float f is 0.123457.
double d is 0.123457.
float in .20f is 0.12345679104328155518.
double in .20lf is 0.12345678912345678380.
double in 25.4lf is
    0.1235.
double in -25.4lf is
    0.1235
int in -8d is 365 .
Exponent form of 123456.78 is 1.234568e+05.

```

double은 float보다 자릿수를 늘려 잡아야 보인다. //1의 우변에 F를 붙여야 float로 인식된다. 그렇지 않으면 디폴트인 double로 인식된다.

//2의 우변은 디폴트인 double로 인식되기 때문에 아무 것도 붙일 필요가 없다.

//3의 %f나 //4의 %lf 모두 디폴트로 소수점 아래 6자리까지 찍는다.

//5, //6처럼 소수점 아래 자릿수를 20자리로 늘려야 그 차이를 알 수 있다. 반올림까지 감안하여 float 형은 소수 이하 7자리까지만 정밀도를 보장한다. 따라서 나머지 숫자는 무의미하다. 반면, double 형은 소수점 이하 16자리까지 정밀도를 보장한다. long double 형은 소수점 이하 33자리까지 정밀도를 보장한다.

//7의 %25.4lf는 소수점 이하 4자리까지 표현하되 소수점을 포함하여 25자리로 표현하라는 것이다. 디폴트는 오른쪽 줄맞춤이다.

//8의 - 기호는 왼쪽 줄맞춤을 의미한다. 이는 //9의 정수 변수에 대해서도 마찬가지이다.

부동 소수의 크기가 클 경우 //10 처럼 10의 승수로 표혀하는 것이 유리하다. %e 대신 %E라 쓰면 출력 기호도 E로 바뀐다.

```
printf("%f", 44);
```

- 뭐든 변환이 가능하지는 않다.

뭐든 변환이 가능한 것은 아니다. 예의 경우 컴파일 오류는 나지 않지만 실행 결과 잘못된 값이 출력될 가능성이 크다. 정수 44를 부동 소수 형식으로 출력하라고 했기 때문이다. 2의 보수로 표현된 정수를 부호부, 지수부, 가

수부로 나뉜 부동 소수 형식으로 바라보면 오류가 일어나는 것은 당연한 일이다. 이런 점을 감안한다면 변수의 자료형에 부합하는 형식 지정자를 쓰거나 변환이 가능한 형식 지정자를 써야한다.

```
int printf(const char* format, arg1, arg2, ...);
```

- **printf 함수 원형** (Function Prototype)
  - 리턴 타입, 함수 명, 인자
  - `const char*`는 `format`(제어 문자열)의 자료형

## scanf

### Example 2-13 실습 및 해설

```
int scanf(const char *format, &arg1, &arg2, ...);
```

- `&` = 주소 연산자
  - `scanf`: 변수 명 앞에 반드시 주소 연산자
  - `&age` = address of `age` = `age` 변수의 주소
- 가변 인자 함수 (Variadic Function)
  - `printf`, `scanf`
  - 인자 (Argument)의 개수가 가변

ex 2-13 :

```
#include <stdio.h>

int main () {
    int age, count;
    double weight, height;

    printf("Enter age.\n");
    scanf("%d", &age);
    printf("Age : %d.\n", age);
    printf("Enter weight and height.\n");
    scanf("%lf%lf", &weight, &height);
    printf("Weight : %lf, Height : %lf.\n", weight, height);
    printf("Enter weight and height again.\n");
    count = scanf("%lf%lf", &weight, &height);
    printf("scanf returns %d.\n", count);
```

```

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch2 $ ./example_2-13
Enter age.
22
Age : 22.
Enter weight and height.
190 90
Weight : 190.000000, Height : 90.000000.
Enter weight and height again.
190 90
scanf returns 2.

```

**scanf 경고 메시지 없애기**

The screenshot shows the Microsoft Visual Studio IDE. In the top-left, there's a code editor window titled "project1" containing C code. The code includes a printf statement and a scanf statement. In the top-right, there's a "main()" tab. Below the code editor is the "Output" window, which displays build logs for "test1.c". It shows several warnings from the compiler (C4996) about the use of unsafe functions like "scanf". At the bottom, there's a properties manager window titled "project1 속성 페이지" (Properties Manager). The "Preprocessor Definitions" section contains the value "MBCS:(%PreprocessorDefinitions);\_CRT\_SECURE\_NO\_WARNINGS".

- 프로젝트 → 속성 → C/C++ → 전처리기  
→ [\\_CRT\\_SECURE\\_NO\\_WARNINGS](#)

## scanf

- 화면에 보이는 모든 것은 문자
  - 2는 '2'
  - %c로 읽으면 '2'
  - %d로 읽으면 숫자 2로 자동 변환(`atoi` 함수)

### Example 2-14 실습 및 해설

- 공백 문자
  - 탭, 빈칸, 이스케이프, 엔터
  - scanf는 숫자 사이를, 공백 문자를 통해 구분

```
scanf("%d\n", &age); // Remove \n
scanf("%lf %lf", &weight, &height); // Remove blank
scanf("%lf,%lf", &weight, &height); // Remove comma
```

ex 2-14 :

```
#include <stdio.h>

int main () {
    unsigned char ch;
    int num;

    printf("Enter a character.\n"); // 1
    scanf("%c", &ch); // 2
    printf("It is %d in decimal.\n", ch); // 3
    printf("Enter the same character again.\n"); // 4
    scanf("%d", &num); // 5
    printf("It is %d in decimal.\n", num); // 6 ←

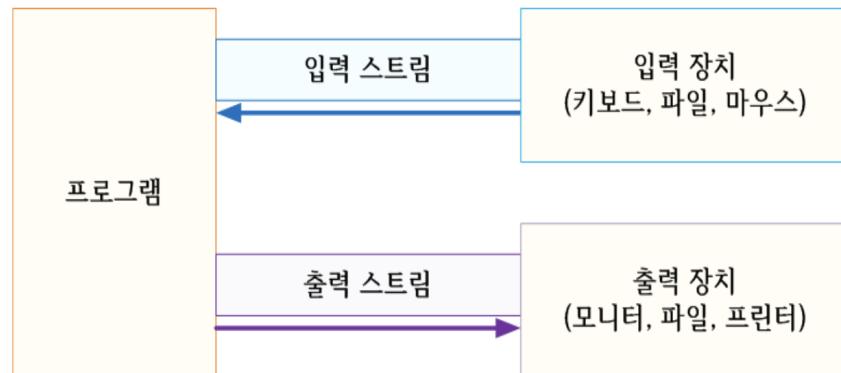
    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev/ch2 $ ./example_2-14
Enter a character.
7
It is 55 in decimal.
Enter the same character again.
7
It is 7 in decimal.
```

//1의 프롬포트에서 숫자 7을 입력하면 이는 문자 7로 간주된다. 이 문자는 //2의 %c에 의해 그대로 문자 '7'로 읽힌다. //3에서 보듯이 문자 '7'은 아스키코드 숫자로는 55다.

//4의 프롬포트에서 또 다시 숫자 7을 입력하면 이 역시 문자 '7'이다. 그러나 이번에는 //5의 %d에 의해 atoi 함수가 호출되어 문자 '7'이 숫자 7로 변환된다. //6에서 다시 변환이 일어난다. 숫자 8을 %d 형식으로 찍더라고 화면에 찍으려면 itoa 함수를 호출하여 문자 '7'로 변환해야 하기 때문이다.

## 입출력 스트림



- 프로그램과 장치 사이의 일대일 통로
- 표준 입출력 스트림
  - 프로그램 실행 시 자동 개방
  - `stdin` (Standard Input Stream) : Keyboard
  - `stdout` (Standard Output Stream) : Monitor
  - `stderr` (Standard Error Stream) : Monitor

## 입출력 버퍼



- 버퍼
  - 버퍼 메모리. 스트림 내부에 존재
  - 입출력 장치와 CPU의 속도 차이 극복

위는 입력 버퍼의 모습이다. 키보드에서 문자 하나하나를 칠 때마다 개별 문자가 버퍼(구체적으로는 키보드 버퍼)에 쌓인다. 물론 CPU 입장에서 볼 때 이 작업은 엄청나게 느린다. 따라서 CPU는 입력이 끝날 때까지 기다리지 않고 다른 일을 처리한다. 이제 입력이 끝났으니 운영체제에 지금까지 버퍼에 입력한 내용을 처리해달라고 요구하는 것이 엔터 키다. 이런 점에서 엔터 키를 방아쇠라고도 부른다. 만약 프로그램에서 `scanf` 함수를 호출했으면 엔터 키는 이제 입력을 읽어 변수에 저장해달라는 요구다.

## scanf

### Example 2-15 실습 및 해설

### Example 2-16 실습 및 해설

- %c로 읽으면 Blank나 Enter도 하나의 문자로 취급
- Enter가 남아있을 경우 Buffer Clearing이 필요
- cf. %d로 읽으면 공백 문자를 건너뛴다.

ex 2-15 :

```
#include <stdio.h>

int main () {
    char ch1, ch2;
    printf("Enter two characters.\n");
    scanf("%c%c", &ch1, &ch2);
    printf("Characters just read are %c and %c.\n", ch1, ch2);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch2 $ ./example_2-15
Enter two characters.
Q R
Characters just read are Q and .
```

ex 2-16 :

```
#include <stdio.h>

int main () {
    char ch1, ch2;
    printf("Enter the first character.\n");
    scanf("%c", &ch1);
    printf("First character is %c.\n", ch1);

    printf("Enter the second character.\n");
    scanf("%c", &ch2);
    printf("Second character is %c.", ch2);

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch2 $ ./example_2-16
Enter the first character.
Q
First character is Q.
Enter the second character.
Second character is
.
```

▼ 09/26 목 - 3장

## 3장. 대입문과 연산자

### 3-1. 대입문

#### 대입문

```
int my_age;
my_age = 20;
```

- 대입문, 할당문
- 등호는 같다는 의미가 아니라 집어 넣으라는 의미

#### L-Value, R-Value

```
int my_age;
my_age = 20;
my_age = 24 + 1;
```

- 주소가 있으면 L-Value, 없으면 R-Value(상수, 연산식)

```
int my_age, your_age, sum;
your_age = 19;
my_age = your_age;
my_age = my_age + 1;
sum = my_age + your_age;
```

```
int my_age;
20 = my_age;           // Error
my_age + 1 = 20;      // Error
```

- 변수

- 대입문의 좌변에 오면 L-Value(변수 그 자체)
- 대입문의 우변에 오면 R-Value(변수의 값)
- 대입문의 좌변에는 L-Value만 가능

## 대입 연산과 연산식

```
int salary;  
salary = 400;  
salary = salary + 20;
```

- 대입은 시차를 두고 이루어지는 작업
  - 우변 값을 계산한 후에 좌변에 대입

$$\begin{array}{c} \text{연산식} \\ \hline 2 * a + 3 * b * c \\ \hline \text{항} & \text{인수} \end{array}$$

- 연산식 (표현식, 수식, Expression): **R-Value**
- 항 (Term), 인수 (Factor)

## 변수 초기화

```
int count, sum = 20;
```

- 변수 선언과 동시에 **초기화** (Initialization) 가능

### Example 3-1 실습 및 해설

- 초기화에 사용된 등호 ('=')는 대입 연산자가 아니다.
  - 초기화: 변수에 메모리를 할당하면서 동시에 우변 값을 넣음.
  - 대입: 프로그램 실행 도중 변수를 찾아가서 우변 값을 넣음.

업무는 초기화와 대입이 다른 특성을 활용해 동시에 우변의 값을 넣으려면  
제작자는 초기화에 대해서는 해당 변수의 값을 초기화하고 대입은 대입으로  
다시 말하자면 초기화는 반드시 같이 제작자에게 활용되거나 아니면 실행 시에 확장되거나해야 한다.  
이런 경에서 초기화에 사용된 등호 ('=')는 대입 연산자가 아니라 등호를 사용하는 이후는 단지 마지막은 기호가 없기 때문이다.

### example\_3-1.c :

```
#include <stdio.h>  
  
int main () {  
    int my_age, your_age;
```

```

int her_age = 20;

my_age = her_age + 1;
printf("My age is %d.\n", my_age);

// my_age = your_age + 1; // 오류
// printf("My age is %d.\n", my_age); // 오류

return 0;
}

```

```
(base) minsung@iminseong-Ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev/ch3 $ ./example_3-1
My age is 21.
```

## 연산식과 대입문의 리턴 값

### Example 3-2 실습 및 해설

- `printf("%d\n", bonus + salary);`
- 연산식은 결과 값을 리턴

```

bonus = salary = 400;
salary = 400; bonus = salary;
• 대입문은 좌변 값을 리턴

```

### Example 3-3 실습 및 해설

- C는 절차적 언어 (Procedural Language)

#### **example\_3-2.c :**

```

#include <stdio.h>

int main () {
    int salary, bonus, total;

    salary = 400;
    bonus = 20;
    salary + bonus;
    total = salary + bonus;
    printf("Total is, %d\n", total);

    bonus = salary = 400;
}

```

```

    printf("Bonus plus salary is, %d\n", bonus + salary);

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-2
Total is, 420
Bonus plus salary is, 800

```

```

bonus = (salary = 400); // a
salary = 400; bonus = salary; // b

```

`bonus = salary = 400;`는 위와 같은 의미이다. 대입문을 실행하면 좌변 변수의 값을 리턴한다. `salary = 400;`을 먼저 실행하면 좌변 `salary`의 값인 400이 리턴되고 그 값이 `bonus`에 대입된다. 사실상 이는 `//b`처럼 연이은 대입문으로 쓸 수 있다. 그럼에도 불구하고 `//a`처럼 쓰는 이유는 C언어가 짧은 표현을 선호하기 때문이다. 대입이 계속되면 오른쪽에서 왼쪽으로 가면서 대입된다. 예를 들어, `a = b = c = d;`라고 하면 이는 `a = (b = (c = d));`와 같은 의미다.

### example\_3-3.c :

```

#include <stdio.h>

int main () {
    int one, two, three;

    one = 1;
    // three = two + 1; // 오류
    // two = one + 1; // 오류
    // printf("one : %d, two : %d, three : %d.\n", one, two, three);

    one = 1;
    two = one + 1;
    three = two + 1;
    printf("one : %d, two : %d, three : %d.\n", one, two, three);

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-3
one : 1, two : 2, three : 3.

```

절차적 언어에서는 명령문의 순서가 중요하다.

## 3-2. 산술 연산자

## 산술 연산

$$\begin{array}{ccc} & \text{연산자} & \\ a & + & b \\ \text{피 연산자 1} & & \text{피 연산자 2} \end{array}$$

- 연산자와 피연산자

### Example 3-4 실습 및 해설

- 산술 연산: +, -, /, \*, %
- DIV (/): 정수연산이면 몫 (cf. 부동소수 연산)
- MOD (%):  $10 \% 3 = 1$ . 자동차 미터기. 시계 바늘

```
a - b  
= a + (256 - b)  
= a + (255 - b) + 1
```

- 8 비트 연산이면  $(256 + n) \% 256 = n$ .
- $(255 - b)$ 는 b에 대한 1의 보수. 1을 더하면 2의 보수

### example\_3-4.c :

```
#include <stdio.h>

int main () {
    int a = 10, b = 3;
    double p = 10.0, q = 3.0;

    printf("10 + 3 = %d.\n", a + b);
    printf("10 - 3 = %d.\n", a - b);
    printf("10 * 3 = %d.\n", a * b);
    printf("10 / 3 = %d.\n", a / b);
    printf("10 %% 3 = %d.\n\n", a % b);

    printf("10.0 + 3.0 = %f.\n", p + q);
    printf("10.0 - 3.0 = %f.\n", p - q);
    printf("10.0 * 3.0 = %f.\n", p * q);
    printf("10.0 / 3.0 = %f.\n", p / q);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev/ch3 $ ./example_3-4
10 + 3 = 13.
10 - 3 = 7.
10 * 3 = 30.
10 / 3 = 3.
10 % 3 = 1.

10.0 + 3.0 = 13.000000.
10.0 - 3.0 = 7.000000.
10.0 * 3.0 = 30.000000.
10.0 / 3.0 = 3.333333.
```

- 정수 연산에는 사칙연산 이외에 모듈로 연산이 제공된다. 이는 나머지를 구하는 연산으로서 % 기호로 표시하고 모드 또는 모듈로라고 읽는다. 이는 정수 사이에서만 가능한데, 부동 소수 나눗셈에는 나머지가 있을 수 없기 때문이다.
- 사실상 뺄셈도 모듈로 연산에 의해 이루어진다.

```
a - b
= a + (256 - b) // 1
= a + (255 - b) + 1 // 2
```

예를 들어, 8비트 연산이라면 표현 가능한 양의 정수는 1부터 255까지다. 숫자가 하나 증가하여 256에 이르면 그것은 0으로 표현된다. 이후 다시 1, 2, ... 순으로 증가한다. 따라서 숫자 316은 8비트로는 60으로 표현된다.  $316 \% 256 = (256 + 60) \% 256 = 60$ 이기 때문이다. 이는 8비트로 표시되는 숫자는 몇 번이고 거기에 256을 더해도 같은 숫자임을 의미한다. 그런 점에서 //1도 사실이다. 그런데 여기서 중요한 것은 //2의  $(255 - b)$ 가  $b$ 에 대한 1의 보수라는 점이다. 예를 들어  $b$ 가 69(0100 0101)라면 255(1111 1111)에서  $b$ 를 뺀 결과는 1011 1010으로서  $b$ 에 대한 1의 보수가 된다. 1에서 0이나 1을 빼면 값이 뒤집히기 때문이다.  $(255 - b)$ 가 1의 보수라면 거기에 1을 더한  $(255 - b) + 1$ 이 2의 보수다. 결국  $a$ 에서  $b$ 를 빼는 연산은 //2에서 보듯이  $a$ 에다가  $b$ 에 대한 2의 보수를 더한 연산으로 바뀐다.

#### 1. 빼기 연산을 덧셈으로 변환:

- $a - b$  같은 빼기 연산을 실제로는  $a + (256 - b)$ 로 계산할 수 있다는 내용입니다. 여기서 256은 8비트로 표현할 수 있는 최대 값인 255보다 1 큰 값이에요.
- 예를 들어, 8비트로 숫자를 표현하면 가능한 범위는 0부터 255입니다. 만약  $b$ 를 빼는 대신,  $b$ 의 1의 보수(컴퓨터에서 값을 뒤집은 것)를 더하면 같은 결과를 얻을 수 있습니다.

#### 2. 예시:

- 8비트로 숫자 316을 표현하려면,  $316 \% 256$ 을 계산하면 60이 됩니다. 즉, 8비트로 316은 60과 동일하게 표현된다는 얘기입니다.
- 그러면, 예를 들어  $a$ 에서  $b$ 를 빼는 연산을 할 때,  $255 - b$ 에 1을 더해  $a$ 에 더해 주는 방식으로 빼기 연산을 덧셈으로 변환할 수 있습니다.

이 내용은 컴퓨터에서 2의 보수(2's complement) 방식을 이용해 빼기 연산을 덧셈으로 처리하는 방법에 대한 설명이에요.

## 오버플로우

### Example 3-5 실습 및 해설

- 정수 오버플로우: 순환 값으로 대치 ( $32767 + 1 = -32768$ )
- 부동 소수 오버플로우: INF를 리턴

### Example 3-6 실습 및 해설

```
a += b; means a = a + b;  
a -= b; means a = a - b;  
a *= b; means a = a * b;  
a /= b; means a = a / b;
```

- 복합 대입 연산자
- 단항 연산자 (-age, &age), 2항 연산자 (a + b, a - b)

#### example\_3-5.c :

```
#include <stdio.h>  
  
int main () {  
    short a = 32768;  
    short b = a / 2;  
    float c = 1E45;  
    float d = c / 2.0;  
  
    printf("a: %d, b: %d, c: %f, d: %f.\n", a, b, c, d);  
  
    return 0;  
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch3 $ ./example_3-5  
a: -32768, b: -16384, c: inf, d: inf.
```

2의 보수 체계에 의하면 정수가 주어진 비트 수를 초과하면 순환이 일어난다. 예를 들어 2바이트 short가 수용 할 수 있는 정수는 -32768에서 32767까지다. 그런데 예에서 보듯이 32767에 1을 더하여 a 변수에 32768을 넣으면 그 값의 반대쪽 끝인 -32768로 되돌아간다. 음수에 대해서도 마찬가지다. -32768에서 1을 빼면 32767로 되돌아간다. 이는 마치 -32768에서 32767까지 정수가 아날로그 시계의 눈금처럼 분포한 상태에서 바늘이 한 바퀴를 완전히 돌면 처음 숫자로 가는 것과 같다. 그런 점에서 정수 순환도 모듈로 연산이라 할 수 있다.

정수 오버플로우에는 오류 메시지가 없다.

정수가 오버플로우를 일으키더라도 컴파일 시점에는 오류 메시지가 뜨지 않는다. 나아가 실행 도중에도 오류 메시지가 뜨지 않는다. 실행 중 오버플로우가 일어나면 순환시킨 값으로 대치할 뿐이다. 아무런 오류 메시지가 없으므로 이 경우 연산의 결과가 올바른지 판단하는 것은 온전히 프로그래머의 책임이다. 그러나 부동 소수가 오버플로우를 일으키면 INF(Infinity)라는 문자열을 출력하기 때문에 정수에 비해서는 디버깅이 조금 더 수월하다.

```
int a; double b;
a = 5 / 10; // 1
b = 5 / 10.0; // 2
```

피연산자의 자료형이 서로 다르면 연산 과정에서 형 변환이 일어난다는 점에도 유의해야한다. //1의 a에는 0이 들어간다. 정수끼리의 연산이므로 나눗셈의 결과로 몫이 0이 되기 때문이다. 하지만 //2의 b에는 0.5가 들어간다. 문자가 int 형인데 비해 분모는 double 형이기 때문이다. 형 변환은 대개 더 큰 쪽을 향해서 일어나므로 우변은 (double)5 / 10.0으로 바뀐다.

#### example\_3-6.c :

```
#include <stdio.h>

int main () {
    int a, b;

    a = 8;
    a += 2;
    printf("a = %d.\n", a);

    b = 2;
    a /= b;
    printf("a = %d.\n", a);

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev/ch3 $ ./example_3-6
a = 10.
a = 5.
```

C는 짧은 표현을 선호하기 때문에 예처럼 줄여서 쓰기도 한다. 예에서 보듯이 산술 연산자와 대입 연산자가 붙어 있기 때문에 이를 복합 대입 연산자라고도 부른다.

피연산자가 둘인 연산자를 2항 연산자라 부른다. 예를 들어 대입 연산자나 복합 대입 연산자는 2항 연산자다. 좌변과 우변에 피연산자가 있기 때문이다. 반면에 피연산자가 하나인 연산자를 단항 연산자라 부른다. 예를 들어 주소 연산자인 &가 단항 연산자다. -25에서 -도 단항 연산자다. 피연산자가 25 앞에 붙어 값을 음수로 만들기 때문이다. +25에서 +도 단항 연산자다. 그러나 +는 양수를 의미하기 때문에 생략해도 아무런 문제가 되지 않는다.

## 증감 연산자: 전위와 후위

### Example 3-7 실습 및 해설

- 증가 연산자
  - 전위 증가:  $++a$ ; 후위 증가:  $a++$ ;
- 감소 연산자
  - 전위 감소:  $--a$ ; 후위 감소:  $a--$ ;

①  $a = a + 1$ ; ②  $a += 1$ ; ③  $a++$ ; ④  $++a$ ;

- 1 만큼 증가시키는 4가지 방법

#### example\_3-7.c :

```
#include <stdio.h>

int main () {
    int a, b;

    a = 0;
    a++;
    printf("a = %d.\n", a);
    b = a++;
    printf("a = %d. b = %d.\n", a, b);

    a = 0;
    ++a;
    printf("a = %d.\n", a);
    b = ++a;
    printf("a = %d. b = %d.\n", a, b);

    a = b = 0;
    printf("a = %d. b = %d.\n", (1 + a++) + 2, ++b);
    printf("a = %d. b = %d.\n", a, b);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-7
a = 1.
a = 2. b = 1.
a = 1.
a = 2. b = 2.
a = 3. b = 1.
a = 1. b = 1.
```

### 3-3. 관계 연산자와 논리 연산자

#### 관계 연산자

관계 연산자	의미
>	greater than
$\geq$	greater than or equal to
<	less than
$\leq$	less than or equal to
$\equiv$	equal to
$\neq$	not equal

- 서로 같은지 비교하려면 등호를 두 번 써야 한다.
- 관계 연산의 결과는 1(true) 또는 0(false)다.

#### Example 3-8 실습 및 해설

##### example\_3-8.c :

```
#include <stdio.h>

int main () {
    int a = 10, b = 5;
    double p = 10.0, q = 5.02, r = 10.000000000000001;

    printf("%d\n", a > b);
    printf("%d\n", a >= b);
    printf("%d\n", a == b);
    printf("%d\n", a != b);

    printf("%d\n", p > q);
    printf("%d\n", p == q);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기/시스템프로그래밍기초/dev/ch3 $ ./example_3-8
1
1
0
1
1
1
```

```
if (2.5 * (1.0 / 3.0) == 5.0 / 6.0) // 1
if ((a - b) > 0.0001) // 2
```

부동 소수끼리 같은지 비교하는 일은 피하는 것이 좋다. 부동 소수끼리 서로 같은지를 비교해서는 안 된다. //1의 조건식 좌변의 분모와 분자에 2를 곱하면 우변과 같아 지지만 이는 참이 아니라 거짓이다. 좌변과 우변 값이 차이가 나므로 좌변에서 우변을 빼면 -0.0000000000000001이 되기 때문이다. 부동 소수 계산에는 항상 오차가 따른다. 비트 수가 제한되어 있기 때문에 계산 도중 결과 값이 반올림되거나 잘려나가기 때문이다. 이런 점을 감안한다면 부동 소수끼리는 가급적 서로 같은지 비교하지 말아야 한다. 굳이 원한다면 //2처럼 해 볼 수는 있다. a와 b가 같은지 비교하는 대신 둘 사이의 차이가 일정한 오차 범위 내인지를 비교했기 때문이다.

논리 연산자								
a	NOT a	a b	a AND b	a b	a OR b	a b	a XOR b	
0	1	0 0	0	0 0	0	0 0	0	
1	0	0 1 1 0	0 0	0 1 1 0	1 1	0 1 1 0	1 1	
		1 1	1	1 1	1	1 1	0	

- Boolean Algebra: 논리 연산 표현을 위한 대수학

논리 연산자	의미
&&	logical AND
	logical OR
!	logical NOT

- C 언어는 세 가지 논리 연산을 제공.
- 논리 연산은 0(false) 아닌 모든 값은 1(true)로 간주.

### Example 3-9 실습 및 해설

#### example\_3-9.c :

```
#include <stdio.h>

int main() {
    int a = 4, b = 3, c = 2, d = 1;

    printf("%d \n", (a > b) && (c > d));
    printf("%d \n", (a < b) || (c > d));
    printf("%d \n", (a < b) && (c > d));
```

```

printf("%d \n", (a > b) || (c > d));
printf("%d \n", (a < b || c > d) && (a == b && c >= d));

printf("%d \n", !(a > b));
printf("%d \n", !d);
printf("%d \n", !a);

return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-9
1
1
0
1
0
0
0
0
0
0

```

## 비트 연산자

비트 연산자	의미
&	bitwise AND
	bitwise OR
^	bitwise XOR
~	bitwise NOT
<<	bitwise LEFT SHIFT
>>	bitwise RIGHT SHIFT
<hr/>	
bitwise AND	bitwise OR
bitwise XOR	bitwise NOT
<hr/>	
0110 1001 (69 <sub>hex</sub> )	0110 1001
0101 0101 (55 <sub>hex</sub> )	0101 0101
0100 0001 (41 <sub>hex</sub> )	0111 1101
<hr/>	
0110 1001	0101 0101
0110 1001	0101 0101
1001 0110	1001 0110

- Example 3-10 실습 및 해설
- Example 3-11 실습 및 해설
- Example 3-12 실습 및 해설

C언어는 고급 언어로서의 기능뿐만 아니라 어셈블리 언어에 가까운 저급 언어로서의 기능도 지닌다. 그 중 하나가 비트 연산자다. 위 그림에서 보듯이 C언어는 6가지 비트 연산자를 제공한다. AND와 OR 기호가 논리 연산과는 다르다는 점에 주의해야 한다. 논리 연산의 AND가 &&라면 비트 연산의 AND는 &다. 또, 논리 연산의 OR이 ||라면 비트 연산의 OR은 |다. 이 연산자들은 부동 소수형 데이터에는 적용되지 않고 정수형 데이터에만 적용된다. signed나 unsigned에 모두 적용되며 char, short, int, long 타입에 모두 적용할 수 있다. 비트 연산을 하려면 위 그림처럼 부울 대수의 규칙을 비트 단위로 적용해야 한다.

**example\_3-10.c :**

```
#include <stdio.h>

int main () {
    int a = 105, b = 85;

    printf("%d \n", a & b);
    printf("%X \n", a | b);
    printf("%X \n", a ^ b);
    printf("%X \n", ~a);

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-10
65
7D
3C
FFFFF96
```

a는 16진수로 69, b는 16진수로 55다. 따라서  $a \& b = 69_{hex} \& 55_{hex} = 0110\ 1001 \& 0101\ 0101 = 0100\ 0001$ 이 된다. 이는 16진수로 41이므로 10진수로는 65다.

논리 연산과 비트 연산을 혼동하면 안된다. 논리 연산의 결과는 항상 0(false) 아니면 1(true)이다. 예를 들어 논리 연산인  $3 \&& 2 = 1 \&& 1 = 1$ 이다. 비트 연산은 그렇지 않다.  $3 \& 2 = 0011 \& 0010 = 0010 = 2$ 가 되어 일정한 값을 갖는다.

비트 연산이 유용할 때도 많다. 사칙연산 중 가장 비싼 연산, 즉 가장 시간이 오래 걸리는 연산이 나눗셈이다. 덧셈을 반복함으로써 곱셈이 이루어진다면 곱셈을 반복함으로써 이루어지는 것이 나눗셈이기 때문이다. 예를 들어 8비트 정수형에서  $105 \% 32$ 를 계산하려 한다고 가정해 보자. 물론 이 연산을 한 번만 수행한다면 속도는 그리 문제되지 않는다. 따라서 그냥  $105 \% 32$ 를 계산하면 된다. 그러나 프로그램 내에서 모듈로 연산을 수만 번 또는 수천만 번 반복해야 할 때도 있다. 그 경우에는 당연히 비트 연산에 의해 속도를 높일 필요가 있다.

#### example\_3-11.c :

```
#include <stdio.h>

int main() {
    int a = 105, b = 32;
    printf("105 modulo 32 is %d.\n", a % b);
    printf("105 bitwise AND 31 is %d.\n", a & (b - 1));

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-11
105 modulo 32 is 9.
105 bitwise AND 31 is 9.
```

위와 같이 모듈로 연산을 비트 단위 AND에 의해 구할 수 있다. 어떤 수를 32로 나누면 나머지는 항상 0에서 31까지의 숫자다. 따라서 105를 구성하는 2진수 0110 1001 중 31까지 크기에 해당하는 숫자만 남기고 32 이

상의 숫자는 빼어버리면 된다. 그런데 31은 2진수 0000 1111에 해당한다. 따라서  $105 \% 32 = 105 \& (32 - 1) = 0110\ 1001 \& 0000\ 1111 = 0000\ 1001 = 9$ 로 계산할 수 있다.

이 계산에서 31에 해당하는 0000 1111을 마스크 비트라고 부른다. 31의 상위 4비트가 0000이기 때문에 어떤 수와 AND 연산을 하더라도 결과 값의 상위 4비트도 0000으로 바뀐다. 다시 말해서 105를 구성하는 0110 1001 비트열 중에 상위의 0110은 마스크 비트와의 AND 연산에 의해 0000으로 바뀐다. 여기서 0000 1111을 마스크 비트라고 부르는 이유는 마치 마스크를 써우듯 불필요한 비트에는 0을 써우고 필요한 비트에만 1을 써워서 필요한 값만 추려내기 때문이다. 단, 이 연산은  $32 (= 2^5)$ 처럼 나누는 수가 2의승수일 때만 가능하다.

### example\_3-12.c :

```
#include <stdio.h>

int main () {
    unsigned int a = 25;
    unsigned left, right;

    left = a << 3;
    right = a >> 3;
    printf("left shift: %d, right shift: %d.\n", left, right);

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch3 $ ./example_3-12
left shift: 200, right shift: 3.
```

비트 시프트 연산은 정수의 비트열을 왼쪽 또는 오른쪽으로 이동시키는 연산이다. 정수 25(0001 1001)를 구성하는 비트열을 왼쪽으로 3비트 이동시킨 결과는  $25 \ll 3$  (1100 1000)이다. 왼쪽 이동으로 비게 되는 오른쪽 비트는 0으로 채워진다. 2진수에서 자릿수가 세 자리 높아지면 이는 원래의 값에  $8 (= 2^3)$ 을 곱한 것과 같다. 반면 자릿수가 세 자리 낮아지면 이는 원래의 값을 8로 나눈 것과 같다. 따라서  $2^n$  또는  $2^{-n}$ 을 곱하는 계산은 비트 시프트 연산으로 대체할 수 있다.

그러나 이러한 방식의 연산은 주의를 요한다. 유효 비트가 날아가면 잘못된 결과를 초래하기 때문이다.

연산자	의미	결합 순서
( )	괄호(묶음)	왼쪽에서 오른쪽
[ ]	배열 인덱스	
.	멤버 선택	
->	포인터에 의한 멤버 선택	
++ --	후위 증감	
++ --	전위 증감	왼쪽에서 오른쪽
+ -	단항 연산(부호)	
! ~	논리 NOT, 비트 단위 NOT	
(type)	형 변환	
sizeof	데이터 크기	
&	주소 연산자	
*	참조 연산자	
* / %	2항 연산(곱셈, 나눗셈, 나머지)	왼쪽에서 오른쪽
+ -	2항 연산(덧셈, 뺄셈)	"
<< >>	비트 단위 왼쪽 쉬프트, 오른쪽 쉬프트	"
> >= < <=	관계 연산(크기 비교)	"
== !=	관계 연산(Equal, Not Equal)	"
&	비트 단위 AND	"
^	비트 단위 XOR	"
	비트 단위 OR	"
&&	논리 AND	"
	논리 OR	"
=	대입(할당)	오른쪽에서 왼쪽
+= -=	복합 대입 연산자(덧셈, 뺄셈)	"
*= /=	복합 대입 연산자(곱셈, 나눗셈)	"

## 연산자 우선 순위

```
a > b && a > c                                ①
(a == 2 || b == 4) && (c == 5 || d == 6)      ②
a == 2 || b == 4 && c == 5 || d == 6          ③
```

- 연산자 우선 순위
  - 관계 연산이 논리 연산보다 높다.
  - ②, ③은 다르다.
  - 대입 연산이 가장 낮다.

$$\begin{array}{c} a * b / c \\ \hline 1 \\ \hline 2 \end{array} \qquad \begin{array}{c} a = b = c \\ \hline 1 \\ \hline 2 \end{array}$$

- 결합순서
  - 일반적으로 왼쪽에서 오른쪽으로
  - 대입 연산자, 복합 대입 연산자만 오른쪽에서 왼쪽으로

## 부수 효과(Side Effects)

```
int a = 1, b;  
double c;  
  
a = 10 + 20;           // a changes  
b = a++;              // a, b changes  
printf("%d\n", a + b); // output  
printf("%d\n", a++);   // output, a changes  
c = a;                // c changes
```

```
int a = 5; result = add(a, ++a);  
if (a < 0 && a++ == 10)  
int a = 1; result = a + a++;  
a + (++a) - (a++)
```

- 부수 효과가 모호
- 연산식 내부에, 값이 바뀌는 변수를 반복하는 일은 피해야 한다.

## 단축 회로 연산(Short Circuit Evaluation)

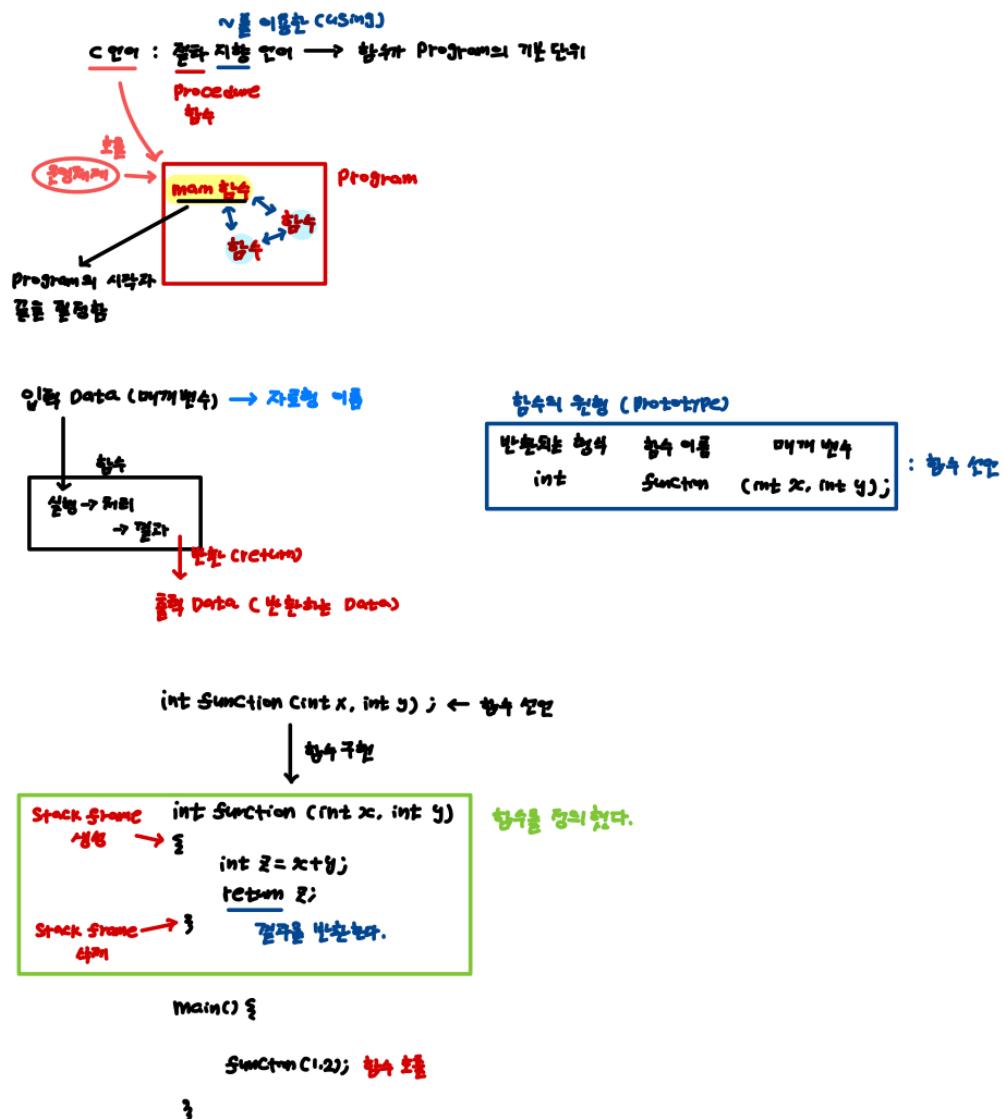
$$\frac{a > b \quad || \quad \frac{c > d \quad \&\& \quad e > f}{\begin{array}{c} 1 \\ 2 \\ \hline 3 \end{array}}}{\begin{array}{c} 4 \\ \hline 5 \end{array}}$$

- 1이 참이면 2, 3, 4는 평가되지 않는다.
  - OR 연산에서 하나라도 참이면 결과는 참

```
a > 1 || b++ > 0
```

- ( $a > 1$ )이 참이면  $b$ 가 증가할 것인가?
- 어느 것이 먼저 평가될 것인지는 컴파일러에 따라 다를 수 있다.
- 단축 회로 연산을 예상한 프로그램은 위험하다.

## 4장. 함수 1



### 4.1 함수 정의

```
#include <stdio.h>

int add (int f, int s) {
    int total;
    total = f + s;
    return total;
}

int main() {
    int first, second, sum;
    printf("Enter two input integers.\n");
```

```

    scanf("%d%d", &first, &second);
    sum = add(first, second);
    printf("The result is %d.\n", sum);

    return 0;
}

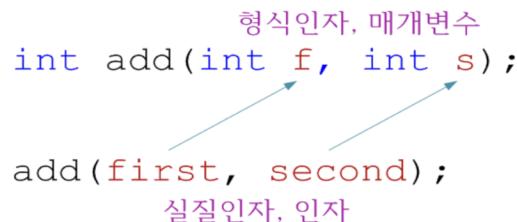
```

- function
  - one of a group of related actions contributing to a larger action

(Webster's New Collegiate Dictionary)

## 함수 호출 시 인자 전달

### ➊ Example 4-1 실습 및 해설



- Parameters (매개변수) 또는 Arguments (인자)
  - 호출함수의 매개변수: 실질 인자
  - 피호출 함수의 매개변수: 형식 인자
  - 매개변수는 함수 내에서 선언한 지역 변수와 동일하게 취급됨
  - 인자 값을 전달하는 것은 일종의 대입 연산
  - 함수 사이의 교신은 매개변수를 통해서 가능

### example\_4-1.c :

```

#include <stdio.h>

int add (int f, int s) {
    int total;
    total = f + s;
    return total;
}

int main() {
    int first, second, sum;
    printf("Enter two input integers.\n");

```

```

        scanf("%d%d", &first, &second);
        sum = add(first, second);
        printf("The result is %d.\n", sum);

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기/시스템프로그래밍기초/dev/ch4 (main*?) $ ./example_4-1
Enter two input integers.
2 3
The result is 5.

```

## 함수

- 특정 기능을 수행하는 명령문의 집합
  - 함수는 단 하나의 기능을 수행하도록 하는 것이 좋다.
  - 반복해서 호출할 수 있다.
  - 함수를 호출하면 CPU 사용권이 피 호출 함수로.
  - 피 호출 함수가 끝나면 CPU 사용권이 다시 호출 함수로.

```

add(2 + 3, 4 + 5);
add(first + 1, second + 1);
add(square(first), square(second));
printf("The result is %d.\n", add(first, second));

```

- 다양한 형식의 인자가 가능

### ● Example 4-2 실습 및 해설

- 함수 실행 중 또 다른 함수를 호출할 수 있다.

#### **example\_4-2.c :**

```

#include <stdio.h>

int square(int m) {
    return m * m;
}

int square_add(int f, int s) {
    int total;
    total = square(f) + square(s);
    return total;
}

```

```

int main() {
    int first, second, sum;
    printf("Enter two integers.\n");
    scanf("%d%d", &first, &second);
    sum = square_add(first, second);
    printf("The result is %d.\n", sum);

    return 0;
}

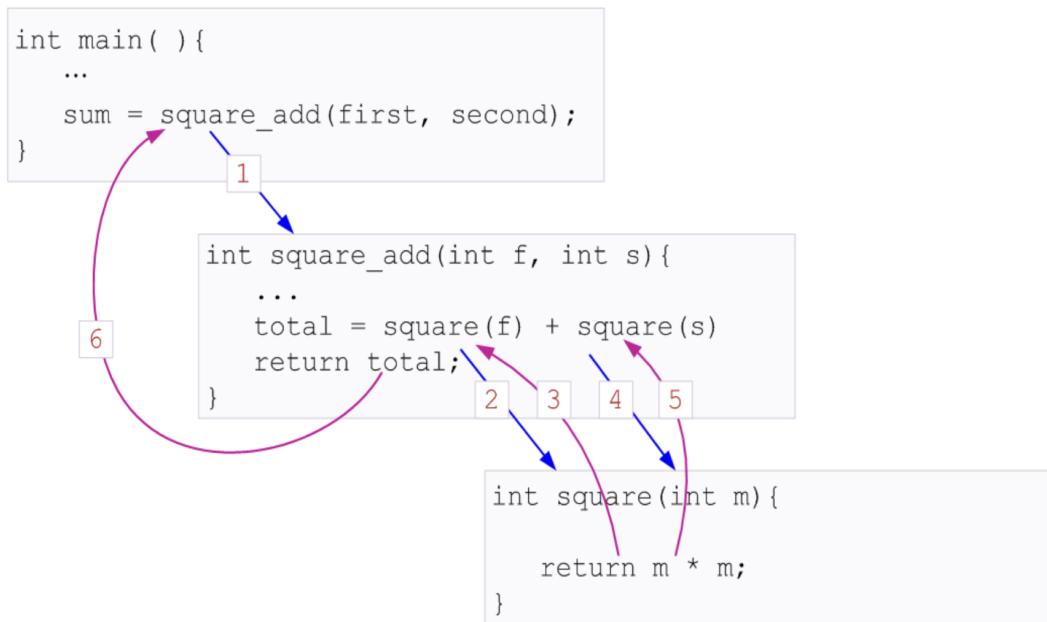
```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch4 (main*?) $ ./example_4-2
Enter two integers.
2 4
The result is 20.

```

## 함수 호출 및 리턴 순서



## One-Pass Compiler

- Visual C는 One-Pass Compiler
  - 소스코드를 한 번만 보고 컴파일을 끝냄
  - A가 B보다 먼저 나와야 B가 A를 호출할 수 있음
  - cf. Two-Pass Compiler

```
#include <stdio.h>
int square(int m);
int square_add(int, int);
```

- 함수 원형을 먼저 선언하면 순서가 문제되지 않는다.

```
int square(int m);    ① 선언
int square(int m) {    ② 정의
    return m*m;
}
square(num);          ③ 호출
• 함수의 선언, 정의, 호출은 다른 의미
```

## 사용자 정의 함수, 라이브러리 함수

### ➊ Example 4-3 실습 및 해설

- 사용자 정의 함수

```
double pow(double x, double y);
```

- math.h에 선언

### • 표준 라이브러리

- stdio.h, stdlib.h, ctype.h, string.h, math.h, time.h, ...
- 대략 145개 정도의 함수를 제공

### example\_4-3.c :

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

int square_add(int, int); // 선언
```

```

int main() {
    int first, second, sum;
    printf("Enter two input integers.\n");
    scanf("%d%d", &first, &second);
    sum = square_add(first, second); // 호출
    printf("The result is %d.\n", sum);

    return 0;
}

int square_add(int f, int s) { // 정의
    double total;
    total = pow((double)f, 2.0) + pow((double)s, 2.0);
    return (int)total;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch4 (main*?) $ ./example_4-3
Enter two input integers.
2 4
The result is 20.

```

## 리턴 값, 리턴 문

### Example 4-4 실습 및 해설

- 리턴 값이 없는 함수는 `void`로 선언

```

int count;
count = printf("Hello.\n");
printf("%d\n", count);
    • printf 문의 리턴 값

```

```

int absolute_value(int n) {
    if (n >= 0)
        return n;
    else
        retrun -n;
}

```

- 리턴 문의 역할: 리턴 값을 돌려준다. 호출 함수로 되돌아간다.
- 리턴 타입이 `void`이면 마지막 리턴 문 생략 가능

### example\_4-4.c :

```

#include <stdio.h>

void show_menu() {
    printf("Press 1 to continue. 2 to exit.\n");
}

```

```

        return;
    }

int main() {
    show_menu();

    return 0;
}

```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch4 (main*?) $ ./example_4-4
Press 1 to continue. 2 to exit.
```

```

int count;
count = printf("Hello.\n");
printf("%d\n", count);

```

printf 함수는 성공적으로 출력한 문자의 개수를 리턴한다. 따라서 예처럼 리턴 값을 count라는 변수에 저장하려 출력하면 7이 찍힌다. 구두점과 '\n'까지 포함해서다. 그러나 피호출 함수가 리턴 값을 돌려주더라도 호출 함수가 그것을 받지 않을 수도 있다. 단순히 printf("Hello.\n");라고만 할 때가 그러할 때다. 함수를 실행시키는데에만 관심이 있고 그 함수가 돌려주는 값에는 관심이 없을 경우에 리턴 값을 받지 않으면 그만이다.

리턴문의 2가지 역할 :

1. 리턴 값을 돌려줄 때도 있다.
2. 즉시 호출 함수로 되돌아간다.

리턴문은 두 가지 역할을 한다. 값을 돌려주는 역할도 하지만 즉시 호출 함수로 되돌아가라는 의미도 있다. 리턴문을 그림처럼 쓰면 그 아래 명령문을 무시하고 곧바로 호출 함수로 되돌아간다. 따라서 만약 show\_menu 함수 내부에 또 다른 명령문이 있으면 그 명령문은 실행되지 않는다. 이런 특성으로 인해 리턴문은 종종 함수 중간에 강제로 함수를 종료하고 호출 함수로 되돌아가게 할 때 사용된다.

리턴 타입이 void일 때는 함수의 마지막 명령문으로 return 문을 쓰지 않아도 된다. 예를 들어, 리턴 타입을 void로 선언할 경우 마지막 리턴문은 생략해도 좋다. 아무것도 돌려 줄 것이 없는 상태에서는 return 문을 쓰지 않아도 함수 끝을 나타내는 }를 만나면 어차피 함수가 종료되어 호출 함수로 되돌아가기 때문이다. 물론 이 경우에도 함수 중간에 강제로 함수를 종료하려면 return 문을 써야한다.

## 함수 단위의 분할 정복

### Example 4-5 실습 및 해설

- `int show_menu( ){ }`  
`int add_it( ){ }`  
`int subtract_it( ){ }`  
`void print_it( ){ }`
- `int main( ){ // 메인은 계속해서 함수만 호출하고 끝낸다.`  
 `int selection, result, a = 10, b = 20;`  
  
 `selection = show_menu(); // 메뉴를 보여주고`  
 `if (selection == 1) // 1을 선택했으면`  
 `result = add_it(a, b); // 덧셈 결과를 계산하고`  
 `if (selection == 2) // 2를 선택했으면`  
 `result = subtract_it(a, b); // 뺄셈 결과를 계산하여`  
 `print_it(result); // 결과를 출력`  
 `return 0;`  
`}`

**example\_4-5.c :**

```
#include <stdio.h>

int show_menu() {
    int choice;
    printf("Press 1 to add. 2 to subtract. 3 to exit.\n");
    scanf("%d", &choice);
    return choice;
}

int main() {
    int selection;
    selection = show_menu();
    printf("%d.\n", selection);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍기초 /dev/ch4 (main*?) $ ./example_4-5
Press 1 to add. 2 to subtract. 3 to exit.
1
```

## 4.2 함수명, 들여쓰기, 주석 처리

## 함수 명, 인자 명

- 함수 명에 의미를 부여
  - add, square, show\_menu
  - 너무 길 경우 입력 오류 우려
  - C 언어는 대소문자에 민감
- ```
int calc_area(int w, int h) {
    return (w * h);
}

int main() {
    int area, width, height;
    area = calc_area(width, height);
}
```

  - 호출 함수의 인자 명과 피 호출 함수의 인자 명은 서로 다르게
  - 호출 함수의 인자 명은 가독성이 중요
  - 피 호출 함수의 인자 명은 상대적으로 가독성이 낮아도 무방

## 주석(Comments)

```
index++;                                // adds 1 to index
printf("Math score is, %d.\n", math);      /* prints math score */
```

- 변수 명, 함수 명이 의미가 있으면 그것으로 충분
- 예는 불필요한 주석
- WHAT
  - 그 함수가 무엇 (WHAT) 을 하는지 역할이나 기능만 설명
  - 방법 (HOW) 은 이야기할 필요가 없음
- 코딩 도중에 주석을 다는 습관이 중요
  - 코딩부터 마치고 주석은 나중에 달겠다
  - 그러나 코딩이 끝나면 주석은 등한시하기 마련

## 프로그램 단위, 함수 단위, 블록 단위의 주석

```
/* 이 프로그램은 진법 변환을 예시하기 위한 것으로서,  
양의 10진수를 입력 받아 원하는 진법으로 변환하여 화면에 출력한다.  
음수가 입력되면 프로그램을 종료한다. */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
// 화면으로부터 원하는 진법을 입력 받는 함수  
int get_number_system( ) {  
    int num_sys;  
    printf("Enter the number system you want. \n");  
  
    // 입력 오류 및 종료조건 처리  
    if (scanf("%d", &num_sys) != 1) {  
        printf("Error while reading input. \n");  
        return EXIT_FAILURE;  
    }  
    else if (num_sys <= 0) {  
        printf("Terminating the program. \n");  
        return EXIT_FAILURE;  
    }  
  
    // 정상적 입력 처리  
    else  
        return num_sys;
```

## 5장. 선택구조

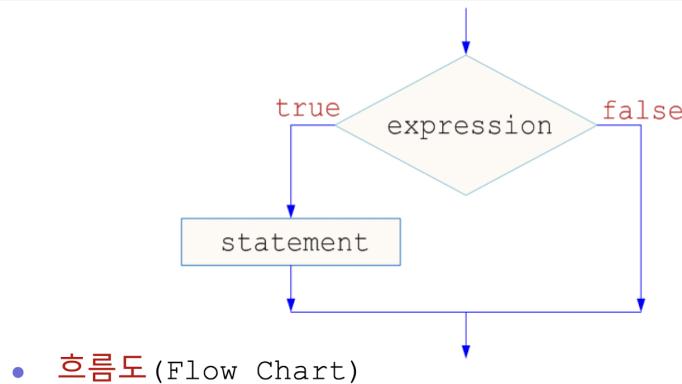
### 5.1 if문

## if statement

### Example 5-1 실습 및 해설

- 조건에 따른 분기(조건문 또는 분기문)

```
if (expression)
    statement;
next statement;
```



- 흐름도 (Flow Chart)

### example\_5-1.c :

```
#include <stdio.h>

void odd_even(int n) {
    if (n % 2 == 1)
        printf("Odd number.\n");
    else
        printf("Even number.\n");
    return;
}

int main() {
    int num;
    printf("Enter an integer.\n");
    scanf("%d", &num);
    odd_even(num);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch5 (main*)? $ ./example_5-1
Enter an integer.
2
Even number.
```

## 조건식

```
if (3 > a)
if (a < 3)           // Better
```

```
#include <math.h>
double fabs(double d);
if (fabs(d) > 2.0)
```

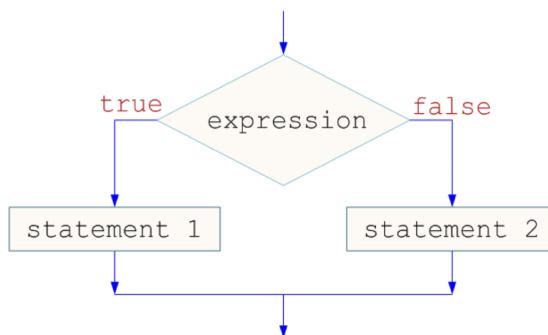
- 조건식 안에 함수 리턴 값이 올 수도 있음

```
double d1, d2;
if (d1 == d2)
if (fabs(d1 - d2) < 0.0001)
```

- 부동소수끼리는 오차 한계 이내인지를 비교

## if, else

```
if (expression)
    statement 1;
else
    statement 2;
next statement;
```



### Example 5-2 실습 및 해설

#### example\_5-2.c :

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

```

void compare(int n1, int n2) {
    if (n1 > n2)
        printf("First is larger.\n");
    else if (n1 < n2)
        printf("Second is larger.\n");
    else
        printf("They are equal.\n");
    return;
}

int main() {
    int first, second;
    printf("Enter two integers.\n");
    scanf("%d%d", &first, &second);
    compare(first, second);

    return 0;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-2
Enter two integers.
10 5
First is larger.

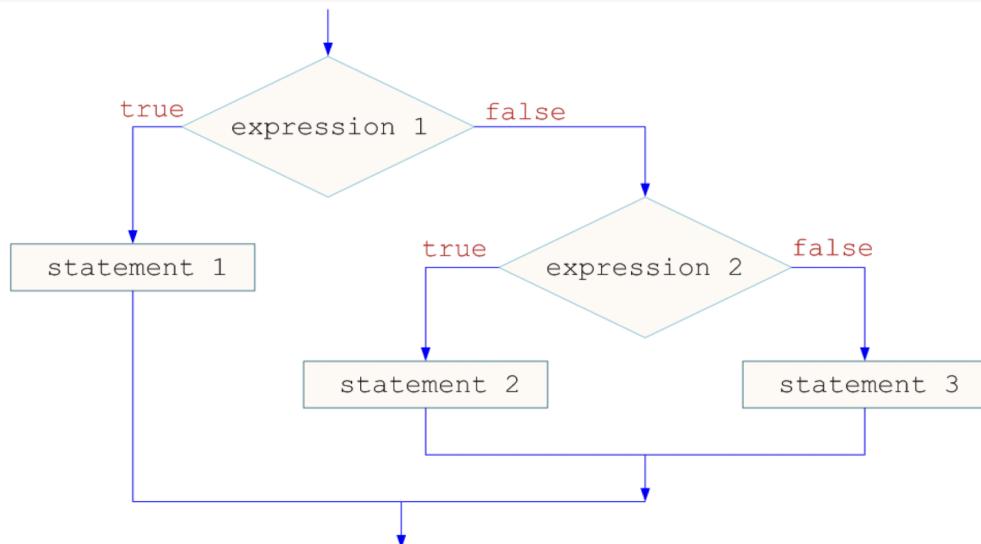
```

### if, else if, else

```

if (expression 1)
    statement 1;
else if (expression 2)
    statement 2;
else
    statement 3;
next statement;

```



## if, else if, else

- if, else if, else if는 하나의 문장이다.

```
int a = 20;
if (a > 5)
    printf("Larger than 5");
else if (a > 10)
    printf("Larger than 10");
return 0;
```

- 먼저 만나는 조건이 참이면 그 아래 조건은 테스트하지 않는다.

- Example 5-3 실습 및 해설
- Example 5-4 실습 및 해설

### example\_5-3.c :

```
#include <stdio.h>

int main () {
    int num;

    printf("Enter an integer.\n");
    scanf("%d", &num);

    if (num > 3)
        printf("num is larger than 3.\n");
    if (num > 5)
        printf("num is larger than 5.\n");

    if (num > 3)
        printf("num is larger than 3.\n");
    else if (num > 5)
        printf("num is larger than 5.\n");

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-3
Enter an integer.
10
num is larger than 3.
num is larger than 5.
num is larger than 3.
```

### example\_5-4.c :

```
#include <stdio.h>

int main () {
    int num;
    printf("Enter an integer.\n");
    scanf("%d", &num);

    if (num > 3)
        printf("num is larger than 3.\n");
    if (num <= 3)
        printf("num is less than or equal to 3.\n");

    if (num > 3)
        printf("num is larger than 3.\n");
    else
        printf("num is less than or equal to 3.\n");

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-4
Enter an integer.
20
num is larger than 3.
num is larger than 3.
```

## 블록

```
if (n % 2) {  
    printf("The number you entered is %d.\n", n);  
    printf("It is an odd number.\n");  
}
```

- 컴파일러는 블록 전체를 하나의 문장(**복문**)으로 취급

```
if (n % 2 == 1)  
    printf("The number you entered is %d.\n", n);  
    printf("It is an odd number.\n");  
• 잘못된 들여쓰기  
• 컴파일러는 이를 아래처럼 해석
```

```
if (n % 2 == 1)  
    printf("The number you entered is %d.\n", n);  
printf("It is an odd number.\n");
```

- 적용 범위가 불분명해 보이면 블록으로 처리하는 편이 낫다.

## else block

| ⓐ                                                                                   | ⓑ                                                                                                               |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <pre>if (a)<br/>    one;<br/>else if (b)<br/>    two;<br/>else<br/>    three;</pre> | <pre>if (a)<br/>    one;<br/>else {<br/>    if (b)<br/>        two;<br/>    else<br/>        three;<br/>}</pre> |

- 생각 방법의 차이일 뿐 동일한 표현

## 최소값을 구하는 세 가지 방법

### Example 5-5, 5-6, 5-7 실습 및 해설

- ```
if (a < b) {
    if (a < c)
        return a;
    else
        return c;
}
else {
    • if 안에 if가 있으면 중첩 if(Nested if)
    • 안쪽 if는 if (a < b && a < c) 와 동일
    • 안쪽 else는 if ((a < b) && !(a < c)) 와 동일
}
}

• 들여쓰기 예 유의
    • 안쪽 if, else가 같은 열, 바깥쪽 if, else가 같은 열
```

#### example\_5-5.c :

```
#include <stdio.h>

int get_min (int a, int b, int c) {
    if (a < b && a < c)
        return a;
    else if (b < a && b < c)
        return b;
    else
        return c;
}

int main () {
    int first, second, third;
    printf("Enter three integers.\n");
    scanf("%d%d%d", &first, &second, &third);
    printf("The minimum is %d.\n", get_min(first, second, third));

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-5
Enter three integers.
3 5 1
The minimum is 1.
```

#### example\_5-6.c :

```

#include <stdio.h>

int get_min (int a, int b, int c) {
    int min;
    if (a < b)
        min = a;
    else
        min = b;
    if (c < min)
        min = c;
    return min;
}

int main () {
    int first, second, third;
    printf("Enter three integers.\n");
    scanf("%d%d%d", &first, &second, &third);
    printf("The minimum is %d.\n", get_min(first, second, third));

    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-6
Enter three integers.
3 5 1
The minimum is 1.

```

### example\_5-7.c :

```

#include <stdio.h>

int get_min(int, int, int);

int main () {
    int first, second, third;
    printf("Enter three integers.\n");
    scanf("%d%d%d", &first, &second, &third);
    printf("The minimum is %d.\n", get_min(first, second, third));

    return 0;
}

int get_min(int a, int b, int c) {
    if (a < b) {
        if (a < c)
            return a;
        else
            return c;
    }
}

```

```

        else {
            if (b < c)
                return b;
            else
                return c;
        }
    }
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-7
Enter three integers.
3 5 1
The minimum is 1.

```

## 들여쓰기

### Example 5-8, 5-9 실습 및 해설: Dangling else

```

if (a)          // if (a == true)
    if (b)      // if (a == true && b == true)
        if (c)  // if (a == true && b == true && c == true)
            one;
        else     // if (a == true && b == true && c == false)
            two;
    else       // if (a == true && b == false)
        three;
else           // if (a == false)
    four;

```

- if 문에서 **들여쓰기는 필수**다.
- 어느 else가 어느 if에 걸리는지 일목요연

#### example\_5-8.c :

```

#include <stdio.h>

int main () {
    int a = 1, b = 0;

    if (a) // 1
        if (b) // 2
            printf("a and b are both true.\n");
    else // 3
        printf("a is not true.\n"); // 4

```

```
    return 0;  
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-8  
a is not true.
```

중첩 if문에서 자주 발생하는 오류가 이른바 땅글링 엘스 오류다. 이는 중첩 if문 다음에 //3처럼 else가 매달려 그것이 그 위의 어떤 if에 걸리는지 모호한 경우를 말한다. 들여쓴 모습으로 보아 이 예제를 작성한 프로그래머는 //3의 else가 //1의 if에 대한 else라고 생각했을 가능성이 크다. 따라서 a가 1(true)이므로 이 else로는 들어오지 않을 것이라 생각 했을 것이다. 그러나 else는 항상 그 직전의 if에 걸린다. 따라서 //3의 else는 //2의 i에 걸린다. 즉, 이 else는 a가 true인 동시에 b가 false일 때에 해당하므로 //4가 실행된다. 결국 //3은 잘못된 들여쓰기다. //2의 if와 같은 위치에 두어야 한다.

### example\_5-9.c :

```
#include <stdio.h>  
  
int main () {  
    int a = 1, b = 0;  
  
    if (a) {  
        if (b)  
            printf("a and b are both true.\n");  
    }  
    else  
        printf("a is not true.\n");  
  
    return 0;  
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-9  
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ █
```

중괄호를 쓰면 프로그래머의 의도가 분명해진다. //3의 else를 //1의 if에 대한 else로 만들려면 예처럼 //1의 if 가 참일 때 실행해야 할 부분을 중괄호로 묶어야 한다. 그래야 컴파일러가 //3의 else를 //1의 if에 대한 else로 취급한다.

## 조건식

```
int a = 1, b = 2, sum;  
if ((a + b) > 0)  
if ((sum = a + b) > 0)
```

- 조건식 내부에 연산식이나 대입문이 올 수 있다.

```
int a = 3;  
if (++a > 2)  
if (a++ > 2)  
if (a + 1 > 2)
```

- 조건식 자체에 **부수 효과**(Side Effect)가 있을 수도 있다.

### Example 5-10 실습 및 해설

- if (max = 4)
- 조건식 내부의 비교에 대입 연산자를 써서는 안 된다.

#### example\_5-10.c :

```
#include <stdio.h>  
  
int main () {  
    int max = 0;  
  
    if (max = 4)  
        printf("Yes.\n");  
    else  
        printf("No.\n");  
  
    return 0;  
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~ /Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*) $ gcc -o example_5-10 example_5-10.c  
example_5-10.c:6:13: warning: using the result of an assignment as a condition without parentheses [-Wparentheses]  
    if (max = 4)  
            ^~~~~~  
example_5-10.c:6:13: note: place parentheses around the assignment to silence this warning  
    if (max = 4)  
            ^  
            ()  
example_5-10.c:6:13: note: use '==' to turn this assignment into an equality comparison  
    if (max = 4)  
            ^  
            ==  
1 warning generated.
```

## 연산자 우선 순위

```
if ((a > 20) && (b == 10))  
if (a > 20 && b == 10)      // Better
```

- 관계 연산자가 논리 연산자보다 우선 순위가 높다.
- 괄호 수를 줄일 필요가 있을 때도 있다.
  - Ex. (! (a > 10)) && (! ((b == 10) || (c == 10))))

### Example 5-11 실습 및 해설

- 2차 방정식의 실근 구하기

#### example\_5-11.c :

정수 a, b, c를 입력받아  $ax^2 + bx + c = 0$ 의 근을 구하는 프로그램을 작성하였다. 단, 2차 방정식의 판별식  $D=b^2 - 4ac$ 의 값은 0 이상으로 가정하기로 한다.

a와 b 모두가 0으로 입력될 때도 있을 것이다. 그 경우에 예는 0차 방정식이라고 할 수 있다. a는 0이지만 b가 0이 아니면 1차 방정식이다. a가 0이 아니면 2차 방정식이다. 따라서 다음과 같은 의사 코드를 써서 알고리즘을 표현할 수 있다.

1. 정수 a, b, c를 입력받는다.
2. 몇 차 방정식인지 판단한다.
3. 2차 방정식이면 판별식 D 값을 계산하고 근의 공식에 대입하여 두 실근을 출력한다.
4. 1차 방정식이면 하나의 근을 계산하여 출력한다.
5. 0차 방정식이면 근이 없다고 출력한다.

```
#include <stdio.h>  
#include <math.h>  
  
void zero_order(int a, int b, int c) {  
    printf("There is no root.\n");  
}  
  
void first_order(int a, int b, int c) {  
    printf("The root is, %.3f.\n", (-c) / (double)b);  
}  
  
double calc_D(int a, int b, int c) {  
    return (double)(b * b - 4 * a * c);  
}
```

```

void second_order(int a, int b, int c) {
    double root1, root2, D, root_D;
    D = calc_D(a, b, c);
    root_D = sqrt(D);
    root1 = (-b + root_D) / (double)(2 * a);
    root2 = (-b - root_D) / (double)(2 * a);
    printf("The roots are, %.3f, %.3f.\n", root1, root2);
}

int main() {
    int a, b, c;
    printf("Enter the coefficients a, b, c.\n");
    scanf("%d%d%d", &a, &b, &c);
    if (a != 0)
        second_order(a, b, c);
    else if (b != 0)
        first_order(a, b, c);
    else
        zero_order(a, b, c);

    return 0;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*)$ ./example_5-11
Enter the coefficients a, b, c.
1 -4 3
The roots are, 3.000, 1.000.

```

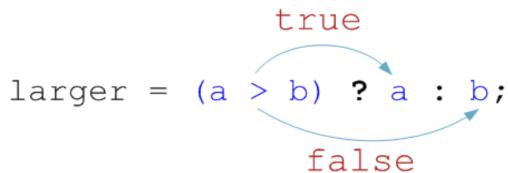
### 3항 연산자

```

if (a > b)
    larger = a;
else
    larger = b;

larger = (a > b) ? a : b;
a > b ? larger = a : (larger = b);

```



- 첫 항의 조건이 참이면 둘째 항을, 거짓이면 셋째 항을 실행
- 둘째 항, 셋째 항이 대입문이면 셋째 항을 괄호로 묶어야 함.
- 그렇지 않으면 대입문의 우선 순위가 낮아 컴파일러는 이를 `(a > b ? larger = a : larger) = b;`로 간주하여 오류 처리

## 5장. 선택구조

### 5-1. switch 문

#### switch statement

##### Example 5-12, 5-13 실습 및 해설

- ```
switch (ch) {  
    case 'A':  
        printf("Excellent.\n");  
        break;  
    case 'B':  
        printf("Good.\n");  
        break;  
    default:  
        printf("No such grade.\n");  
        break;  
}
```

- case 다음에는 단 하나의 정수 상수만 올 수 있다.
  - case (a > 20)는 불가. case 24.01도 불가
  - break 문을 만나면 곧바로 switch 문을 빠져 나간다.

#### example\_5-12.c :

```
#include <stdio.h>  
  
void comment(char);  
  
int main () {  
    char grade;  
    printf("Enter your grade in capital letter. __\b\b");  
    scanf("%c", &grade);  
    comment(grade);  
  
    return 0;  
}  
  
void comment (char ch) {  
    if (ch == 'A')  
        printf("Excellent.\n");  
    else if (ch == 'B')
```

```

        printf("Good.\n");
    else if (ch == 'C')
        printf("Not bad.\n");
    else if (ch == 'D')
        printf("Need effort.\n");
    else if (ch == 'F')
        printf("You can do better than this.\n");
    else
        printf("No such grade.\n");
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-12
Enter your grade in capital letter. A_
Excellent.

```

### example\_5-13.c :

```

#include <stdio.h>

void comment(char);

int main () {
    char grade;
    printf("Enter your grade in capital letter. __\b\b");
    scanf("%c", &grade);
    comment(grade);

    return 0;
}

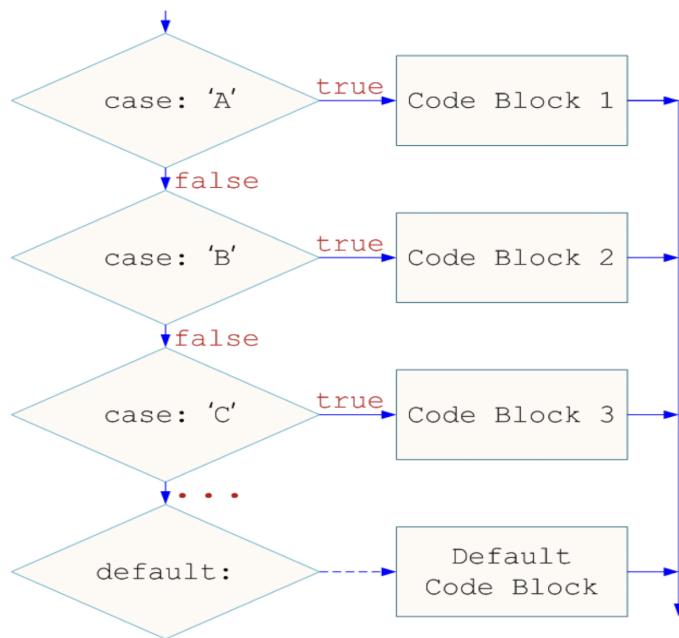
void comment (char ch) {
    switch (ch) {
        case 'A' :
            printf("Excellent.\n");
            break;
        case 'B' :
            printf("Good.\n");
            break;
        case 'C' :
            printf("Not bad.\n");
            break;
        case ('C' + 1) :
            printf("Need effort.\n");
            break;
        case 'F' :
            printf("You can do better than this.\n");
            break;
        default :
            printf("No such grade.\n");
    }
}

```

```
        break;  
    }  
    return;  
}
```

```
● (base) minsung@iminseong-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-13  
Enter your grade in capital letter. D_  
Need effort.
```

### Flow chart of switch statement



## switch statement

### Example 5-14 실습 및 해설

- ```
switch (ch) {
    case 'A':
        printf("Excellent.\n");
    case 'B':
        printf("Good.\n");
        break;
    case 'c':
    case 'C':
        printf("Not bad.\n");
        break;
```
- case가 일치하면 break 문을 만날 때까지 이후의 모든 명령문을 '무조건' 실행한다

### Example 5-15 실습 및 해설

**example\_5-14.c :**

```
#include <stdio.h>

void comment(char);

int main () {
    char grade;
    printf("Enter your grade.\n");
    scanf("%c", &grade);
    comment(grade);

    return 0;
}

void comment (char ch) {
    switch (ch) {
        case 'A' :
            printf("Excellent.\n");
        case 'B' :
            printf("Good.\n");
            break;
        case 'c' :
        case 'C' :
            printf("Not bad.\n");
```

```

        break;
    }
    return;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-14
Enter your grade.
A
Excellent.
Good.
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-14
Enter your grade.
C
Not bad.

```

### example\_5-15.c :

```

#include <stdio.h>

void calculate (double op1, char op, double op2) {
    switch (op) {
        case '+':
            printf("%.3f + %.3f = %.3f.\n", op1, op2, op1 + op2);
            break;
        case '-':
            printf("%.3f - %.3f = %.3f.\n", op1, op2, op1 - op2);
            break;
        case '*':
            printf("%.3f * %.3f = %.3f.\n", op1, op2, op1 * op2);
            break;
        case '/':
            printf("%.3f / %.3f = %.3f.\n", op1, op2, op1 / op2);
            break;
        default:
            printf("Not an allowable operator.\n");
            break;
    }
}

int main () {
    char operator;
    double operand1, operand2;
    printf("Enter an expression. For EXAMPLE, 2.20 + 3.30.\n");
    scanf("%lf %c %lf", &operand1, &operator, &operand2);
    calculate(operand1, operator, operand2);

    return 0;
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch5 (main*?) $ ./example_5-15
Enter an expression. For EXAMPLE, 2.20 + 3.30.
2.20 - 3.30
2.200 - 3.300 = -1.100.

```

## 6장. 반복구조

### 6-1. while 문과 for 문

#### WHY LOOP?

```
sum = 0;  
sum += 1;  
sum += 2;  
sum += 3;  
...  
sum += 10;
```

- 1부터 10까지의 합.
- 1부터 1000까지의 합은 어떻게?

#### Example 6-1 실습 및 해설

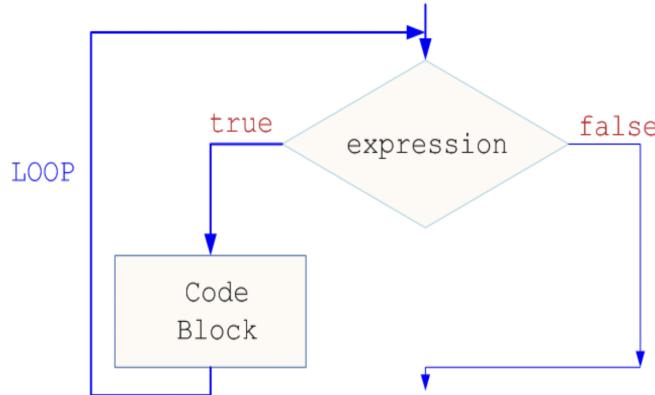
##### example\_6-1.c :

```
#include <stdio.h>  
  
int main () {  
    int sum = 0, count;  
    count = 1;  
    while (count <= 10) {  
        sum += count;  
        count++;  
    }  
    printf("Sum of 1 to 10 is %d.\n", sum);  
  
    return 0;  
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*)? $ ./example_6-1  
Sum of 1 to 10 is 55.
```

## while loop

```
count = 1;  
while (count <= 10){  
    sum += count;  
    count ++;  
}
```



- 루프 제어변수
- 무한 루프, 종료 조건

## 초기식, 조건식, 증감식

```
count = 1;          ① 초기식 (Initialization)  
while (count <= 10){ ② 조건식 (Condition Test)  
    sum += count;  
    count ++;          ③ 증감식 (Increment/Decrement)  
}
```

- 반복구조는 초기식, 조건식, 증감식으로 구성
  - 초기식은 루프를 들어가기 전에 실행된다.
  - 증감식에 의해 제어 변수의 값이 바뀐다.
  - 조건식에 의해 루프를 빠져 나간다.

## for statement

while	for
<pre>count = 1; while (count &lt;= 10) {     sum += count;     count++; }</pre>	<pre>for (count = 1; count &lt;= 10; count++)     sum += count;</pre>

- while 문과 for 문 비교

```
for (initialize; test; increment) {
    statement;
    statement;
    ...
}
```

- for 문의 초기식, 조건식, 증감식

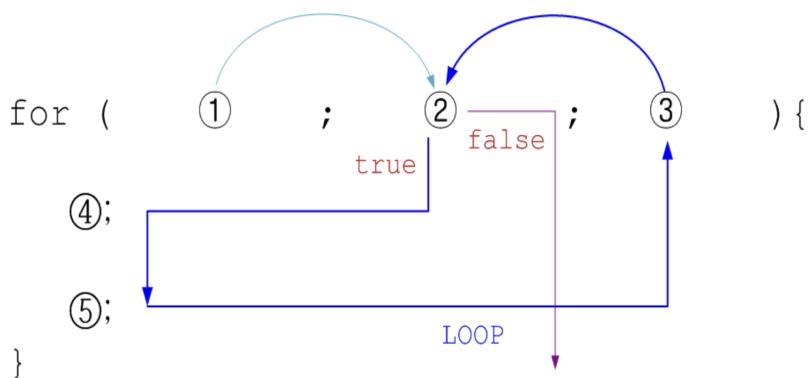
## for statement

```
for ( ; a < 10; )           // while 루프와 동일
for (a = 1; a < 10; )       // 증감식이 루프 내부에 있어야
for (a = 1;      ; a++)     // 조건식이 루프 내부에 있어야
```

```
for (a = 1; a < 10; a++) {
    if (a % 2 == 0)
        sum += a;
    a += 1;
}
```

- 초기식, 조건식, 증감식 중 어떤 것이라도 생략 가능
- 제어 변수가 증감식에도 있고 루프 내부에도 있으면 복잡

## flow of for statement



A: ① → ②(거짓) → 빠져나감.

B: ① → ②(참) → ④ → ⑤ → ③ → ②(거짓) → 빠져나감.

C: ① → ②(참) → ④ → ⑤ → ③ → ②(참) → ④ → ⑤ → ③ → ② ...

- 초기식은 단 한 번만 실행

## for statement, while statement

### Example 6-2 실습 및 해설

### Example 6-3 실습 및 해설

- for (sum = 0, count = 1; count <= 10; )
- 콤마 연산자는 나중 것을 평가한 결과를 리턴한다.

### Example 6-4 실습 및 해설

- Visual C는 for (int i = 1; i <= 10; i++)를 허용한다.
- 호환성을 고려하여 사용하지 않는 편이 낫다.

### Example 6-5 실습 및 해설

- 계속해서 입력을 받아 실행하기

### Example 6-6 실습 및 해설

- break 문을 만나면 루프를 빠져 나간다.

example\_6-2.c :

```
#include <stdio.h>

int main () {
    int sum = 0, count;
    for (count = 1; count <= 10; count++)
        sum += count;
    printf("After loop, count is %d.\n", count);
    printf("Sum of 1 to 10 is %d.\n", sum);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-2
After loop, count is 11.
Sum of 1 to 10 is 55.
```

### example\_6-3.c :

```
#include <stdio.h>

int main () {
    int sum = 0, count;
    for (count = 1; count <= 10; count++) // 1
        sum += count++; // 2
    printf("Sum of 1 to 10 is %d.\n", sum);

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-3
Sum of 1 to 10 is 55.
```

콤마 연산자는 나중 것을 평가한 결과를 리턴한다.

이전 예를 달리 표현한 것이다. //1의 초기식은 sum = 0, count = 1이라는 두 개의 명령문으로 되어 있다. 여기서 콤마 연산자는 2항 연산자로서 처음 피연산자와 나중 피연산자를 차례로 평가하되 나중 피연산자를 평가한 결과만 돌려준다. 예를 들어 int a = (2, 4);라고 하면 a에는 4가 대입된다. 또, result = (sum = 0, count = 1);이라고 하면 result에 1이 대입된다. sum = 0을 실행한 결과는 버려지지만 count = 1을 실행한 결과인 1이 리턴되어 result로 들어가기 때문이다.

그러나 예의 콤마 연산자는 단순히 sum과 count를 한꺼번에 초기화하기 위한 목적으로 사용되고 있다. 즉, sum = 0의 실행 결과인 0은 버려지지만 실행 도중 이미 sum에 0을 대입했다. 이는 count에 대해서도 마찬가지다. 콤마 연산자의 실행 결과를 다른 변수에 대입하지는 않았지만 실행 도중에 이미 count에 1이 들어갔다.

//1의 괄호 안에는 증감식도 비어 있다. 이렇게 할 수 있는 이유는 블록 내부의 //2에서 count++에 의해 증감을 가하고 있기 때문이다. 우변의 count++가 후위 증가하는 점에도 유의해야 한다. 이는 일단 현재의 count 값을 sum에 더한 다음에 증가하라는 뜻이다. 이를 만약 ++count로 하면 결과가 제대로 나오지 않는다.

사실상 이러한 형태의 코드는 그리 바람직하지 않다. 증감식을 비워 놓음으로 인해 제어 변수의 변화를 일목요연하게 추적하기 어렵기 때문이다. //1에서 sum = 0은 루프로 들어가기 전에 초기화하는 것이 낫다. 또 //2대신 sum += count;라고 쓰고 count++는 //1의 증감식 위치에 넣는 것이 가독성 면에서도 낫다.

### example\_6-4.c :

```
#include <stdio.h>

int main () {
    int sum = 0;
    for (int i = 0; i <= 10; i++)
        sum += i;
    printf("%d\n", sum);

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-4
55
```

### example\_6-5.c :

```
#include <stdio.h>

int main () {
    int num;

    printf("Enter a number to be converted.\n");
    scanf("%d", &num);
    while (num >= 0) {
        printf("%d is %X in hexadecimal.\n", num, num);
        printf("Enter a number to be converted.\n");
        scanf("%d", &num);
    }

    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-5
Enter a number to be converted.
10
10 is A in hexadecimal.
Enter a number to be converted.
16
16 is 10 in hexadecimal.
Enter a number to be converted.
30
30 is 1E in hexadecimal.
Enter a number to be converted.
-1
```

입력 10진수를 16진수 형태로 찍기 위한 프로그램으로서 입출력을 반복하기 위해 while 문을 쓰고 있다. 이 프로그램은 음의 정수를 입력하면 조건식에 의해 실행을 종료한다. 왜냐하면 printf 함수가 음수에 대해서는 16진 변환을 못하기 때문이다. 이처럼 계속해서 입력을 받아 루프를 돌릴 때는 조금 번거로운 점이 있다. 루프 밖에서 첫 입력을 받기 위한 명령문을 루프 내부에도 그대로 반복해야 하기 때문이다. 이렇게 된 이유는 첫 입력을 받지 않고 while 문의 조건식을 테스트할 수 없기 때문이다. 또, 루프 내부에도 반복해야 하는 이유는 루프를 돌 때마다 새로운 입력을 받아야 하기 때문이다.

### example\_6-6.c :

```
#include <stdio.h>

int main() {
    int num;
    while (1) {
        printf("Enter a number to converted.\n");
        scanf("%d", &num);
        if (!(num >= 0))
            break;
        printf("%d is %X in hexadecimal.\n", num, num);
    }

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-6
Enter a number to converted.
30
30 is 1E in hexadecimal.
Enter a number to converted.
16
16 is 10 in hexadecimal.
Enter a number to converted.
-1
```

example\_6-5.c 같은 번거로움을 피하려면 이처럼 짤 수도 있다.

### do while statement

while	do while
<pre>while (expression) {     statement;     statement; }</pre>	<pre>do {     statement;     statement; } while (expression)</pre>

- while 문은 들어가기 전에 조건을 테스트
- do while 문은 일단 한 번 실행한 다음 조건을 테스트
- 서로 다른 의미

### Example 6-7 실습 및 해설

#### example\_6-7.c :

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

```

int main () {
    int num;

    do {
        printf("Enter a number to be converted.\n");
        scanf("%d", &num);
        printf("%d is %X in hexadecimal.\n", num, num);
    } while (num >= 0);
    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-7
Enter a number to be converted.
16
16 is 10 in hexadecimal.
Enter a number to be converted.
30
30 is 1E in hexadecimal.
Enter a number to be converted.
-1
-1 is FFFFFFFF in hexadecimal.

```

## 6-2. 반복 구조 예시

### while, for

#### Example 6-8 실습 및 해설

while loop	for loop
<p>ⓐ</p> <pre> while (y &gt; 0){     rate *= (1.0 + i);     y--; }</pre>	<p>ⓑ</p> <pre> for (; y &gt; 0; y--)     rate *= (1.0 + i);</pre>
<p>ⓒ</p> <pre> int count = 0; while (count &lt; y){     rate *= (1.0 + i);     count++; }</pre>	<p>ⓓ</p> <pre> int count; for (count = 0; count &lt; y; count++)     rate *= (1.0 + i);</pre>

#### example\_6-8.c :

```

#include <stdio.h>

int total(int, double, int);

int main () {
    int principal; // 원금

```

```

double interest; // 연리
int years; // 햇수
printf("Enter principal, annual interest, and years.\n");
scanf("%d%lf%d", &principal, &interest, &years);
printf("Total is %d.\n", total(principal, interest, years));

return 0;
}

int total (int p, double i, int y) {
    double rate = 1.0;
    while (y > 0) {
        rate *= (1.0 + i);
        y--;
    }
    return (int)(p * rate);
}

```

```

(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기/시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-8
Enter principal, annual interest, and years.
200 0.15 2
Total is 264.

```

### EOF, getchar, break, Ctrl-Z

```
#define EOF -1
```

- 1) End of File를 만나거나
- 2) 입출력 실행 중 오류가 일어날 때의 리턴 값

```

int getchar( );
int putchar(int ch);
```

- 문자 하나를 입력하거나 출력하는 함수

### Example 6-9 실습 및 해설

- EOF를 입력 받으려면 정수형 변수를 선언
- break 문은 switch 문이나 루프를 빠져나갈 때 쓰인다.
- 키보드 입력에서 파일 끝을 표시하려면 Ctrl-Z (cf. Ctrl-D in Unix)

```
#define EOF -1
```

stdio.h는 EOF를 예처럼 정의하고 있다. 여기서 EOF(End of File)라는 상수는 두 가지 목적으로 쓰인다. 첫째, 입력이 정상적일 경우 함수 실행 결과 파일 끝에 도달했음을 알리기 위한 리턴 값으로 쓰인다. 둘째, 입력 함수를 실행하는 도중에 오류가 발생할 경우 그것을 알리기 위한 리턴 값으로도 쓰인다. 사실상 컴퓨터 하드웨어나 소프트웨어 환경에 문제가 있지 않은 한, 출력력 함수를 실행하는 도중 오류가 발생할 일은 거의 없다. 또, 그로 인해 오류가 발생할 경우에 대처할 방법도 없다. 그런 점에서 EOF는 주로 파일의 끝에 도달했는지 확인하기 위한 목적으로 쓰인다.

```
int getchar();
int putchar(int ch);
```

getchar(get character)도 scanf와 마찬가지로 stdio.h에 정의된 표준 입력 함수다. 이 함수는 단순히 문자 하나를 입력받아 해당 아스키코드 값을 돌려준다. 물론 getchar 대신 scanf("%d", &ch);라고 해도 같은 효과를 초래한다. 그러나 getchar는 scanf처럼 다양한 입력 형식을 지원하지는 않는다. getchar가 입력 함수라면 putchar(put character)는 출력 함수다. 이 역시 stdio.h에 정의된 함수로서 문자 하나를 표준 출력 장치인 화면에 찍는 역할을 한다. 또, 출력 중 오류가 일어날 경우 EOF를 돌려준다.

문자 하나를 읽을 때는 getchar 함수가 효율적이다.

단순히 문자 하나를 읽을 때는 scanf 함수보다 getchar 함수가 효율적이다. getchar 함수에 비해 scanf 함수는 복잡한 입력 형식을 처리해야 하기 때문에 코드가 길어질 뿐 아니라 형식 지정자를 처리하는 과정에 시간이 소요되기 때문이다. 같은 맥락에서 단순히 문자 하나를 출력할 때는 printf 함수보다 putchar 함수가 효율적이다. 파일 끝을 나타내는 EOF가 돌아올 때까지 입력받은 문자를 화면에 메아리치는(Echo) 프로그램을 작성해 보자.

#### example\_6-9.c :

```
#include <stdio.h>

int main () {
    int ch; // EOF를 받아들이려면 정수형으로 선언해야함
    for ( ; ; ) { // 무한 루프 형태
        ch = getchar();
        if (ch == EOF)
            break;
        else
            putchar(ch);
    }
    return 0;
}
```

```
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-9
a
a
b
b
c
c
d
d
```

## 정수 승격

### Example 6-10 실습 및 해설

- 정수 승격 (Integer Promotion)
  - char, short
  - 정수형보다 작은 자료형은 정수형으로 바뀌어 연산이 가해진다.

example\_6-10.c :

```
#include <stdio.h>

int main () {
    int ch; // 1
    printf("Enter a character.\n");
    scanf("%c", &ch); // 2
    if (ch == 'y') // 3
        printf("yes.\n");
    else
        printf("no.\n");

    return 0;
}
```

```
(base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-10
Enter a character.
y
no.
```

scanf 함수를 써서 정수 변수에 문자를 넣을 때 유의할 점이 있다. 예의 프로그램은 제대로 작동하지 않는다. 'y'를 입력해도 'no'가 프린트된다. 이는 입력한 'y'가 정수형 변수 ch에 제대로 들어가지 않았기 때문이다. C언어는 문자를 1바이트 정수로 취급한다. 그러나 //1에 의해 정수형 변수 ch에는 4바이트가 할당된다. 그런데 이 변수는 초기화를 하지 않았으므로 //2에서 읽은 'y'의 아스키코드 값인 10진수 121을 하위 1비트에 넣더라도 상위 3바이트에는 아직 초기화되지 않은 쓰레기 값이 그대로 남아있다.

정수형보다 작은 자료형은 정수형으로 바뀐 다음에 연산이 가해진다.

char, short 타입의 자료형은 일단 정수형으로 바뀐 다음에 연산이 가해진다. 이를 정수 승격이라 부른다. //3의 조건식에 있는 문자형 상수 'y'도 4바이트 정수로 승격한 다음에 비교 연산이 가해진다. 그 과정에서 'y'는 상위 3바이트를 모두 0으로 채워 넣은 정수로 바뀐다. 예는 그것이 쓰레기 값이 남아 있는 ch와 같지 않기 때문에 일어난 오류다. 이 문제를 해결하려면 //1을 int ch = 0;이라고 해서 0으로 초기화시켜야 한다. 그래서 4바이트 모두 0으로 초기화된 상태에서 하위 1바이트에 'y'를 넣기 때문에 정수 승격이 가해진 'y'와 같아진다.

getchar 함수를 쓰면 초기화를 하지 않고도 동일한 효과를 기할 수 있다. 이 함수는 디폴트로 4바이트 정수형 값을 리턴하기 때문이다. 다시 말해 //2 대신 ch = getchar();라고 해서 문자를 읽으면 자동으로 ch의 상위 3바이트가 0으로 채워진다. getchar 함수를 쓰면 //2를 char ch;라고 선언해도 좋다. 그 경우 getchar가 리턴한 4바이트 정수와 1바이트 정수 ch를 비교하는 과정에서 ch를 4바이트 정수로 바꾸는 자동 형 변환이 일어난

다. 또, 그 결과 ch의 상위 3바이트가 0으로 채워진다. 정수 승격이든 자동 형 변환이든 모두가 임시로 형을 바꾸어 연산에 활용하는 것일 뿐이고 변수 선언 당시의 원래 자료형은 그대로 유지된다.

## 입력 문자의 화면 에코

a	ch = getchar(); while (ch != EOF) { putchar(ch); ch = getchar(); }
b	while ((ch = getchar()) != EOF) putchar(ch);
c	for (ch = getchar(); ch != EOF; ch = getchar()) putchar(ch);

- 대입 연산자의 우선 순위가 가장 낮다.
  - while (ch = getchar() != EOF) 라고 하면 오류

## 입력 소문자를 대문자로 출력

- Example 6-11 실습 및 해설
  - 직접 변환
- Example 6-12 실습 및 해설
  - ctype.h (character type) 라이브러리 이용

함수 원형	의미	리턴 값
int isalpha(int)	알파벳인지 확인	1(true), 0(false)
int isdigit(int)	숫자인지 확인	1(true), 0(false)
int islower(int)	소문자인지 확인	1(true), 0(false)
int isupper(int)	대문자인지 확인	1(true), 0(false)
int toupper(int)	대문자로 변환	대문자의 아스키 값
int tolower(int)	소문자로 변환	소문자의 아스키 값

### example\_6-11.c :

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

```

int main () {
    int ch;

    printf("Enter a character.\n");
    for (ch = getchar(); ch != EOF; ch = getchar()) {
        if (ch >= 'a' && ch <= 'z') // ch가 영문 소문자인지를 확인
            ch -= 'a' - 'A'; // 소문자라면 아스키코드 값이 32만큼 줄어 대문자가
        됨. 가독성을 위해 ch -= 32가 아닌 ch -= 'a' - 'A'라 구현한 것
        putchar(ch);
        putchar('\n');
        getchar(); // 한 줄 입력이 끝날 때마다 버퍼에 남아 있는 엔터 키를 먹어치우
    기 위한 것
    printf("Enter a character.\n");
}
return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-11
Enter a character.
c
C
Enter a character.
a
A
Enter a character.
b
B
Enter a character.
d
D
Enter a character.

```

### example\_6-12.c :

```

#include <stdio.h>
#include <ctype.h>

int main () {
    int ch;
    printf("Enter a character.\n");
    for (ch = getchar(); ch != EOF; ch = getchar()) {
        if (islower(ch))
            ch = toupper(ch);
        putchar(ch);
        putchar('\n');
        getchar();
        printf("Enter a character.\n");
    }
    return 0;
}

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*) $ ./example_6-12
Enter a character.
a
A
Enter a character.
c
C
Enter a character.
d
D
Enter a character.

```

## Nested for

### Example 6-13 실습 및 해설

- `for (i = 1; i <= 6; i++) {  
 for (j = 1; j <= 6; j++) {  
 for (k = 1; k <= 6; k++) {`
- 중첩 for 문(Nested for)
  - 가장 안쪽 루프가 끝나야 직전 루프로 되돌아감.
- Pencil Tracing
  - (1, 1, 1), (1, 1, 2), ..., (1, 1, 6),
  - (1, 2, 1), (1, 2, 2), ..., (1, 2, 6),
  - (1, 3, 1), ..., (1, 6, 6),
  - (2, 1, 1), ..., (6, 6, 6)

#### example\_6-13.c :

```

#include <stdio.h>

int main () {
    int num, i, j, k;
    printf("Enter an integer between 2 and 19.\n");
    scanf("%d", &num);
    for (i = 1; i <= 6; i++) {
        for (j = 1; j <= 6; j++) {
            for (k = 1; k <= 6; k++) {
                if (i + j + k == num)
                    printf("Sum of %d, %d, %d is %d.\n", i, j, k, num);
            }
        }
    }
    return 0;
}

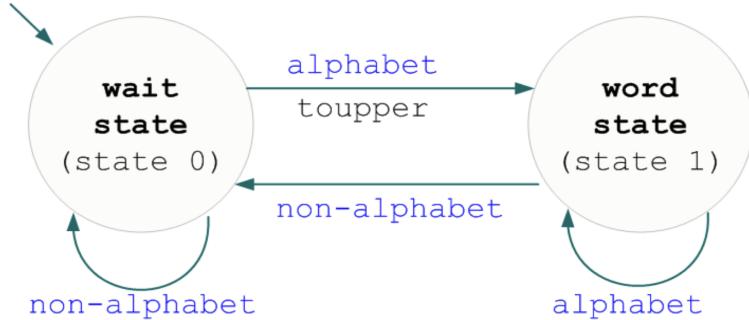
```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch6 (main*?) $ ./example_6-13
Enter an integer between 2 and 19.
5
Sum of 1, 1, 3 is 5.
Sum of 1, 2, 2 is 5.
Sum of 1, 3, 1 is 5.
Sum of 2, 1, 2 is 5.
Sum of 2, 2, 1 is 5.
Sum of 3, 1, 1 is 5.

```

### 상태 전이도(State Transition Diagram)



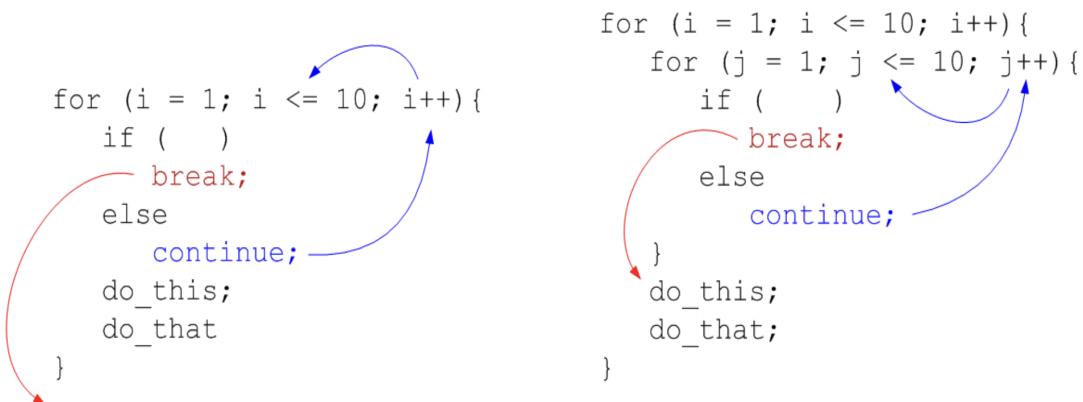
- 문장 내 모든 단어의 첫 글자를 대문자로 바꾸는 문제
- 컴퓨터는 거대한 상태 전이 기계 (State Transition Machine)
- 유한 상태 기계 (FSM: Finite State Machine)
- 상태 전이도 (State Transition Diagram)

### Example 6-14 실습 및 해설

#### break, continue

- Example 6-15 실습 및 해설
- Example 6-16 실습 및 해설
- Example 6-17 실습 및 해설

- break는 if 문이 아니라 루프를 빠져나가기 위한 것이다.**
- 중첩 루프의 break나 continue는 직전의 루프에 적용된다.



## goto statement

```
#include <stdio.h>
int main() {
    int sum = 0, count = 1;
LOOP:
    sum += count;
    count++;
    if (count <= 10)
        goto LOOP;
    printf("Sum is %d.\n", sum);
    return 0;
}
```

- 어셈블리 언어의 **무조건 분기** (Unconditional Branch)에 해당
- 사용하지 않는 편이 낫다.
  - 가독성 저하
  - 스파게티 코드가 될 우려

교수님이 작성해주신 코드 :

```
// EX 6
//int main() {
//    int sum = 0;
//
//    for ( int i = 1; i < 11; i++)
//    {
//        if (i % 2 == 0) {
//            sum = sum + i;
//            printf("sum : %d, sum from 0 to %d. \n", sum, i);
//        }
//    }
//    //printf("%d. \n", i);
//
//    return 0;
//}

// EX 약수 찾기
//int main() {
//    int num = 10, count = 0, sum = 0;
//
//    for (int i = 1; i < (num + 1); i++)
//    {
//        if (num % i == 0) {
```

```

//           count++;
//           sum = sum + i;
//           printf("%d는 %d번째 약수. \n", i, count);
//       }
//   }
//
//   printf("약수의 개수 : %d. \n", count);
//   printf("총 약수의 합 : %d. \n", sum);
//
//   if (count == 2) {
//       printf("%d는 소수입니다. \n", num);
//   }
//   else {
//       printf("%d는 합성수입니다. \n", num);
//   }
//   return 0;
//}

// EX 6-
//int main() {
//    int sum = 0;
//    for (int i = 1; i < 4; i++)
//    {
//        printf("%d. \n", i);
//        break;
//    }
//    //
//    //
//    for (int i = 1; i <= 10; i++)
//    {
//        if (i % 2 == 1)
//        {
//            continue;
//        }
//        sum += i;
//    }
//    //
//    printf("%d. \n", sum);
//    return 0;
//}

// EX 6-
//int main() {
//    int i = 1; // 제어변수 초기값
//
//    for ( ; ; )
//    {
//        if (i > 4) // 제어변수 조건

```

```

//      {
//          break;
//      }
//      printf("%d. \n", i);
//      i++; // 증감식
//  }
//
//  return 0;
//}

// EX 6-
//int main() {
//    int i = 1; // 초기식
//    while (i < 4) { // 조건식
//        printf("%d. \n", i);
//        i++; // 증감식
//    }
//    printf("----- \n");
//
//    i = 0;
//    do {
//        i++;
//        printf("%d. \n", i);
//    } while (i<4);
//
//    return 0;
//}

// EX 6-
//int main() {
//
//    int i = 1;
//    while (1) {
//        if (i > 3) {
//            break;
//        }
//        printf("%d. \n", i);
//        i++;
//    }
//    return 0;
//}

// EX 6-9
//int main() {
//    int ch;
//    for (;;) {
//        ch = getchar();
//        // ctrl + c => EOF

```

```

//      if (ch == EOF) {
//          break;
//      } else {
//          putchar(ch);
//      }
//  }
//  return 0;
//}

// EX 6-10
int main() {
    int ch;

    printf("Enter a character. \n");
    //scanf("%c", &ch);
    ch = getchar();

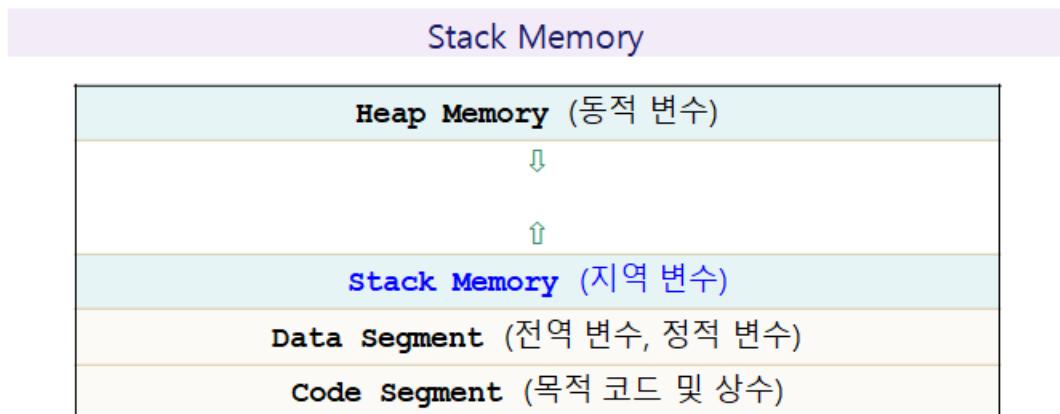
    if (ch == 'y') {
        printf("Yes. \n");
    }
    else {
        printf("No. \n");
    }
    return 0;
}

```

▼ 10/31 목 - 8장, 9장

## 8장. 함수 2

### 8-1. 스택 프레임



- 프로그램 실행 중의 메모리 구조
- 스택이 위로 자라고 힙이 아래로 자란다. (또는 그 반대)

## ADT Stack



- 추상 자료형 (ADT: Abstract Data Type)
  - 작업 (Operation)을 기준으로 추상적으로 정의한 자료형
- ADT Stack
  - 쌓아놓은 더미
  - Operation: Push, Pop

## 스택

- 어떤 식당에는 접시 스택 아래 스프링이 달려 있다. 새로운 접시를 위에 올리면 무게로 인해 스프링이 수축하여 그 아래 접시들이 카운터 아래로 내려간다. 즉, 가장 위에 놓인 접시만 카운터 높이 위에 위치한다. 가장 위 접시를 치우면 스프링이 팽창하여 바로 밑에 있던 접시 한 개가 카운터 위로 올라온다. 따라서 항상 접근할 수 있는 유일한 접시는 가장 위에 있는 접시이고 나머지 접시는 카운터 아래에 있으므로 접근이 불가능하다.
- 어머님이 월요일 아침에 내 책상 위에 메모를 올려놓으셨다. 화요일에 또 다른 메모를 그 위에 올려놓으셨다. 수요일 또 다른 메모를 그 위에 올려놓으셨다. 목요일, 또 다른 메모를 그 위에 올려놓으셨다. 금요일 날 어머님이 오셔서 '내가 월요일 날 올려놓은 메시지를 보았느냐'라고 하신다. 내 책상 위에는 넉 장으로 이루어진 메모지 스택이 있다. 월요일 메모는 가장 바닥에 깔려있다. 그것을 보기 위해 나는 목요일 메모를 내려놓는다. 다시 수요일 것을 내려놓는다. 다시 화요일 것을 내려놓는다. 마지막으로 월요일 것을 집어서 본다.
- 스택의 모든 작업은 **스택 탑 (stack top)**에서만 이루어진다. 그 아래 깔린 것은 접근할 수 없다. (Ex. Push, Pop)

## 스택 프레임

<b>main( )의 스택 프레임 (활성 상태)</b>	매개 변수		
	지역 변수	first second sum	
	리턴 주소		2040
	리턴 값		
	코드 세그먼트	1004 1008 1016	<pre style="font-family: monospace; margin: 0;">int main() {     int first, second, sum;     printf("Enter two input integers.\n");     scanf("%d%d", &amp;first, &amp;second);     sum = add(first, second);     printf("The sum is %d.\n", sum);     return 0; } int add(int f, int s) {     int total;     total = f + s;     return total; }</pre>

## 스택 프레임 (Stack Frame)

- 함수 호출
  - 함수를 호출할 때마다 스택 메모리에 **스택 프레임**을 푸쉬
  - **활성화 레코드 (Activation Record)**라고도 부름
  - 운영체제가 main 함수를 호출할 때 만들어진 스택 프레임
  
- 스택 프레임
  - **매개 변수 (Parameter)**
    - 호출 함수가 전달한 인자 값
  - **지역 변수 (Local Variables)**
    - 함수 내에서 선언한 변수
  - **리턴 주소 (Return Address)**
    - 함수 실행을 마친 다음 실행할 명령문의 주소
  - **리턴 값 (Return Value)**
    - 호출 함수에게 돌려줄 값

## 스택 프레임

add( )의 스택 프레임 (활성 상태)	매개변수	f, s
	지역 변수	total
	리턴 주소	1008
	리턴 값	
main( )의 스택 프레임 (비활성 상태)	매개변수	none
	지역 변수	first, second, sum
	리턴 주소	2040
	리턴 값	

- 운영체제가 main을 호출
  - main의 스택 프레임을 푸쉬
- main이 add를 호출
  - main의 스택 프레임 위에 add의 스택 프레임을 푸쉬
  - 리턴 주소 1008은 add 실행을 끝내고 되돌아갈 main의 주소

## 스택 프레임과 변수의 유효 영역

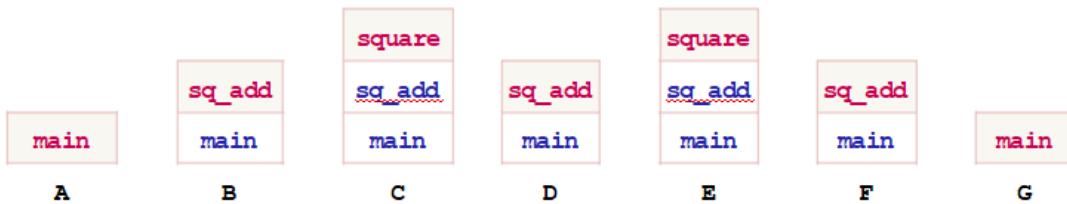
- 매 순간, 스택 탑에 있는 하나의 스택 프레임만 활성화된다.
  - add를 실행하는 동안에는 add의 스택 프레임 안에 있는 f, s, total에만 접근 가능
  - 그 아래 깔려 있는 main의 스택 프레임 안에 있는 first, second, sum에는 접근 불가능
  - 매개변수나 지역 변수의 유효 영역 (Scope of Variables)은 이런 방법에 의해 제한된다.
- 스택 프레임의 크기는 컴파일 타임에 결정된다.
  - 매개변수, 지역 변수의 개수와 타입을 고려

## 스택 프레임과 컨텍스트 스위칭

- 맥락의 전환 (Context Switching)
  - 전환 이전에 반드시 이전 상태를 기록
  - 피호출 함수를 호출하기 이전에 현재 함수와 관련된 정보를 기록
  - 피호출 함수로부터 되돌아온 다음, 이전 상태를 복원
- 스택 구조에 의해 맥락의 전환을 쉽게 구현할 수 있다.
  - add 실행이 끝나면 add의 스택 프레임을 팝
  - 이제 그 아래 깔려 있는 main의 스택 프레임이 스택 탑에 위치
  - main의 스택 프레임이 다시 활성화

## 스택 메모리

### ● Example 8-1 실습 및 해설



- 스택 메모리의 변화 모습
- 스택 오버플로우 (Stack Overflow)
  - 스택 프레임이 계속 쌓여 힙 메모리와 맞닿은 상태
  - 더 이상 가용 메모리가 없는 상태
- 지역 변수
  - 처음 쓰래기 값 (Garbage)
  - = 이전에 팝된 스택 프레임에 있던 값
- 지역 변수는 스택 프레임 안에 존재한다.
  - 지역 변수는 함수를 호출할 때 생성되어 함수에서 빠져나올 때 소멸된다. 이는 지역 변수가 스택 프레임 내부에 존재하기 때문이다. 예를 들어 sq\_add 함수의 지역 변수인 total은 sq\_add가 호출되는 순간 스택 프레임 내부에 만들어진다. 그러나 이 변수는 sq\_add를 빠져나가는 순간 사라진다. 팝에 의해 sq\_add의 스택 프레임 자체가 사라졌기 때문이다.
  - 지역 변수를 선언할 때는 그 안에 쓰래기 값이 들어가 있다는 것도 이런 측면에서 이해해야 한다. 늘었다 줄었다하면서 스택 공간을 재사용하기 때문에 새로 만들어진 스택 프레임의 지역 변수가 있는 곳에, 팝에 의해 지금은 사라졌지만 이전에 존재하던 스택 프레임이 사용하던 값이 그대로 남아 있기 때문이다.

example\_8-1.c :

```

#include <stdio.h>

int square(int m) {
    return m * m;
}

int sq_add(int f, int s) {
    int total;
    total = square(f) + square(s);
    return total;
}

int main() {
    int first, second, sum;
    printf("Enter two integers.\n");
    scanf("%d%d", &first, &second);
    sum = sq_add(first, second);
    printf("The result is %d.\n", sum);

    return 0;
}

```

## 변수와 스택 프레임

```

sum = sq_add(first, second);
int sq_add(int first, int second)

```

sq_add( )의 스택 프레임 (활성 상태)	매개변수	first, second
	지역 변수	total
main( )의 스택 프레임 (비활성 상태)	매개변수	none
	지역 변수	first, second, sum

- 변수 명이 같아도 다른 변수
  - 서로 다른 스택 프레임에 존재
  - Sq\_add의 first는 sq\_add의 스택 프레임에 존재.
  - main의 first는 접근 불가
  - 매개변수는 물론 지역 변수도 마찬가지

## Visual C의 스택 관리

### Example 8-2 실습 및 해설

```
12 int main() {
13     int first, second, sum;
14     printf("Enter two integers.\n");
15     scanf("%d%d", &first, &second);
16     sum = sq_add(first, second);
17     printf("The result is %d.\n", sum);
18     return 0;
19 }
```

로컬	이름	값	형식
조사식 1	first	3	int
	second	4	int
로컬	자동		
조사식 1	이름	값	형식
C:\sample\#solution1#\Debug\#project1.exe	Enter two integers.		
	3 4		

- Example 8-1

- F9으로 중단점 설정
- 이후 F5로 중단점 직전까지 실행(중간에 입력)
- 디버그 → 창 → 지역 선택. 로컬 창 (Local Window)을 띠움
- F11을 눌러 sq\_add 함수로 진입

## Visual C의 스택 관리

```
6 int sq_add(int f, int s) { 경과 시간 2.000ms
7     int total;
8     total = square(f) + square(s)
9     return total;
10 }
11
12 int main() {
13     int first, second, sum;
14     printf("Enter two integers.\n");
15     scanf("%d%d", &first, &second);
16     sum = sq_add(first, second);
```

로컬	이름	값	형식
조사식 1	f	3	int
	s	4	int
	total	19799768	int
로컬	자동		

호출 스택	이름	언어
project1.exe!sq_add(int f, int s)	[6]	C
	[16]	C

- 로컬 창의 지역 변수가 f, s, total로 바뀜
- 디버그 → 창 → 호출 스택을 선택
  - 호출 스택 창 (Call Stack Window)이 보임
- 노란 화살표가 현재 활성화된 스택 프레임
  - test.exe!sq\_add(int f, int s)
- F11을 두 번 눌러 square 함수로 진입

## Visual C의 스택 관리

The screenshot shows the Visual Studio interface with the code editor displaying C code for calculating the sum of squares. Three windows are open in the background:

- 조사식 1**: Shows the expression `m * m` with value 9.
- 로컬**: Shows local variables `f` (3), `s` (4), and `total` (-858993460).
- 호출 스택**: Shows the call stack:
  - project1.exe!square(int m) 줄 2
  - project1.exe!sq\_add(int f, int s) 줄 8
  - project1.exe!main() 줄 16

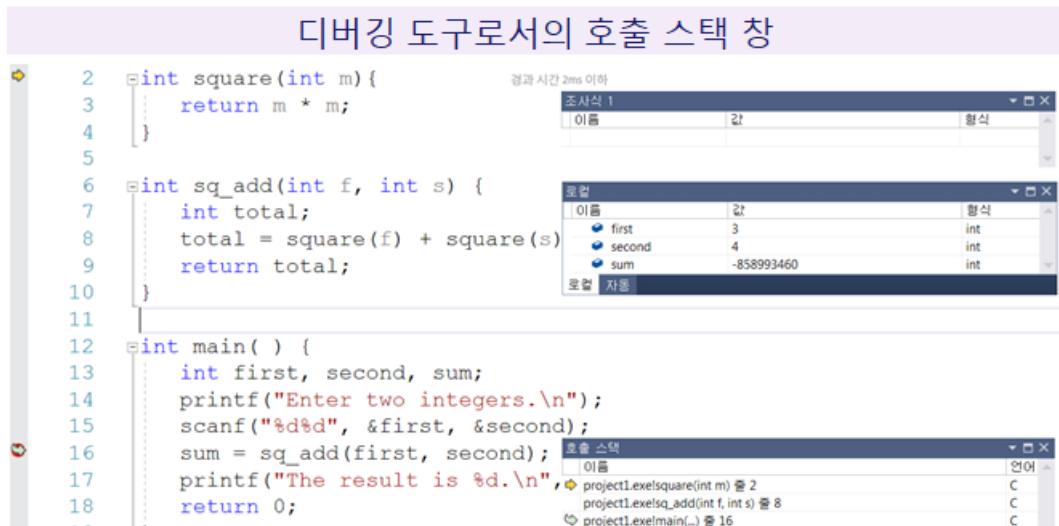
- 호출 스택 창
  - main, sq\_add, square 함수의 스택 프레임이 쌓여 있음.
- 로컬 창
  - square 함수의 지역 변수인 m이 보임

## 디버깅 도구로서의 호출 스택 창

The screenshot shows the Visual Studio interface with the code editor displaying C code for calculating the sum of squares. Three windows are open in the background:

- 조사식 1**: Shows the expression `m * m` with value 9.
- 로컬**: Shows local variables `f` (3), `s` (4), and `total` (-858993460).
- 호출 스택**: Shows the call stack:
  - project1.exe!square(int m) 줄 2
  - project1.exe!sq\_add(int f, int s) 줄 8
  - project1.exe!main() 줄 16

- 호출 스택 창에서 스택 탭 바로 아래 스택 프레임을 더블 클릭
  - 왼편 휘어진 화살표에서 square 함수를 호출함.
  - 로컬 창
    - 호출 당시의 매개변수와 지역 변수인 f, s, total 값



- 다시 그 아래 `main`의 스택 프레임을 더블 클릭
  - 원편 화살표와 로컬 창을 관찰

### example\_8-2.c :

```
#include <stdio.h>

void try(int a, int b) {
    printf("Inside try, a at %p, b at %p.\n", &a, &b);
    printf("Inside try, a = %d, b = %d.\n", a, b);
    a++; b++;
    printf("Upon increasing, a = %d, b = %d.\n\n", a, b);

    return;
}

int main() {
    int one = 1, two = 2;
    printf("In main, one at %p, two at %p.\n", &one, &two);
    printf("In main, one = %d, two = %d.\n\n", one, two);

    try(one, two);

    printf("Back to main, one at %p, two at %p.\n", &one, &two);
    printf("Back to main, one = %d, two = %d.\n\n", one, two);

    return 0;
}
```

```

In main, one at 000000F78059F6F4, two at 000000F78059F714.
In main, one = 1, two = 2.

Inside try, a at 000000F78059F6D0, b at 000000F78059F6D8.
Inside try, a = 1, b = 2.
Upon increasing, a = 2, b = 3.

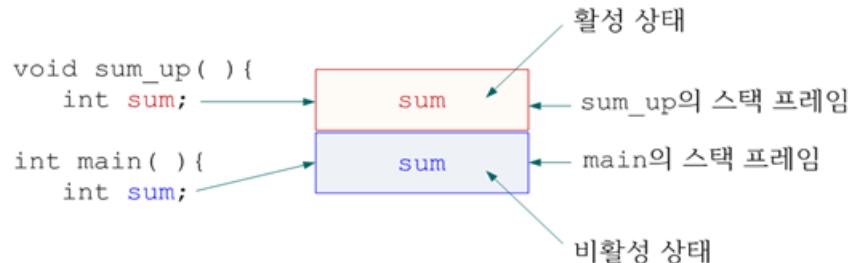
Back to main, one at 000000F78059F6F4, two at 000000F78059F714.
Back to main, one = 1, two = 2.

```

## 8-2. 변수의 영역과 수명

### ● Example 8-3 실습 및 해설

- **변수의 영역 (scope, Visibility)**
  - 변수의 유효 범위
  - 지역 변수 (Local Variables) 나 매개 변수 (Parameter)의 영역은 그 변수를 선언한 함수 내부로 국한됨
  - WHY? 변수의 영역은 현재 활성화된 스택 프레임 내부에 국한됨



- 이름이 같더라도 서로 다른 변수인 이유

### example\_8-3.c :

```

#include <stdio.h>

void sum_up(int n) {
    int sum = 0;
    sum += n;
    printf("Accumulated sum is %d.\n", sum);
}

int main() {
    int num, i;
    for (i = 1; i < 4; i++) {
        printf("Enter an integer.\n");
        scanf("%d", &num);
        sum_up(num);
    }

    return 0;
}

```

```

Enter an integer.
10
Accumulated sum is 10.
Enter an integer.
5
Accumulated sum is 5.
Enter an integer.
2
Accumulated sum is 2.

```

## 전역 변수

- void sum\_up(int n){
 int sum = 0; ①
 sum +=n ;
 printf("Accumulated sum is %d.\n", sum);
 }
- 누계를 구하는 함수이나 제 역할을 못함
- 지역 변수는 함수 호출 시마다 새로 만들어짐
- 해결책: 전역 변수(Global Variable)에 의한 공유

### Example 8-4 실습 및 해설

```

int sum;
void sum_up( ) {
    sum += n;
    ...
}

int main( ){
    int sum = 0;
    ...
    printf(..., sum);
}

```

1

### example\_8-4.c :

```

#include <stdio.h>

int sum;

void sum_up(int n) {
    sum += n;
    printf("Accumulated sum is %d.\n", sum);
}

int main() {
    int num, i; sum = 0;

    for (i = 1; i < 4; i++) {
        printf("Enter an integer.\n");
        scanf("%d", &num);
        sum_up(num);
    }
}

```

```

    }

    printf("Finally, the sum is %d.\n", sum);

    return 0;
}

```

```

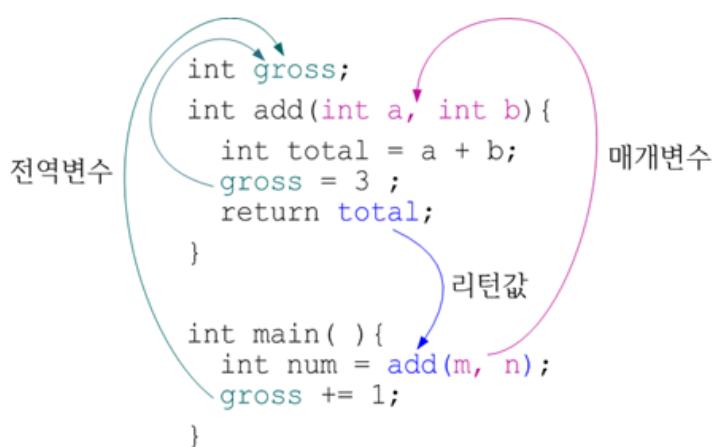
Enter an integer.
1
Accumulated sum is 1.
Enter an integer.
2
Accumulated sum is 3.
Enter an integer.
3
Accumulated sum is 6.
Finally, the sum is 6.

```

## 전역 변수

- 지역 변수와 전역 변수
  - 지역 변수는 그 동네에서만 (함수 내부에서만) 알려진 인사
  - 전역 변수는 전국적으로 (모든 함수에게) 알려진 인사
  - 함수 내부에 선언하면 지역 변수, 함수 외부에 선언하면 전역 변수
  - 전역 변수는 가급적 **프로그램 시작 부분**에 선언 (원 패스 컴파일러)
  - 동일한 이름이라면 **지역 변수가 우선권**

## 함수 사이의 소통 방법



- 세 가지 소통 방법: 전역 변수, 인자(매개 변수), 리턴 값

## 전역 변수

- 전역 변수는 피하는 것이 좋다.
  - 메모리 문제
    - 전역 변수는 프로그램이 시작해서 끝날 때까지 존재
  - 변수 명의 문제
    - 지역 변수는 서로 다른 함수가 동일한 변수명을 써도 됨
    - 전역 변수는 모든 함수가 공유하는 변수. 프로그램을 여럿이 나누어 짤 경우, 변수명이 같으면 컴파일 오류
  - 디버깅 문제
    - 변수가 그릇이라면 전역 변수는 대문 밖에 내다 놓은 그릇
    - 누가 어떤 목적으로 그 그릇을 썼는지 알기 어려움
    - 여러 함수가 동일한 변수를 건드리면 변수값 추적이 어려움
- 함수 사이의 소통은 두 함수 사이에서 이루어져야 한다.
  - 지역 변수를 복사하여 건네주고 또 리턴 값을 건네받는 형태

## 지역 변수

### ● Example 8-5 실습 및 해설

- 지역 변수는 블록 내부에서는 어디든 선언 가능
- 블록을 빠져 나오면 소멸됨

example\_8-5.c :

```
#include <stdio.h>

int main() {
    int i = 1;
    {
        printf("printing from 1 to 3.\n");
        int count = i;
        while (count < 4)
            printf("%d ", count++);
        printf("\n");
        printf("%d\n", count);
    }

    // printf("%d\n", count); // count를 정의된 블록 밖에서 사용했기 때문에 오류
    발생
```

```
    return 0;  
}
```

## 변수의 수명

```
auto int sum;
```

- 변수 영역 (Scope, Visibility)은 공간 개념
- 변수 수명 (Lifetime, Extent)은 시간 개념
- 지역 변수(매개변수)는 함수 호출 시에 생성, 함수 종료 시에 소멸
- 지역 변수는 자동 변수
  - 스택 프레임의 생성과 소멸에 따라 자동으로 수명이 결정됨
  - 대부분 지역 변수가 자동 변수이기 때문에 `auto`를 생략
- 전역 변수의 수명
  - 프로그램이 시작할 때부터 끝날 때까지
  - 지역 변수는 스택 프레임에 저장. 스택 프레임이 팝 되면 소멸
  - 전역 변수는 데이터 세그먼트라는 고정 메모리에 생성
  - 지역 변수는 스택 프레임에 남아 있던 쓰레기 값으로 초기화
  - 전역 변수는 프로그램 시작과 함께 자동으로 0으로 초기화

## 정적 변수

- 변수의 영역에 따른 구분
  - 지역 변수, 전역 변수
- 변수의 수명에 따른 구분
  - 자동 변수, 정적 변수
- 정적 변수 (static Variable)
  - 전역 변수는 기본적으로 정적 변수

### ● Example 8-6 실습 및 해설

- `void sum_up(int n){  
 static int sum = 0;`
- `static`을 붙여 지역 변수를 정적 변수로 만들 수 있다.
- 정적 지역변수는 데이터 세그먼트에 저장됨
- 정적 변수는 단 한 번만 초기화

example\_8-6.c :

```

#include <stdio.h>

void sum_up(int n) {
    static int sum = 0;

    sum += n;
    printf("Accumulated sum is : %d.\n", sum);
}

int main() {
    int num, i;

    for (i = 1; i < 4; i++) {
        printf("Enter an integer.\n");
        scanf("%d", &num);
        sum_up(num);
    }

    return 0;
}

```

```

Enter an integer.
1
Accumulated sum is : 1.
Enter an integer.
2
Accumulated sum is : 3.
Enter an integer.
3
Accumulated sum is : 6.

```

- 전역 변수와 정적 지역 변수와의 차이점 ?
  - 전역 변수는 여러 함수가 공유하지만, 정적 지역 변수는 여전히 지역 변수로 해당 함수 내에서만 접근이 가능하다.
  - 예의 sum 변수는 정적 변수의 특성을 가지면서도 동시에 다른 함수가 접근하지 못하게 하는 지역 변수로서의 장점을 지닌다.

## 레지스터 변수

### ● Example 8-7 실습 및 해설

```
void add( ){
    int i, j;
    register int count = 0;

    for (i = 0; i < MAX; i++)
        for (j = 0; j < MAX; j++)
            count++;
}
```

- 변수를 CPU 레지스터에 저장해 달라는 요구
- WHY? 메모리에 비해 빠른 접근 속도
- 컴파일러 스스로 최적화에 의해 레지스터를 할당하기도 함
- 모든 레지스터가 사용 중이면 할당 받지 못할 수도 있음

example\_8-7.c :

```
#include <stdio.h>

int a; double b; char c;

int main() {
    static int d;
    int e;
    printf("%d, %lf, %x, %d\n", a, b, c, d);
    // printf("%d\n", e); // 초기화되지 않은 지역변수 e를 사용하였으므로 오류 발생

    return 0;
}
```

0, 0.000000, 0, 0

### const

```
double total_money(const int principal){  
    double total;  
    const double rate = 0.03;  
  
    total = principal(1 + rate);           // O.K.  
    principal *= 10;                     // Error  
    return total;  
}
```

- 읽기만 해야 할 변수
  - 쓰기를 범하면 컴파일러 오류 메시지

```
#define rate 0.03  
const double rate = 0.03;
```

- 매크로와의 차이
  - 유효 영역 조정 가능(전역 또는 지역)
  - 타입 매칭 확인 가능

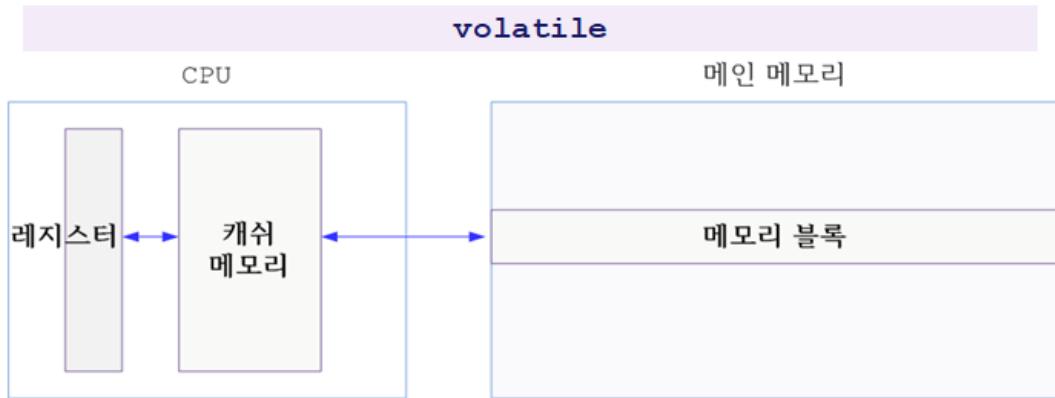
### volatile

```
int ready = 0;  
while (!ready){  
    printf("Waiting for the ready signal.\n");  
}
```

- 프로그래머
  - 루프 도중 외부 코드가 ready를 1로 바꾸어 줄 것을 기대
- 컴파일러
  - while (1)로 최적화(무한 루프)

```
volatile int ready = 0;
```

- 컴파일러 최적화 대상에서 제외



- **Cache Memory**
  - 고속 메모리.
  - 캐시 크기 (32KB ~ 8MB), 블록 단위 (32 ~ 128 Bytes)로 저장
  - 캐시 메모리로 옮긴 이후 메인 메모리 변수 값이 바뀔 경우가 문제. 그 경우 CPU는 **바뀌기 이전 상태**의 변수 값을 읽은 것이 됨

`volatile int ready = 0;`

- 접근이 필요한 바로 그 시점에 **메인 메모리에 직접 접근**
- volatile은 캐시 메모리 와도 연관되어 있다. 대부분의 컴퓨터는 메인 메모리 접근 횟수를 줄이기 위해 메모리를 직접 접근하지 않는다. 대신 일정한 크기(예를 들어 32바이트나 128바이트)의 블록 단위로 메모리 내용을 한꺼번에 가져다가 캐시 메모리에 저장한다. 메인 메모리에 비해 캐시 메모리가 더욱 고속의 메모리이므로 CPU 입장에서는 메인 메모리보다 캐시 메모리를 통해 데이터를 읽고 쓰는 것이 더욱 빠르기 때문이다. 메인 메모리, 캐시 메모리, 레지스터로 가면서 메모리 접근 속도가 빨라진다. 또, 그만큼 가격이 비싸기 때문에 메모리 용량이 제한된다. 역으로 레지스터, 캐시, 메인 메모리 순으로 접근 속도가 느려진다. 메모리에도 계층이 있다는 점에서 이를 메모리 계층(Memory Hierarchyt)라고도 부른다.
- 그런데 문제는 어떤 변수를 메인 메모리에서 캐시 메모리로 옮겨 놓은 상태에서 메인 메모리에 있는 변수 값이 바뀌는 경우다. 만약 그처럼 바뀐 변수 값이 아직 캐시 메모리에 반영되어 있지 않은 상태에서 CPU가 캐시 메모리에 접근하여 변수 값을 읽으면 이는 바뀌기 이전 상태의 변수 값을 읽은 것이다.
- volatile 변수는 최적화 및 캐시 대상에서 제외된다.  
변수를 volatile로 지정하면 그 변수는 컴파일러의 최적화 대상에서 제외될 뿐만 아니라 캐시 대상에서도 제외된다. volatile은 문자 그대로 변덕스럽다는 뜻으로서 값이 자주 바뀐다는 의미다. 따라서 volatile 지정자를 쓰면 언제 값이 바뀔지 모르니 접근이 필요한 바로 그 시점에 직접 메인 메모리로 가서 거기에 저장된 변수 값을 읽어오라는 뜻이 된다. 특히 전역 변수에는 volatile 지정자를 붙여야 할 때가 많다. 하나의 전역 변수를 여러 프로세스가 공유할 때는 어느 시점에 어떤 프로세스가 그 변수 값을 바꾸는지 예측하기 어렵기 때문이다.

### 8-3. 값 호출과 참조 호출

## 값 호출, 참조 호출

### ● Example 8-8 실습 및 해설

- 값 호출 (Call by Value)
  - 사본 전달
- 참조 호출 (Call by Reference, Call by Variable)
  - 원본 전달

example\_8-8.c :

```
#include <stdio.h>

void increment(int m) {
    m++;
    printf("Inside function, m is %d.\n", m);

    return;
}

int main() {
    int a = 10;
    printf("Before calling function, a is %d.\n", a);
    increment(a);
    printf("After calling function, a is %d.\n", a);

    return 0;
}
```

```
Before calling function, a is 10.
Inside function, m is 11.
After calling function, a is 10.
```

## 값 호출

- `void increment(int m)`  
`increment(a);`

increment의 스택 프레임	매개변수	m	10 → 11
	...		
	리턴 값		copy
main의 스택 프레임	지역변수	a	10
	...		

- C 언어는 값 호출 만을 지원한다.
  - increment가 m을 변경하더라도 그 아래 main의 스택 프레임에 있는 원본 a는 불변
- `return m;` 이라고 할 경우 리턴 값도 사본
  - WHY? increment 실행이 끝나면 스택 프레임이 팝 되면서 m 변수는 사라져 버림

## 값 호출

### Example 8-9 실습 및 해설

```
int increment_both(int m, int n) {
    m++; n++;
    printf("Inside function, m is %d and n is %d.\n", m, n);
    return m;
}
```

- 피호출 함수가 변경한 값을 호출 함수에게 돌려주려면
  - 함수 리턴 값을 이용
- 리턴 값은 단 하나여야 한다.
  - m, n 모두를 돌려 줄 수는 없다. (cf. 구조체 리턴)

example\_8-9.c :

```
#include <stdio.h>

int increment(int m) {
```

```

m++;
printf("Inside function, m is %d.\n", m);

return m;
}

int main() {
    int a = 10;
    printf("Before calling function, a is %d.\n", a);
    a = increment(a);
    printf("After calling function, a is %d.\n", a);

    return 0;
}

```

```

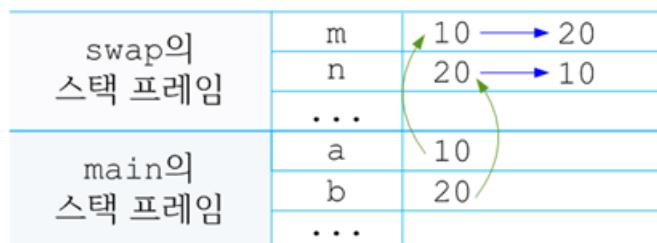
Before calling function, a is 10.
Inside function, m is 11.
After calling function, a is 11.

```

## 값 호출

### ● Example 8-10 실습 및 해설

- 값 호출에 의한 swap
- 원본은 변하지 않음



### example\_8-10.c :

```

#include <stdio.h>

void swap(int m, int n) {
    int temp;
    printf("Before swap, m = %d and n = %d.\n", m, n);
    temp = m;
    m = n;
    n = temp;
    printf("After swap, m = %d and n = %d.\n", m, n);
    return;
}

```

```

}

int main() {
    int a = 10, b = 20;
    printf("Before calling function, a = %d and b = %d.\n", a, b);
    swap(a, b);
    printf("After calling function, a = %d and b = %d.\n", a, b);

    return 0;
}

```

```

Before calling function, a = 10 and b = 20.
Before swap, m = 10 and n = 20.
After swap, m = 20 and n = 10.
After calling function, a = 10 and b = 20.

```

## 8-4. 인라인 함수와 가변 인자 함수

### 인라인 함수

#### ● Example 8-11 실습 및 해설

```

inline int add(int a, int b) {
    return (a + b);
}
sum = add(first, second);

```

- 피 호출 함수의 소스 코드를 적절히 처리하여 복사
  - sum = a + b;로 치환

```
#define add(a, b) ((a) + (b))
```

- 매크로 함수는 전 처리기가 처리, 타입 체킹 없음
- 인라인 함수는 컴파일러가 처리, 타입 체킹.
  - 함수 호출에 따른 오버헤드(스택 프레임)가 없음
  - 비교적 짧은 코드는 인라인 함수로 대체
- 인라인 함수는 프로그램 실행 속도를 높이기 위한 것이다.
- 인라인 함수의 장점은 함수 호출에 따른 오버헤드를 줄일 수 있다는 점이다. 즉, 함수를 호출하는 대신 직접 피호출 함수의 소스코드를 가공하여 삽입함으로써 스택 프레임을 생성하는데 따르는 시간적, 공간적 오버 헤드를 없앨 수 있기 때문이다. 이는 함수 내부 코드가 비교적 간단할 경우 인라인 함수를 쓰므로써 실행 속도를 높일 수 있음을 의미한다.

**example\_8-11.c :**

```

#include <stdio.h>

inline int add(int a, int b) {
    return (a + b);
}

int main() {
    int sum, first = 10, second = 20;
    sum = add(first, second);
    printf("sum is %d.\n", sum);

    return 0;
}

```

## 인라인 어셈블리

### ● Example 8-12 실습 및 해설

- 어셈블리 명령어를 소스 코드에 직접 넣을 수도 있다.
- 컴파일러에게 맡기는 번역에 비해 효율적일 수 있음

#### **example\_8-12.c :**

```

#include <stdio.h>

int power2(int num, int power) {
    __asm {
        mov eax, num;
        mov ecx, power;
        shl eax, cl;
    }
}

int main(void) {
    printf("3 times (2 to the 5) is %d.\n", power2(3, 5));
    return 0;
}

```

### ● Example 8-13 실습 및 해설

- 가변 인자 함수(Variadic Function)
  - 인자의 수가 가변인 함수
  - Ex. `printf`, `scanf`
  - `va_start`, `va_arg`, `va_end` 세 가지 매크로 함수를 호출

**example\_8-13.c :**

```
#include <stdio.h>
#include <stdarg.h> // 1

int find_min(int count, ...) {
    int min, temp, i;

    va_list(p); // 3
    va_start(p, count); // 4
    min = va_arg(p, int); // 5
    for (i = 2; i <= count; i++) {
        temp = va_arg(p, int); // 6
        if (temp < min)
            min = temp;
    }
    va_end(p); // 7

    return min;
}

int main() {
    int arg_count = 3;
    printf("Minimum is %d.\n", find_min(arg_count, 30, 20, 10)); // 8

    return 0;
}
```

- //8에서 호출한 `find_min` 함수는 둘째, 셋째, 넷째 인자인 30, 20, 10 중 가장 작은 수를 돌려주는 함수다. 첫째 인자는 `arg_count`는 뒤따르는 인자의 개수로서 여기서는 3으로 초기화되어 있다. 만약 첫 인자가 2라면 후속 인자도 두 개여야 한다. 이처럼 인자의 개수를 자유롭게 가져가려면 함수를 정의할 때 //2처럼 첫 인자 이후에 세 개의 점을 찍어야 한다.
- 가변 인자 함수를 사용하려면 `va_start`, `va_arg`, `va_end`라는 세 가지 매크로 함수를 호출해야 한다. 이는 //1의 `stdarg.h` 파일에 포함되어 있다.

//3의 va\_list(p)라는 매크로를 확장하면 p가 이 함수에 전달된 인자 리스트를 가리키게 된다. 여기서 p는 프로그래머가 임의로 정한 변수명이다.

이후 //4의 va\_start 매크로에 의해 리스트를 초기화해야 한다. 여기서 count는 //2에서 전달받은 count 와 일치해야 한다.

//5의 va\_arg는 현 위치의 인자를 돌려준다. 처음 호출했으므로 현재로서는 첫 인자다. 괄호 안의 int는 인자의 타입을 의미한다. 매번 va\_arg 매크로를 실행할 때마다 차례차례 그 다음 인자를 돌려준다. 따라서 //5에 의해 첫 인자(예에서는 30)가 min으로 들어간 후 //6에 의해 나머지 인자들이 min과 비교되어 가장 작은 값이 min에 저장된다.

//4의 va\_start로 매크로를 시작했으면 반드시 //7의 va\_end로 매크로를 끝내야 한다.

## 8-5. 재귀 호출

### 재귀 호출

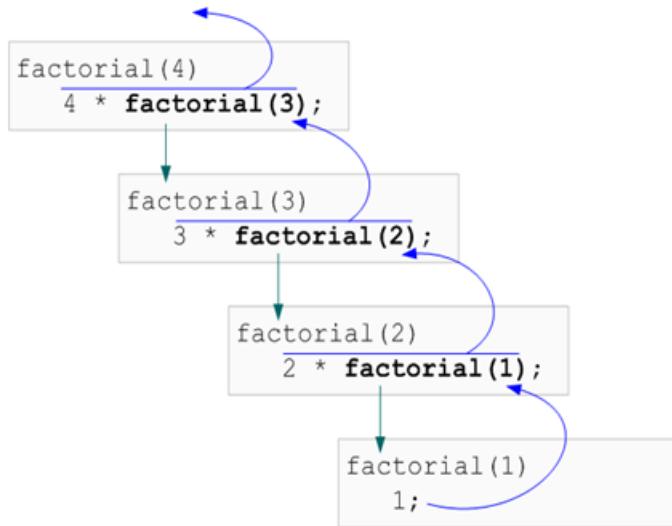
```
n! = n * (n-1) * (n-2) * ... * 1 (단, 1! = 1)
```

```
n! = n * (n - 1)!
```

- **분할 정복 (Divide and Conquer)**
  - 5!을 구하는 문제는 4!을 구하는 문제로, 이는 다시 3!을 구하는 문제로, ..., 결국 1!을 구하는 문제로. 정의에 의해 1!은 1.
  - 작은 문제 역시 큰 문제와 동일한 성격
  - 도미노 게임(100번째 막대기가 반드시 쓰러짐을 증명하라.)
    - 수학적 귀납법 (Mathematical Induction) 과는 **반대 순서**
- **재귀 호출 (Recursive Call)**
  - 재귀 (再歸, Recursion) = 되돌아 옴
  - Recurse = Re + Occur
  - **재귀 함수 (Recursive Function)**
    - 실행 도중 **자기 자신을 호출 (Self Call)** 하는 함수

## 재귀 호출

### ● Example 8-14 실습 및 해설



- 재귀 호출 들어가기와 나오기

**example\_8-14.c :**

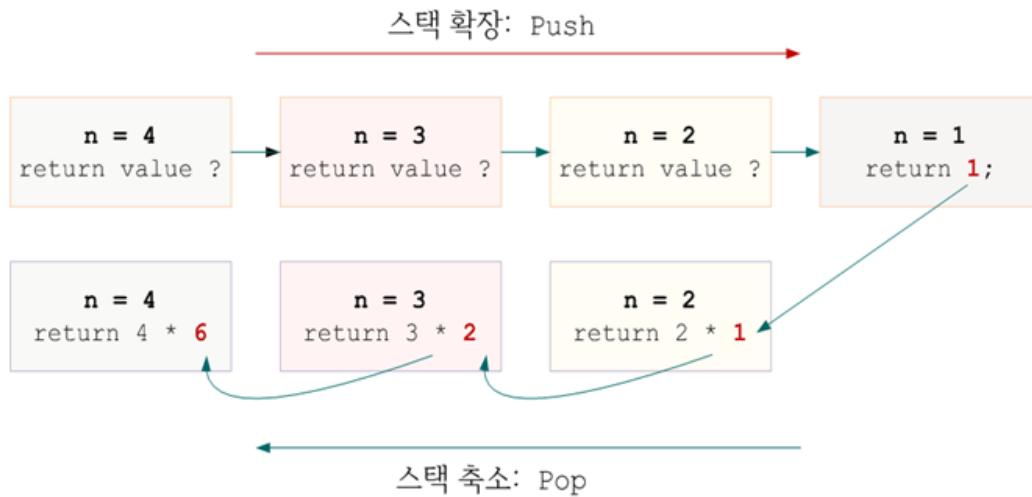
```
#include <stdio.h>

int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return (n * factorial(n - 1));
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    printf("Factorial of %d is %d.\n", num, factorial(num));

    return 0;
}
```

## 재귀 호출에 따른 스택의 변화



## 베이스 케이스

- 베이스 케이스 (Base Case)
  - 또는 Degenerate Case
  - 문제 크기가 충분히 작아져서 직접 해결할 수 있는 경우
  - 예에서는  $n = 1$ 일 때 베이스 케이스
- 모든 알고리즘은 반드시 종료해야 한다
  - 종료 조건 (Termination Condition)이 존재
  - 재귀 호출은 반드시 베이스 케이스에 도달해야 함
  - 그렇지 않으면 스택 오버플로우 (Stack Overflow) 오류

## 꼬리 재귀, 머리 재귀

- Example 8-15 실습 및 해설
- Example 8-16 실습 및 해설

- `printf("%d ", n);`  
`recurse(n - 1);`
  - **꼬리 재귀 (Tail Recursion, End Recursion)**
  - 먼저 일을 한 다음에 재귀 호출로 들어간다.
- `recurse(n - 1);`  
`printf("%d ", n);`
  - **머리 재귀 (Head Recursion)**
  - 먼저 재귀 호출을 한 다음에 되돌아오면서 일을 한다.

### example\_8-15.c :

```
#include <stdio.h>

void recurse(int n) {
    if (n == 0)
        return;
    else {
        printf("%d ", n);
        recurse(n - 1);
    }
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    recurse(num);

    return 0;
}
```

```
Enter a positive integer.
4
4 3 2 1
```

### example\_8-16.c :

```

#include <stdio.h>

void recurse(int n) {
    if (n == 0)
        return;
    else {
        recurse(n - 1);
        printf("%d ", n);
    }
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    recurse(num);

    return 0;
}

```

```

Enter a positive integer.
4
1 2 3 4

```

## 재귀 호출

- Example 8-17 실습 및 해설
  - 명시적 베이스 케이스
- Example 8-18 실습 및 해설
  - 묵시적 베이스 케이스

- Example 8-19 실습 및 해설
  - 10진수의 2진 변환
- Example 8-20 실습 및 해설
  - 문자열을 역순으로 출력하기

**example\_8-17.c (10진수를 역순으로 출력 - 명시적 베이스 케이스) :**

```

#include <stdio.h>

void recurse(int n) {
    if (n == 0)

```

```

        return;
    else {
        printf("%d", n % 10);
        recurse(n / 10);
    }
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    recurse(num);
    printf("\n");
    return 0;
}

```

```

Enter a positive integer.
1234
4321

```

**example\_8-18.c (10진수를 역순으로 출력 - 묵시적 베이스 케이스) :**

```

#include <stdio.h>

void recurse(int n) {
    if (n != 0) {
        printf("%d", n % 10);
        recurse(n / 10);
    }
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    recurse(num);
    printf("\n");

    return 0;
}

```

```

Enter a positive integer.
1234
4321

```

**example\_8-19.c :**

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

```

void to_binary(int n) {
    if (n == 0)
        return;
    else {
        to_binary(n / 2);
        printf("%d", (n % 2));
    }
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    to_binary(num);
    printf("\n");

    return 0;
}

```

```

Enter a positive integer.
14
1110

```

### **example\_8-20.c :**

```

#include <stdio.h>

void reverse_it() {
    char ch;
    scanf("%c", &ch);
    if (ch == '\n')
        return;
    else {
        reverse_it();
        printf("%c", ch);
    }
}

int main() {
    printf("Enter a sentence.\n");
    reverse_it();
    printf("\n");

    return 0;
}

```

```

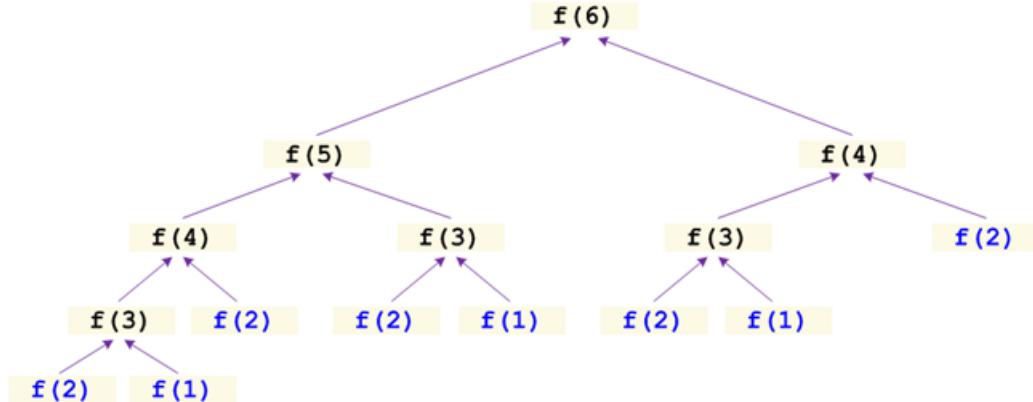
Enter a sentence.
hello
olleh

```

## 재귀 호출

```
1, 1, 2, 3, 5, 8, 13, 21, ....  
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)  
단, fibonacci(2) = fibonacci(1) = 1
```

- Example 8-21 실습 및 해설
  - Fibonacci 수열



example\_8-21.c :

```
#include <stdio.h>

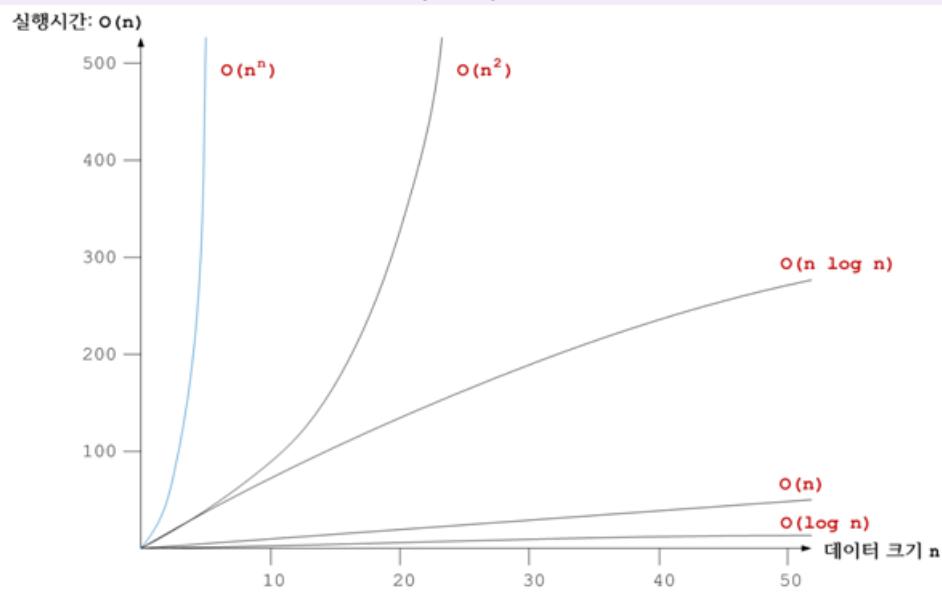
int fib(int n) {
    if (n < 3)
        return 1;
    else
        return (fib(n - 1) + fib(n - 2));
}

int main() {
    int num;
    printf("Enter a number N.\n");
    scanf("%d", &num);
    printf("%dth fibonacci number is %d.\n", num, fib(num));

    return 0;
}
```

```
Enter a number N.  
5  
5th fibonacci number is 8.
```

## 알고리즘의 효율



- $O(n)$ 
  - 알고리즘의 시간적 효율을 나타내는 기호
  - 재귀 호출에 의한 피보나치 알고리즘
    - $O(2^n) = \text{비상식적 시간 (Unreasonable Time)}$

## 재귀 호출의 장단점

- Example 8-22 실습 및 해설
  - 재귀 호출에 의한 Fibonacci
- Example 8-23 실습 및 해설
  - for 루프에 의한 factorial
- 재귀 함수의 단점
  - 속도 면에서 불리 (스택 프레임의 생성과 소멸)
- 재귀 함수의 장점
  - 아름다울 정도로 간명한 코드
  - 데이터가 작을 때, 반복문으로 바꾸기 어려울 때 유리

example\_8-22.c :

```
#include <stdio.h>

long long fib(long long n) {
    if (n < 3)
```

```

        return 1;
    else
        return (fib(n - 1) + fib(n - 2));
}

int main() {
    long long num;
    printf("Enter a number N.\n");
    scanf("%lld", &num);
    printf("%lldth fibonacci number is %lld.\n", num, fib(num));

    return 0;
}

```

- 더 큰 숫자를 표현하기 위해 소스코드에 나타난 int는 모두 long long으로 바꾸었고 그에 따라 입출력도 모두 %lld로 바꾸었다.

#### **example\_8-23.c:**

```

#include <stdio.h>

int factorial(int n) {
    int i, fact = 1;
    for (i = n; i > 0; i--)
        fact *= i;
    return fact;
}

int main() {
    int num;
    printf("Enter a positive integer.\n");
    scanf("%d", &num);
    printf("Factorial of %d is %d.\n", num, factorial(num));

    return 0;
}

```

```

Enter a positive integer.
5
Factorial of 5 is 120.

```

## **9장. 배열**

### **9-1. 필요성 및 정의**

## 9장. 배열

- 배열 소개



- array

- a regular and imposing grouping or arrangement
- a number of mathematical elements arranged in rows and columns
- an arrangement of computer memory elements (as magnetic cores) in a single plane  
(Webster's New Collegiate Dictionary)

### 배열의 필요성

```
printf("%d\n", rabbit1);
printf("%d\n", rabbit2);
printf("%d\n", rabbit3);
...
// 100 마리 토끼라면 100개의 변수 선언?
```

- 여러 변수를 하나의 그룹으로 묶은 것이 배열

```
printf("%d\n", rabbit[1]);
printf("%d\n", rabbit[2]);
printf("%d\n", rabbit[3]);
```

- 인덱스를 써서 개별 변수를 지칭

```
for (i = 0; i < 100; i++)
    printf("%d\n", rabbit[i]);
```

## 배열 선언

```
int rabbit[100];
```

↑  
요소 타입      ↑  
배열 명      ↑  
요소 개수

- 배열 선언(Element Type, Array Name, Number of Elements)
  - `int`는 배열의 타입이 아니라 배열을 구성하는 요소의 타입
- 배열의 모든 요소는 동종 자료형(Homogeneous Data Type)
  - `char char_set[26];`
  - `double math_score[100];`
- C 언어의 배열 인덱스는 0부터 시작
  - `int rabbit[100]`이면 요소 개수가 100개
  - 인덱스는 0부터 99
  - `rabbit[100]`은 존재하지 않음

## 배열 요소의 주소

```
int rabbit[4];
    rabbit[0]  rabbit[1]  rabbit[2]  rabbit[3]
    ┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐
    |          | |          | |          | |          |
    ↓          ↓          ↓          ↓          ↓
100번지    104번지   108번지   112번지
```

```
&rabbit[i] = &rabbit[0] + sizeof(int) * i
```

- 배열은 연속된 메모리 공간(Contiguous Memory Space)에 존재
  - 요소끼리 서로 붙어 있기 때문에 위 공식이 성립
  - cf. 만약 배열 인덱스를 1부터 시작하면 이는  
`&rabbit[i] = &rabbit[1] + sizeof(int) * (i - 1)`로 바뀜
  - `(i - 1)` 계산으로 인한 시간적 부담 초래

## 초기화, 배열 요소의 개수

```
int sales[4] = {5, 6, 7, 8};  
int sales[] = {5, 6, 7, 8};           // 알아서 크기 계산  
int sales[4] = {5, 6};              // 나머지는 0으로  
int sales[4] = {0};                // 모두 0으로
```

- 배열 선언과 동시에 초기화 (cf. 정적 변수로 선언하면 자동으로 0)

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  
             30, 31};  
int size_days = sizeof(days); // size of days[]  
num_elts = sizeof(days) / sizeof(days[0]);  
  
for (i = 0; i < num_elts; i++){  
    printf("Month %d has %d days.\n", i + 1, days[i]);  
}
```

- sizeof 연산자는 배열 전체나 배열 요소의 크기를 계산
  - 배열 전체의 크기 / 첫 요소의 크기 = 요소의 개수

## 배열

### ● Example 9-1 실습 및 해설

- 배열 및 배열 요소 주소 확인

### 배열 명 ≡ 배열 시작 주소

```
rabbit ≡ &rabbit[0]  
&rabbit[i] = rabbit + sizeof(int) * i
```

### ● Example 9-2 실습 및 해설

- 배열에 의한 피보나치 수열 계산

example\_9-1.c :

```
#define MAX 5  
#include <stdio.h>
```

```

int main() {
    int i, sum = 0; double average;
    int rabbit[MAX];

    printf("Enter weights of five rabbits.\n");
    for (i = 0; i < MAX; i++)
        scanf("%d", &rabbit[i]);

    for (i = 0; i < MAX; i++)
        sum += rabbit[i];
    printf("Average weight is %.3lf.\n", ((double)sum) / MAX);

    printf("&rabbit[0] is %p.\n", &rabbit[0]);
    printf("&rabbit[0] is %p.\n", &rabbit[1]);
    printf("rabbit is %p.\n", rabbit);

    return 0;
}

```

```

Enter weights of five rabbits.
10 10 10 10
Average weight is 10.000.
&rabbit[0] is 00000049F18FF788.
&rabbit[0] is 00000049F18FF78C.
rabbit is 00000049F18FF788.

```

### example\_9-2.c :

```

#define MAX 200
#include <stdio.h>

int main() {
    int i, num;
    int fib[MAX];

    printf("Enter a number N.\n");
    scanf("%d", &num);

    fib[0] = fib[1] = 1;
    for (i = 2; i <= num; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    printf("%dth fibonacci number is %d.\n", num, fib[i - 1]);

    return 0;
}

```

```

Enter a number N.
5
5th fibonacci number is 8.

```

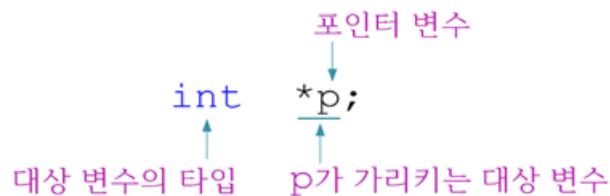
▼ 11/7 목 - 10장

## 10장. 포인터

### 10-1. 포인터 정의

#### 포인터

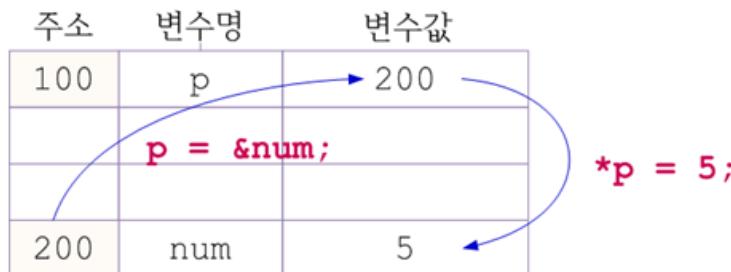
- 포인터 (Pointer) 또는 포인터 변수 (Pointer Variable)
  - 다른 변수를 가리키는 변수
  - 다른 변수의 주소를 담는 변수



- \*p는 p가 가리키는 것을 의미
- Something pointed by p is integer type
- 변수 p가 어떤 주소 값을 담고 있는데 그 주소를 따라가면 다른 변수가 있고 그 변수는 타입이 정수다.
- 일반적인 변수 값을 담는 그릇이 아니라 주소 값을 담는 그릇

## 포인터 활용

```
int num;
int *p;
p = &num;
*p = 5;
```



- 간접 참조 또는 참조 (Dereferencing, Indirection)
  - \*p = 5; 이면 num에 간접 접근
  - num = 5; 이면 num에 직접 접근

## SYMTAB(Symbol Table)

```
char *p;
int *p;
```

- 포인터 선언 시 대상의 타입을 명시. WHY? 포인터가 가리키는 것은 변수의 시작 주소. 거기서부터 몇 바이트를 읽을지를 알아야 함.

### Example 10-1 실습 및 해설

SYMTAB			Main Memory
타입	주소	변수 명	
int	0028F718	num	
...	...	...	
pointer	0028F70C	p	5

- SYMTAB에는 변수 타입, 주소, 변수 명이 저장되어 있음
  - 실행 시 이 도표를 참고로 변수에 접근 (Ex. \*p => num)

### example\_10-1.c :

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

```

int main() {
    int num, *p;

    p = &num; // p에 주소 값 저장
    *p = 5; // *p로 간접 참조를 하여 값 저장

    printf("&num = %p, num = %d.\n", &num, num);
    printf("p = %p, *p = %d, &p = %p\n", p, *p, &p);

    printf("sizeof(p) = %d.\n", sizeof(p));
    printf("sizeof(*p) = %d.\n", sizeof(*p));

    return 0;
}

```

```

&num = 00000047446FF784, num = 5.
p = 00000047446FF784, *p = 5, &p = 00000047446FF7A8
sizeof(p) = 8.
sizeof(*p) = 4.

```

## 어셈블리 언어와 포인터

OP code	Operand
---------	---------

- OP(eration) code(연산자), Operand(피 연산자)

```

LOAD AX, #200
a = 200;

```

LOAD	AX, #200
------	----------

- 즉시 접근 (Immediate Addressing)
  - 상수 200을 명령문에 저장
  - 메모리 접근 불필요

## 어셈블리 언어와 포인터

```
LOAD AX, 200  
a = b;
```



- 직접 접근 (Direct Addressing)
  - 피 연산자의 메모리 주소를 지정
  - SYMTAB에 변수 b의 주소가 200번지라면 이는 a = b;에 해당
  - 한 번의 메모리 접근

## 어셈블리 언어와 포인터

```
LOAD AX, @200  
a = *b;
```



- 간접 접근 (Indirect Addressing)
  - 피 연산자의 메모리 주소를 담은 변수의 주소를 지정
  - c 언어의 포인터에 해당
  - 두 번의 메모리 접근
  - 즉시 접근, 직접 접근, 간접 접근 순으로 속도 저하
  - 접근할 수 있는 주소의 범위를 늘리는 수단
    - 포인터 변수의 비트 수 > 피 연산자의 비트 수

## 포인터 유의사항

```
int * p;           // O.K.  
int *p;           // 동일. p가 가리키는 것이 정수  
int* p;           // 동일. p는 정수 포인터  
int* p, q;        // q는 포인터가 아님  
                   // int *p, *q; 라고 해야 함
```

```
int* p;  
p = 240;          // 우변이 (int*) 240이 아니라서 경고  
                   // 그러나 직접 번지수를 대입할 경우 접근 위반  
                   // 메모리 주소는 운영체제가 관리  
printf("Pointer value is %p.\n", p);
```

- $p = 240;$  처럼 포인터 변수에 직접 번지 수를 대입해도 printf에 의해 값을 찍을 수는 있다. 그러나 컴파일러의 경고 메시지가 따른다.  $\text{int } * p;$ 에서  $p$  변수의 타입을  $(\text{int } *)$ , 즉 정수 포인터로 선언했지만  $p = 240;$ 의 우변은 포인터 타입이 아니라 정수 타입이기 때문이다.

물론 240 대신  $(\text{int } *) 240;$  처럼 형 변환 연산자를 가해서 타입을 일치시킬 수는 있다. 그러나 메모리를 관리하는 것은 운영체제의 고유 권한이기 때문에 프로그래머가 구체적인 번지수를 대면서 해당 주소의 데이터에 직접 접근하는 것은 허용되지 않는다.

예를 들어, 프로그래머가 운영체제가 상주하는 주소에 접근하여 운영체제 프로그램을 뜯어고치도록 놔둘 수는 없는 노릇이다. 실제로도 C 프로그래머가 직접 포인터 변수에 주소를 대입할 일은 거의 없다.

## L-value, R-value

```
*p = 5;  
*p = *p + 5;      // 좌변은 L-value, 우변은 R-value  
printf("%d", *p);  // 인자는 R-value  
&num = 5;          // 오류. 주소 연산의 결과는 상수로서 R-value
```

주소	변수명	변수값
100	p	200
200	num	5

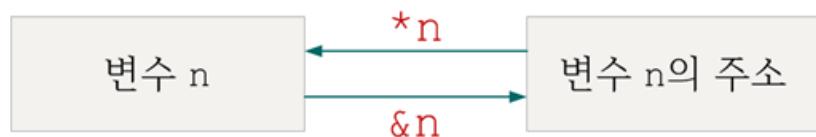
\*p: L-value      \*p: R-value

- $*p$ 가 L-value
  - 변수 자체를 의미
- $*p$ 가 R-value
  - 변수의 값을 의미

- 포인터를 쓸 때도 L-value와 R-value의 차이점에 유의해야 한다. `*p = 5;`의 `*p`는 대입문의 좌변에 있으므로 L-value다. 따라서 이는 변수 자체를 의미한다. `p`가 가리키는 변수에 5를 대입하라는 것이다.
- 그러나 `*p = *p + 5;`의 우변에 있는 `*p`는 R-value로서 변수 자체가 아니라 그 변수의 값을 의미한다. `*p = *p + 5;`의 의미는 `p`가 가리키는 변수 값에 5를 더해서 그것을 `p`가 가리키는 변수에 대입하라는 뜻이다.
- `printf("%d", *p);`의 `*p`도 R-value다. `printf` 문의 인자는 R-value로 간주되기 때문에 여기서 `*p`는 변수 자체가 아니라 변수의 값을 의미한다.
- `&num = 5;`는 오류다. `&num`에서 `&`는 주소 연산자로서 연산의 결과인 주소는 상수, 즉 R-value다. 대입문의 좌변에 상수가 올 수는 없다.
  - `&num`은 변수 `num`의 주소를 나타내는 R-value이다.
  - R-value는 변경 불가능한 값이므로, 대입문의 좌변(L-value)에 올 수 없다.
  - 따라서 `&num = 5;`는 문법적으로 올바르지 않다.

## 역함수

### Example 10-2 실습 및 해설



- 역함수 (Inverse Function)
  - 곱하기와 나누기는 역함수 관계
  - `*n`과 `&n`은 역함수 관계
  - `&`는 주소 연산자로서 변수로부터 주소를 찾으라는 것
  - `*`는 참조 연산자로서 주소로부터 변수를 찾으라는 것
  - 역함수끼리는 상쇄

`*(&n)` is equal to `n`.  
`(*(&n))` is equal to `*n`.

#### example\_10-2.c :

```

#include <stdio.h>

int main() {
    int first = 50, second = 200, * p;

    printf("&first is %p.\n", &first);
    printf("&p is %p.\n", &p);

    p = &first;
  
```

```

printf("p is %p.\n", p);
printf("*p is %d.\n", *p);

*p += 50;
printf("*p changed to %d.\n", *p);

p = &second;
printf("*p changed to %d.\n", *p);

return 0;
}

```

```

&first is 0000004B21D0F5A4.
&p is 0000004B21D0F5E8.
p is 0000004B21D0F5A4.
*p is 50.
*p changed to 100.
*p changed to 200.

```

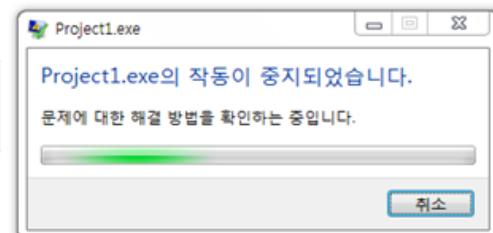
## 포인터 관리

```

double num = 2.0;
int *p;
p = &num;
• Type Mismatch Error

int *p = NULL;
*p = 100;           // Error

```



- **NULL은 가리키는 대상이 없다는 의미**
- **\*p = 100;**에 의해 프로그램 실행 중 **크래쉬** 창이 뜸
  
- C 언어에서 포인터 관리는 **프로그래머의 책임**
  - 비롯한 대부분 언어가 포인터를 제공하지 않음 (WHY?)
  - 포인터는 C 언어의 **단점이자 최대 장점**

## NULL 포인터

```
int *p = NULL;  
#define NULL 0  
#define NULL ((void *) 0)
```

- 매크로 상수 NULL

- 0 번지가 아니라 아무 것도 가리키지 않는다는 의미

```
int *p = NULL;  
...  
if (p == NULL) {  
    printf("p points to nothing.\n");  
    exit(1);  
}  
*p = 100;
```

- 가리키는 것이 없는 포인터에는 NULL을 대입 (초기화 포함)
- 포인터를 따라갈 때에는 미리 NULL인지 확인

- 포인터가 아무것도 가리키지 않을 때는 int \* p = NULL;처럼 NULL이라는 매크로 상수로 초기화하는 것이 좋다. NULL 값을 가진 포인터를 널 포인터라고 부른다. 여기서 NULL은 아무것도 가리키지 않는다는 의미로 쓰인다.  
표준 라이브러리 헤더는 NULL을 #define NULL 0처럼 정의하고 있다. 여기서 0을 주소 0번지라는 의미로 해석할 수도 있지만 실제는 이는 아무 것도 가르키지 않는다는 뜻으로 약속한 상수다.
- 컴파일러에 따라서는 NULL을 #define NULL ((void \*) 0)처럼 정의하기도 한다. 정수형 변수 x에 대해 (double)x라고 하면 x가 double형으로 바뀐다. 이른바 형 변환 연산자다. 또, (int)x라고 하면 x가 정수 포인터 형으로 바뀐다.  
같은 맥락에서 (void \*)0은 정수 0을 포인터 형으로 바꾸라는 의미다. 다시 말해서 0을 주소 0번지로 간주하라는 의미다. 또, void\*는 포인터가 가리키는 대상의 타입이 아직 정해지지 않았다는 의미다. 물론 그렇게 하더라도 NULL 값이 여전히 정수 0임에 변함이 없다. 프로그래머로서는 자신이 쓰는 컴파일러가 NULL을 어떤 방식으로 정의했는지 알 필요가 없다. 가리키는 것이 없을 때 단순히 NULL이라고만 쓰고 그것이 정수 0이라고 생각하면 된다.
- 조심성 있는 프로그래머는 int \*p = NULL;처럼 포인터 변수를 항상 NULL로 초기화한다. 그리고 조금 번거롭지만 포인터를 따라갈 때는 if(p == NULL)처럼 항상 그것이 널 포인터인지 확인한다. 만약 널 포인터라면 프로그래머가 직접 작성한 오류 메시지를 띄운 후 exit로 빠져나감으로써 크래시를 예방할 수 있다. 결국 NULL이 아닐 때만 실행되므로 안전하다.

## 타입 지정자 const

```
int num = 1;
const int *p = &num;           // *p는 상수
*p = 5;                      // 오류
```

```
int num1 = 1, num2 = 2;
int* const p = &num1;         // p는 상수
*p = 5;                      // O.K.
p = &num2;                   // 오류
```

```
const int num = 10;           // num은 상수
const int *p = &num;          // O.K., *p는 상수
int *p = &num;              // 컴파일러 경고 (Warning)
                           // *p(num)를 바꿀 우려
```

- `const int num = 10;`에 의해 `num` 변수를 상수로 선언하면서 초기값을 부여하고 있다. 이른바 상수 변수다. 이렇게 되면 `num` 값이 고정되어 이후에는 바꿀 수가 없다.  
`const int * p = &num;`는 포인터 `p`를 `&num`으로 초기화하면서 `*p`가 상수임을 선언하고 있다. 이는 `const int num = 10;`과도 부합하는 정의다. `num`을 상수로 선언했고 `p`를 `&num`으로 초기화했으니 `*p`도 당연히 상수여야 하기 때문이다.  
`int * p = &num;`에 대해서는 컴파일러 경고가 따른다. 좌변에 `const` 지정자가 없으므로 컴파일러는 `*p`가 변수라고 생각한다. 그런 상태에서 `p`를 `&num`으로 초기화하면 이후에 `*p` 값, 즉 `num` 값을 바꿀 우려가 있기 때문이다.

## 10-2. 포인터와 참조 호출

## 포인터에 의한 참조 호출

### Example 10-3 실습 및 해설

	변수	주소	변수값
call_by_value	p		500 → 501 copy
main	num	200	500

	변수	주소	변수값
call_by_reference	p		200
main	num	200	500 → 501 copy *p += 1;

- 참조 호출

- 변수 값(500)이 아니라 변수의 주소 값(200번지)을 복사
- 복사된 값을 따라가서 원본 변수 값을 변경
- C 언어는 주소를 통해서는 아래에 깔린 스택 프레임에 접근 허용

example\_10-3.c :

```
#include <stdio.h>

void call_by_value(int p) {
    p += 1;
}

void call_by_reference(int* p) {
    *p += 1;
}

int main() {
    int num;

    num = 500;
    printf("Before call_by_value, num is %d.\n", num);
    call_by_value(num);
    printf("After call_by_value, num is %d.\n", num);

    num = 500;
    printf("Before call_by_reference, num is %d.\n", num);
    call_by_reference(&num);
    printf("After call_by_reference, num is %d.\n", num);
}
```

```
    return 0;  
}
```

```
Before call_by_value, num is 500.  
After call_by_value, num is 500.  
Before call_by_reference, num is 500.  
After call_by_reference, num is 501.
```

## 입력 인자, 출력 인자

### ● Example 10-4 실습 및 해설

- 큰 수와 작은 수 구하기

- void min\_max(int a, int b, int\* min, int\* max);
- a, b는 입력 인자
  - 값 호출
  - 대개 읽기 전용. 쓰기를 하더라도 원본은 불변
- min, max는 출력 인자
  - 참조 호출
  - 함수 내부에서 min, max를 변경하면 원본 변경
  - 함수의 리턴 값이 두 개인 셈

#### example\_10-4.c :

```
#include <stdio.h>  
  
void min_max(int a, int b, int* min, int* max) {  
    if (a < b) {  
        *min = a;  
        *max = b;  
    }  
    else {  
        *min = b;  
        *max = a;  
    }  
}  
  
int main() {  
    int first, second, smaller, larger;  
    printf("Enter two different integers.\n");  
    scanf("%d%d", &first, &second);
```

```

min_max(first, second, &smaller, &larger);
printf("Smaller : %d, Larger : %d\n", smaller, larger);

return 0;
}

```

```

Enter two different integers.
5 10
Smaller : 5, Larger : 10

```

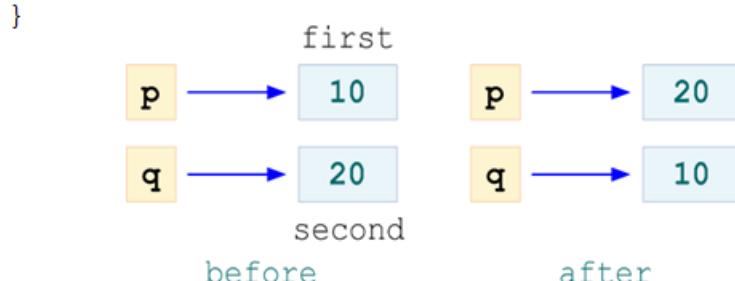
## 포인터에 의한 원본 스왑

- Example 10-5 실습 및 해설

- `int* larger(int* a, int* b); // 포인터를 리턴하는 함수`

- Example 10-6 실습 및 해설

- `swap(&first, &second);`
- `void swap(int *p, int *q) {`
- `int temp;`
- `temp = *p;`
- `*p = *q;`
- `*q = temp;`



### example\_10-5.c :

```

#include <stdio.h>

int* larger(int* a, int* b) {
    if (*a > *b)
        return a;
    else
        return b;
}

int main() {
    int first, second, *p;
    printf("Enter two different integers.\n");
    scanf("%d%d", &first, &second);
    p = larger(&first, &second);
    printf("%d is larger.\n", *p);
}

```

```
    return 0;  
}
```

```
Enter two different integers.  
2 5  
5 is larger.
```

### example\_10-6.c :

```
#include <stdio.h>  
  
void swap(int*, int*);  
  
int main() {  
    int first, second;  
    printf("Enter two numbers to swap.\n");  
    scanf("%d%d", &first, &second);  
    printf("Before swap : first = %d and second = %d.\n", first, second);  
    swap(&first, &second);  
    printf("After swap : first = %d and second = %d.\n", first, second);  
  
    return 0;  
}  
  
void swap(int* p, int* q) {  
    int temp;  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

```
Enter two numbers to swap.  
10 20  
Before swap : first = 10 and second = 20.  
After swap : first = 20 and second = 10.
```

## 10-3. 포인터에 의한 배열 전달

## 배열 단위의 대입 및 복사

```
int a[4] = {1, 2, 3, 4};  
int b[4] = {5, 6, 7, 8};  
a = b; // Error  
if (a == b) // Error  
for (i = 0; i < 4; i++) // O.K.  
    a[i] = b[i];
```

- C 언어는 배열 단위의 대입이나 복사를 허용하지 않는다.
  - WHY? C 언어 설계 과정에서 실행 속도를 고려
- 배열 명은 그 자체로 배열의 시작 주소
  - 배열 명은 상수 포인터
    - a, b는 배열이 만들어질 때 할당된 시작 주소(불변)
    - 상수는 L-value가 아니기 때문에 좌변에 올 수 없음
  - (a == b)
    - 배열 요소가 같은지를 비교하는 것이 아님
    - 배열의 시작 주소가 같은지를 비교. 당연히 다름

## 배열을 인자로 전달하기

### ● Example 10-7 실습 및 해설

- int rabbit[MAX] = {10, 20, 30, 40}; // 배열 선언
- increment(rabbit); // 함수 호출
- void increment(int rb[]); // 피 호출 함수
- 배열 명 = 배열 시작 주소 = 포인터
- 배열 명을 인자로 전달하면 포인터가 복사되어 넘어감
  - 참조 호출 효과를 기할 수 있음

	변수 명	시작 주소	변수 값
increment의 스택 프레임	rb	96	copy 100
main의 스택 프레임	rabbit[0]	100	10 → 11
	rabbit[1]	104	20 → 21
	rabbit[2]	108	30 → 31
	rabbit[3]	112	40 → 41

example\_10-7.c :

```

#define MAX 4
#include <stdio.h>

void increment(int rb[]) {
    int i;
    for (i = 0; i < MAX; i++)
        rb[i]++;
}

int main() {
    int i, rabbit[MAX] = { 10, 20, 30, 40 };
    increment(rabbit);
    for (i = 0; i < MAX; i++)
        printf("%d ", rabbit[i]);
    printf("\n");

    return 0;
}

```

11 21 31 41

## 배열 인자 선언

- `void increment(int rb[ ]);`
  - rb는 배열 명이자 포인터
- `void increment(int rb[MAX]);`
  - 상동.
  - 컴파일러는 포인터 값만 복사. 요소 개수에는 무관심
- `Void increment(int* rb);`
  - 상동.
  - 포인터 rb가 가리키는 것이 int. 배열이라는 의미가 약함

```

increment(rabbit[3]);      // 배열 요소 전달
void increment(int rb3);   // 값 호출

```

```

void increment(const int rb[])
// 원본 배열을 읽기만 할 때에는 const를 붙이는 것이 좋음

```

- 배열 요소 자체를 전달하면 참조 호출 효과를 기할 수 없다. 예를 들어, main이 increment(rabbit[3])처럼 호출하고 increment가 void increment(int rb3);처럼 받았다면 이것은 완전한 값 호출이다. rabbit[3]은 변수 주소가 아니라 변수 자체이기 때문에 rb3에는 변수 값이 복사되어 들어간다. 따라서 되

돌아오면 rabbit[3]은 바뀌지 않는다. 배열 요소 하나만 바꾸려면 이 역시 &rabbit[3]로 해서 포인터로 호출하고 int \* rb3로 해서 포인터로 받아야 한다.

- 피호출 함수가 호출 함수의 원본 변수를 변경해야 할 때도 있지만 읽기만 해야 할 때도 있다. 그럼에도 불구하고 실수로 원본 변수를 변경할 때도 있다. 그 경우 피호출 함수를 예처럼 선언하는 것이 좋다. 여기서 const 지정자는 rb 배열 요소 하나하나를 상수라고 선언한 것이기 때문에 함수 내부에서 배열 요소를 변경할 수 없게 된다. 만약 함수 내부에 원본 변수를 변경하는 명령문이 있을 경우 컴파일 단계에서 오류 메시지가 뜬다. 따라서 읽기만을 원할 때는 무조건 const 지정자를 붙이는 것은 상당히 좋은 습관이다. 프로그래머의 부주의로 인한 오류를 방지할 수 있기 때문이다.

## 배열 요소의 실제 개수

### Example 10-8 실습 및 해설

- `int rabbit[MAX];`
  - 배열 요소의 최대 개수는 MAX
  - 컴파일 시에 확정되어야 컴파일러가 스택 프레임의 크기를 확정
    - 배열 = 정적 데이터 타입 (static Data Type)
- `void print_array(const int arr[], int length) {  
 int i;  
 for (i = 0; i < length; i++)  
 printf("%d ", arr[i]);  
}`
- 배열 요소의 최대 개수와 실제로 들어간 개수가 다를 수 있음
  - length는 실제로 들어간 개수를 추적

example\_10-8.c :

```
#define MAX 100  
#include <stdio.h>  
  
void print_array(const int arr[], int length) {  
    int i;  
    for (i = 0; i < length; i++)  
        printf("%d ", arr[i]);  
}  
  
int main() {  
    int i, temp, rabbit[MAX];  
    printf("Enter an integer.\n");  
    scanf("%d", &temp);  
    for (i = 0; temp >= 0; i++) {  
        rabbit[i] = temp;  
        printf("Enter an integer.\n");  
    }  
}
```

```

        scanf("%d", &temp);
    }
print_array(rabbit, i);
printf("\n");

return 0;
}

```

```

Enter an integer.
1
Enter an integer.
2
Enter an integer.
3
Enter an integer.
4
Enter an integer.
5
Enter an integer.
-1
1 2 3 4 5

```

## 선택 정렬

- 정렬 (Sorting)
  - 오름차순 (Ascending Order) 과 내림차순 (Descending Order)
- 선택 정렬 (Selection Sort)

	70	90	30	40	20
1 단계	70	20	30	40	<b>90</b>
2 단계	40	20	30	<b>70</b>	
3 단계	30	20	<b>40</b>		
4 단계	<b>20</b>	<b>30</b>			

- 매 단계 가장 큰 것이 오른 쪽 끝으로

### Example 10-9 실습 및 해설

#### example\_10-9.c :

```

#define MAX 100
#include <stdio.h>

void print_array(const int arr[], int length);
void swap(int*, int*);

```

```

void selection_sort(int arr[], int length);

int main() {
    int i = 0, temp, data[MAX];
    printf("Enter integers seperated by a blank.\n");
    while (1) {
        scanf("%d", &temp);
        if (temp < 0)
            break;
        data[i++] = temp;
    }

    printf("Before sorting: ");
    print_array(data, i);
    printf("\n");

    selection_sort(data, i);
    printf("After sorting: ");
    print_array(data, i);
    printf("\n");

    return 0;
}

void print_array(const int arr[], int length) {
    int i;
    for (i = 0; i < length; i++)
        printf("%d ", arr[i]);
}

void swap(int* p, int* q) {
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

void selection_sort(int arr[], int length) {
    int last, largest, current;

    for (last = length - 1; last > 0; last--) {
        largest = 0;
        for (current = 1; current <= last; current++) {
            if (arr[current] > arr[largest])
                largest = current;
        }
        swap(&arr[largest], &arr[last]);
    }
}

```

```
    }  
}
```

```
Enter integers seperated by a blank.  
5 2 4 1 3 -1  
Before sorting: 5 2 4 1 3  
After sorting: 1 2 3 4 5
```

## 버블 정렬

1 단계	7 9 3 4 2	7 9 3 4 2	7 3 9 4 2	7 3 4 9 2	7 3 4 2 9
2 단계	7 3 4 2	3 7 4 2	3 4 7 2	3 4 2 7	
3 단계	3 4 2	3 4 2	3 2 4		
4 단계	3 2	2 3			

- 선택 정렬
  - 가장 큰 숫자를 마지막 숫자와 단 번에 스왑
- 버블 정렬
  - 여러 번의 스왑을 거쳐 가장 큰 숫자가 마지막으로 감

- Example 10-10 실습 및 해설
- Example 10-11 실습 및 해설

### example\_10-10.c :

```
#define MAX 100  
#include <stdio.h>  
  
void print_array(const int arr[], int length);  
void swap(int*, int*);  
void bubble_sort(int arr[], int length);  
  
int main() {  
    int i = 0, temp, data[MAX];  
    printf("Enter integers seperated by a blank.\n");  
    while (1) {  
        scanf("%d", &temp);  
        if (temp < 0)  
            break;  
        data[i++] = temp;  
    }  
  
    printf("Before sorting: ");
```

```

    print_array(data, i);
    printf("\n");

    bubble_sort(data, i);
    printf("After sorting: ");
    print_array(data, i);
    printf("\n");

    return 0;
}

void print_array(const int arr[], int length) {
    int i;
    for (i = 0; i < length; i++)
        printf("%d ", arr[i]);
}

void swap(int* p, int* q) {
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

void bubble_sort(int arr[], int length) {
    int pass, current;

    for (pass = 1; pass < length; pass++) {
        for (current = 0; current < (length - pass); current++) {
            if (arr[current] > arr[current + 1])
                swap(&arr[current], &arr[current + 1]);
        }
    }
}

```

```

Enter integers separated by a blank.
5 2 4 1 3 -1
Before sorting: 5 2 4 1 3
After sorting: 1 2 3 4 5

```

### **example\_10-11.c :**

```

#define MAX 100
#include <stdio.h>

void print_array(const int arr[], int length);
void swap(int*, int*);
void bubble_sort(int arr[], int length);

```

```

int main() {
    int i = 0, temp, data[MAX];
    printf("Enter integers seperated by a blank.\n");
    while (1) {
        scanf("%d", &temp);
        if (temp < 0)
            break;
        data[i++] = temp;
    }

    printf("Before sorting: ");
    print_array(data, i);
    printf("\n");

    bubble_sort(data, i);
    printf("After sorting: ");
    print_array(data, i);
    printf("\n");

    return 0;
}

void print_array(const int arr[], int length) {
    int i;
    for (i = 0; i < length; i++)
        printf("%d ", arr[i]);
}

void swap(int* p, int* q) {
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

void bubble_sort(int arr[], int length) {
    int pass, current, sorted = 0;
    // sorted : 이미 정렬이 되어 있는지를 확인하기 위한 플래그 변수

    for (pass = 1; (pass < length) && (!sorted); pass++) {
        sorted = 1;
        for (current = 0; current < (length - pass); current++) {
            swap(&arr[current], &arr[current + 1]);
            sorted = 0;
        }
    }
    // 일단 매 단계에 들어갈 때마다 이미 정렬이 되어 있다고 가정한다.
    // 이어 루프 내에서 한 번이라고 스와핑이 있었으면 sorted 값을 거짓으로 바꾼다.
}

```

```

    // 그렇지 않으면 초기화 값인 true가 그대로 유지되기 때문에 바깥쪽 루프를 빠져나온다.
    // 따라서 그 다음 단계로 넘어가지 않는다.
}

```

▼ 11/14 목, 11/21 목 - 11장

## 11장. 포인터와 배열

### 11-1. 포인터 산술 연산

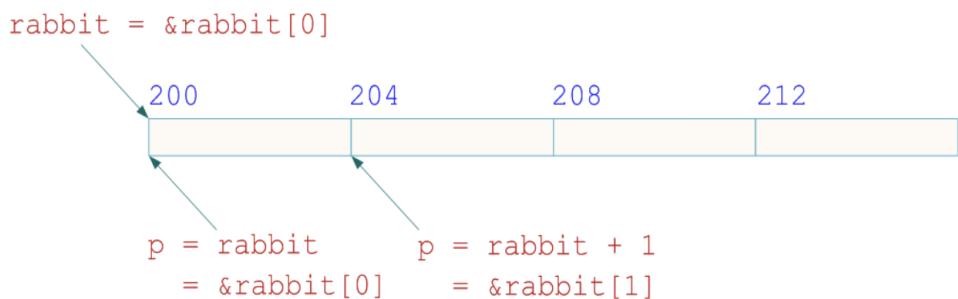
#### 포인터 산술 연산

<code>int *p;</code>	① <code>p = 400, p + 1 = 404, p + 2 = 408</code>
<code>char *p;</code>	② <code>p = 400, p + 1 = 401, p + 2 = 402</code>
<code>double *p;</code>	③ <code>p = 400, p + 1 = 408, p + 2 = 416</code>

- 포인터 산술 연산 (Pointer Arithmetics)
  - 포인터에 숫자를 더하거나 빼는 연산
  - 가리키는 대상의 타입에 따라 연산의 결과가 달라짐
  - 바이트 단위로 주소를 지정하는 컴퓨터를 가정
  
- 포인터 사이의 뱃셈
  - 같은 배열 사이에서만 의미가 있음
  - `P = &arr[1]; q = &arr[3];`
  - `q - p = 2.` 두 포인터 사이에 있는 요소의 개수
  - 포인터 사이의 뱃셈은 허용되지 않음(무의미)

## 배열과 포인터 산술 연산

```
int rabbit[4], *p;  
  
p = rabbit;           // p와 rabbit 모두가 정수 포인터  
p = &rabbit[0];       // 상동  
  
p = rabbit + 1;      // p는 정수 포인터이기 때문에 4 바이트씩 증가  
p = &rabbit[1];       // 상동
```



## 포인터와 증감 연산자

```
int num, *p, rabbit[ ] = {10, 20, 30};  
p = rabbit;  
num = *(++p);           // num = 20  
p = rabbit;  
num = ++(*p);           // num = 11
```

### ● Example 11-1 실습 및 해설

- $*p++ = *(p++)$ 
  - 후위 증감 연산자가 참조 연산자보다 우선 순위가 높음
- $*p++$ 는 포인터 값을 증가.  $(*p)++$ 는 변수 값을 증가

example\_11-1.c :

```
#include <stdio.h>  
  
int main() {  
    int i, *p, arr[] = {5, 10, 15, 20, 25};
```

```

printf("arr[0] is %d.\n", arr[0]);
printf("arr is %d.\n", *arr);

p = arr;
for (i = 0; i < 5; i++)
    printf("%d\t", *(p + i));
printf("\n");

p = arr;
for (i = 0; i < 5; i++)
    printf("%d\t", *p++); // *p++ = *(p++)로 *p++는 포인터 값을 증가
printf("\n");

p = arr;
for (i = 0; i < 5; i++)
    printf("%d\t", (*p)++); // (*p)++는 변수 값을 증가
printf("\n");
}

```

```

arr[0] is 5.
arr is 5.
5      10      15      20      25
5      10      15      20      25
5      6       7       8       9

```

메모리 창

The screenshot shows a debugger interface with three windows:

- 메모리 1**: A memory dump window showing memory starting at address 0x002FFC70. It displays two rows of memory, each containing five bytes: 05 00 00 00 0a, 00 00 00 00 0f, followed by several question marks.
- 자동**: An auto variable table showing the current values of variables:
 

이름	값	형식
*arr	5	int
arr	0x002ffc70 {5, 10, 15, 20, 25}	int[5]
i	-858993460	int
p	0x002ffc70 {5}	int *
- 조사식 1**: A watch expression table showing the value of arr[0]:
 

이름	값	형식
arr[0]	5	int
p	0x002ffc70 {5}	int *

Code pane:

```

2 int main( ){
3     int i, *p, arr[ ] = {5, 10,
4     printf("arr[0] is %d.\n", arr[0]);
5     printf("*arr is %d.\n", *arr);
6
7     p = arr;
8     for (i = 0; i < 5; i++)
9         printf("%d\t", *(p + i));
10    printf("\n");
11
12    p = arr;
13    for (i = 0; i < 5; i++)
14        printf("%d\t", *p++);
15    printf("\n");
16
17 }

```

List of memory dump locations:

- arr = {5, 10, 15, 20, 25}, arr = p, \*arr = arr[0] = 5
- 메모리 창 (Memory Window)
  - 디버그 → 창 → 메모리 → 메모리 1
  - 조사식 창이나 자동 창에서 포인터 변수를 눌러 메모리 창의 주소란으로 끌고 갑.
  - arr 배열 첫 요소인 5는 "05 00 00 00"으로 저장

메모리 창

The screenshot shows the Visual Studio debugger interface with three windows:

- 메모리 1**: Shows memory dump starting at address 0x002FFC80. The first few bytes are 19 00 00 00 cc cc cc cc cc cc ...??????.
- 자동**: Shows local variables:
 

이름	값	형식
arr	0x002ffc70 {5, 10, 15, 20, 25}	int[5]
i	4	int
p	0x002ffc70 {5}	int *
- 조사식 1**: Shows expression evaluation for p+i:
 

이름	값	형식
p+i	0x002ffc80 (25)	int *
*(p+i)	25	int

```

2 int main() {
3     int i, *p, arr[ ] = {5, 10,
4
5     printf("arr[0] is %d.\n", a
6     printf("*arr is %d.\n", *ar
7
8     p = arr;
9     for (i = 0; i < 5; i++)
10        printf("%d\t", *(p + i))
11    printf("\n");
12
13    p = arr;
14    for (i = 0; i < 5; i++)
15        printf("%d\t", *p++);
16    printf("\n");
17

```

### ● 메모리 창도 디버깅 도구

- for 루프 내부에서 i가 4인 상태
- 조사식 창에 (p + i)와 \*(p + i)를 입력하여 확인
- 메모리 창은 (p + i)에서 시작하는 메모리 내용

## 배열 요소의 합을 구하는 세 가지 방법

### ● Example 11-2 실습 및 해설

- ```
for (p = arr; p < &arr[MAX]; p++)
    sum += *p; // 포인터 산술 연산
```
- ```
for (i = 0; i < MAX; i++)
    sum += *(arr + i); // 배열 명(포인터) 산술 연산
```
- ```
p = arr;
for (i = 0; i < MAX; i++)
    sum += p[i]; // 포인터를 배열 명으로 사용
```

**example\_11-2.c :**

```

#define MAX 5
#include <stdio.h>

int main() {
    int i, sum, *p, arr[MAX];

    sum = 0;
    for (p = arr; p < &arr[MAX]; p++)
        sum += *p; // 포인터 산술 연산

```

```

printf("Sum is %d.\n", sum);

sum = 0;
for (i = 0; i < MAX; i++)
    sum += *(arr + i); // 배열명(포인터) 산술 연산
printf("Sum is %d.\n", sum);

sum = 0;
p = arr;
for (i = 0; i < MAX; i++)
    sum += p[i]; // 포인터를 배열명으로 사용
printf("Sum is %d.\n", sum);

return 0;
}

```

## 배열과 포인터

```

arr[i] = *(arr + i)

p = arr;
&p[i]
= &p[0] + sizeof(int) * i
= p + sizeof(int) * i
= arr + sizeof(int) * i // 포인터를 배열 명으로 사용 가능

```

```

int num, arr[4];
num = arr[2]; // O.K.
num = 2[arr]; // O.K. WHY? Translated to *(2 + arr)

```

- 배열 기호는 컴파일러에 의해 포인터 기호로 바뀐다.
  - 배열의 정체는 포인터
  - 어셈블리 언어에는 배열이 존재하지 않는다.
  - 배열 시작 주소(Base Address)와 이격 거리(Offset)만 존재

## 배열과 포인터

```
int num, *p = &num;
printf("*p is %d.\n", *p);           // 변수
printf("p[0] is %d.\n", p[0]);       // 변수 하나를 배열로 간주
int *a; // a가 가리키는 것이 단일 변수라는 의미가 강함(아닐 수도 있음)
int a[]; // a가 가리키는 것이 배열이라는 의미가 강함(아닐 수도 있음)
```

```
int *p, arr[MAX];
arr = p; arr = 100;                  // 오류. 배열 명은 상수 포인터
p = arr; p = arr + 2;               // O.K. p는 변수 포인터
```

- 배열 명 = 배열 시작 주소 = 상수

```
int arr[10];
int *p = arr + 9;
int num = *(++p);                  // 오류. arr[10]은 배열 바깥
```

- 배열 인덱스 초과 (Array Index Out Of Range) 오류
  - 컴파일 타임 오류가 아니라 런 타임 오류

## 난수 생성

- Example 11-3, 11-4, 11-5 실습 및 해설
- 프로그램은 입력이 같으면 출력도 같다.
  - `rand()`를 호출할 때마다 출력이 같다.
- 해결책
  - `srand(time(NULL))`를 호출하여 `random seed`를 변경
  - 한 번 호출로 첫 난수를 바꾸면 이어지는 난수도 바뀜
  - 난수 발생 알고리즘이 직전의 난수를 이용하여 그 다음 난수를 만들기 때문
- `rand`가 생성하는 난수의 범위는 0부터 `RAND_MAX`
  - `first ≤ rand() ≤ last` 범위로 사상 시키려면
  - `first + rand() % (last - first + 1)`를 이용

### example\_11-3.c :

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

```

int main() {
    int i;
    for (i = 0; i < 6; i++)
        printf("%d ", rand());
    printf("\n");

    return 0;
}

```

**16807 282475249 1622650073 984943658 1144108930 470211272**

- 여섯 개의 난수(Random Number)를 생성하는 프로그램이다. stdlib.h 파일에 선언된 rand 함수는 0부터 RAND\_MAX까지의 정수 중 임의의 숫자를 돌려준다. 여기서 RAND\_MAX는 stdlib.h에 정의된 상수로서 int형으로 표현할 수 있는 최대 정수를 말한다. 사용되는 stdlib.h에 따라 값이 달라지지만 이 값은 최소 32767 이상이다. 그런데 이상하게도 이 프로그램은 실행될 때마다 매번 동일한 숫자 열(Number Sequence)을 출력한다.
- 프로그램은 입력이 같으면 출력도 같다.
- 입력이 같은데도 어떨 때는 이렇게 출력하고 어떨 때는 저렇게 출력하는 프로그램은 존재하지 않는다. rand 함수도 마찬가지다. 입력이 같으면 출력도 같은 수밖에 없다. rand 함수에 입력되는 값을 seed라 부른다. 난수를 생성하는 씨앗이라는 뜻이다. 예를 실행할 때마다 매번 동일한 숫자 열을 내보내는 이유는 디폴트로 사용된 seed 값이 같기 때문이다. seed가 같으면 첫 난수가 같아질 뿐 아니라 숫자 열 전체가 같아진다. 대부분의 난수 발생 알고리즘이 바로 직전의 난수를 이용하여 그 다음 난수를 만들기 때문이다.
- `void srand(unsigned seed);`
- stdlib.h 파일에는 srand 함수가 선언되어 있다. 이는 rand 함수에 입력될 seed 값을 설정하는 함수로서 원형은 위와 같다. 만약 이 함수를 호출하지 않으면 디폴트 seed 값으로 1이 적용되어 srand(1)이 호출된다. 예를 실행할 때마다 매번 동일한 숫자 열이 나온 이유는 seed 값이 계속해서 1로 들어갔기 때문이다. 그렇다면 어떻게 하면 프로그램이 실행될 때마다 srand 함수를 써서 seed 값을 바꿀 수 있을 것인가가 관건이다.

#### **example\_11-4.c :**

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i;
    srand(time(NULL));
    for (i = 0; i < 6; i++)
        printf("%d ", (1 + rand() % 45));
    printf("\n");

    return 0;
}

```

```
9 21 22 11 9 27
```

- 이에 대한 해법이 time.h 헤더 파일에 선언된 time 함수다. 이 함수는 NULL을 인자로 주면서 time(NULL)이라고 하면 1970년 1월 1일 이후 지금까지 몇 초가 경과했는지를 정수로 돌려준다. 따라서 프로그램이 실행되는 매 순간마다 값이 바뀐다.
  - `first <= rand() <= last`
  - `first + rand() % (last - first + 1)`
- rand가 생성하는 난수의 범위는 0부터 RAND\_MAX까지다. 그런데 이 범위를 조절해야 할 때가 있다. 예를 들어, 로또 숫자는 1부터 45까지로 제한되어 있다. 그 경우 0부터 RAND\_MAX까지의 숫자를 1부터 45까지 숫자로 사상(Mapping)시켜야 한다.  
( $1 + \text{rand()} \% 45$ )이 그 예다. 어떤 수를 45로 나누면 그 나머지는  $0 \sim 44$ 에 범위에 분포한다. 따라서 여기에 1을 더하면  $1 \sim 45$  범위가 된다. 이를 일반화하면 예처럼 공식화할 수 있다. 여기서 first는 범위 내의 첫 숫자, last는 마지막 숫자를 의미한다. 만약 rand 함수의 결과를  $10 \sim 40$  범위로 사상하려면  $10 + \text{rand()} \% 31$ 로 해야 한다.

#### example\_11-5.c :

```
#define MAX 6
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h> // for sleep()

void print_array(const int *, int);
void swap(int *, int *);
void bubble_sort(int *, int);
int is_duplicate(const int *, int, int);
void set_numbers(int *);

void print_array(const int * arr, int length) {
    int i;
    for (i = 0; i < length; i++)
        printf("%d ", arr[i]);
}

void swap(int * p, int * q) {
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

void bubble_sort(int * arr, int length) {
    int pass, current, sorted = 0;
    for (pass = 1; (pass < length) && (!sorted); pass++) {
        sorted = 1;
        for (current = 0; current < (length - pass); current++) {
```

```

        if (arr[current] > arr[current + 1]) {
            swap(&arr[current], &arr[current + 1]);
            sorted = 0;
        }
    }
}

int main(void) {
    int money, i, lottery[MAX];
    while (1) {
        printf("Enter available money.\n");
        scanf("%d", &money);
        if (money == 0)
            break;
        printf("Possible sets of lottery numbers are,\n");
        for (i = 1; i <= money / 1000; i++) {
            set_numbers(lottery);
            bubble_sort(lottery, MAX);
            print_array(lottery, MAX);
            printf("\n");
        }
    }

    return 0;
}

int is_duplicate(const int * arr, int length, int n) {
    int i;
    for (i = 0; i < length; i++) {
        if (arr[i] == n)
            return 1;
    }
    return 0;
}

void set_numbers(int * lotto) {
    int i, num;
    sleep(10);
    srand(time(NULL));
    for (i = 0; i < MAX; ) {
        num = 1 + rand() % 45;
        if (!is_duplicate(lotto, i, num))
            lotto[i++] = num;
    }
}

```

```

Enter available money.
5000
Possible sets of lottery numbers are,
7 12 15 22 31 45
3 14 17 36 38 42
8 12 15 20 40 44
7 23 26 30 31 32
2 10 20 30 32 36
Enter available money.

```

## 11-2. 2중 포인터

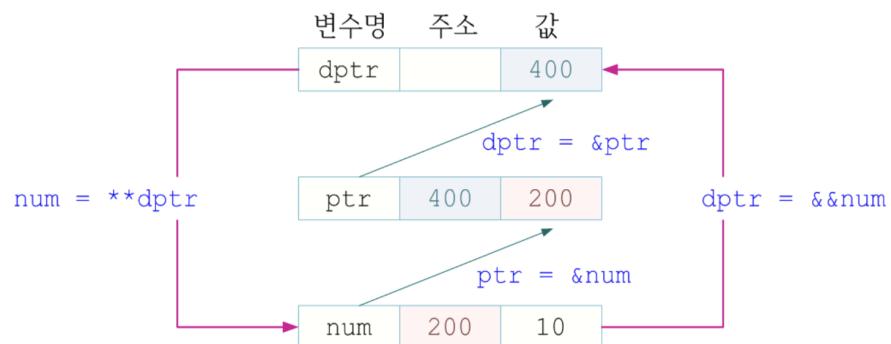
### 2중 포인터

- Example 11-6 실습 및 해설

- ```

int num = 10;
int *ptr = &num;
int **dptr = &ptr;

```



- Double Pointer

- Pointer to a Pointer = 포인터를 가리키는 포인터

#### example\_11-6.c :

```

#include <stdio.h>

int main() {
    int num = 10;
    int * ptr = &num;
    int ** dptr = &ptr;

    printf("num is %d.\n", num);
    printf("*ptr is %d.\n", *ptr);
    printf("**dptr is %d.\n", **dptr);
}

```

```

    return 0;
}

```

```

num is 10.
*ptr is 10.
**dptr is 10.

```

## 2중 포인터에 의한 스왑

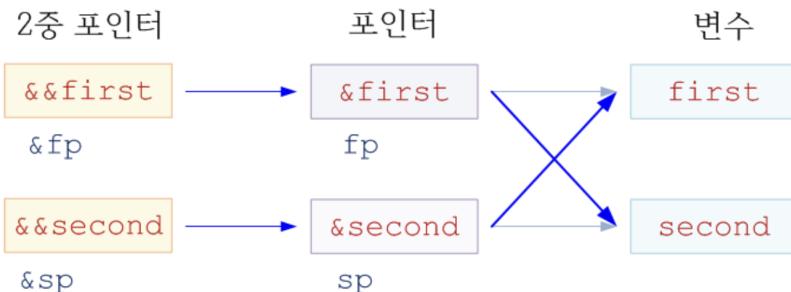
### Example 11-7 실습 및 해설

- ```

int first, second; int *fp = &first, *sp = &second;
swap(&fp, &sp);

• void swap(int** p, int** q){ // 인자가 2중 포인터
    int *temp; // 포인터를 스왑
    temp = *p;
    *p = *q;
    *q = temp;
}

```



### example\_11-7.c :

```

#include <stdio.h>

void swap(int **, int **);

int main() {
    int first, second;
    int * fp = &first, * sp = &second;

    printf("Enter two integers to swap.\n");
    scanf("%d%d", &first, &second);
    printf("Before swap : *fp is %d and *sp is %d\n", *fp, *sp);
    swap(&fp, &sp);
    printf("After swap : *fp is %d and *sp is %d\n", *fp, *sp);

    return 0;
}

```

```

}

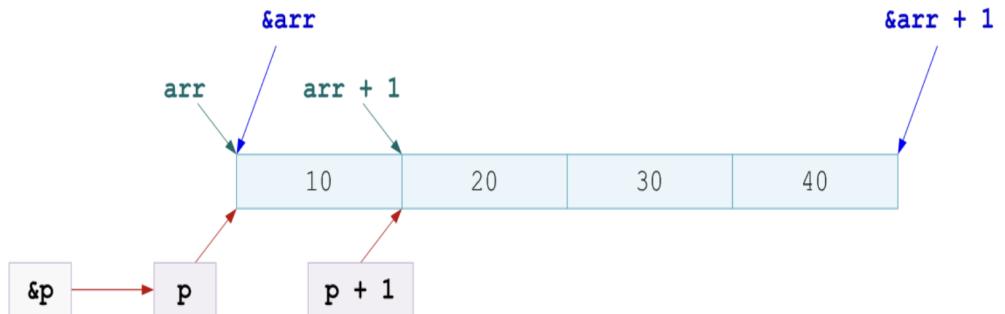
void swap(int ** p, int ** q) {
    int * temp;
    temp = *p;
    *p = *q;
    *q = temp;
}

```

## 배열 명, 배열의 주소, 포인터 변수

- Example 11-8 실습 및 해설

- `int *p, arr[4] = { 10, 20, 30, 40 };`



- 포인터 산술 연산의 단위
  - `arr`은 배열 첫 요소의 시작 주소
  - `&arr`은 배열 전체의 시작 주소
  - `p`는 단순히 정수 포인터

### example\_11-8.c :

```

#include <stdio.h>

int main () {
    int * p, arr[4] = {10, 20, 30, 40};

    printf("arr is %p.\n", arr);
    printf("arr + 1 is %p.\n", arr + 1);
    printf("sizeof(*arr) is %d.\n\n", sizeof(*arr));

    printf("&arr is %p.\n", &arr);
    printf("&arr + 1 is %p.\n", &arr + 1);
    printf("sizeof(*(&arr)) is %d.\n\n", sizeof(*(&arr)));

    p = arr;
}

```

```

printf("p is %p.\n", p);
printf("p + 1 is %p.\n", p + 1);
printf("&p is %p.\n", &p); // &p는 p의 주소
printf("&p + 1 is %p.\n", &p + 1); // &p+1은 p의 주소 + 1, 포인터의 크기
는 8바이트

return 0;
}

```

```

arr is 0x16ae06f40.
arr + 1 is 0x16ae06f44.
sizeof(*arr) is 4.

&arr is 0x16ae06f40.
&arr + 1 is 0x16ae06f50.
sizeof(*(&arr)) is 16.

p is 0x16ae06f40.
p + 1 is 0x16ae06f44.
&p is 0x16ae06f30.
&p + 1 is 0x16ae06f38.

```

### 11-3. 동적 메모리

#### 지역 변수의 주소 반환

- Example 11-9 실습 및 해설

```

• int* find_bigger(const int a, const int b){
    int bigger;
    bigger = (a > b) ? a: b;
    return &bigger;
}

```

**출력**

출력 보기 선택(S): 빌드

1>----- 빌드 시작: 프로젝트: project1, 구성: Debug Win32 -----  
1>test1.c  
1>c:\sample\sample\project1\#test1.c(6): warning C4172: 지역 변수 또는 임시: bigger의 주소를 반환하고 있습니다.

warning: function returns address of local variable

- 지역 변수의 주소를 리턴해서는 안 된다.
  - bigger는 지역 변수. 함수 실행이 끝나면 사라짐
  - 사라질 변수의 주소를 main에게 리턴하고 있음

**example\_11-9.c :**

```

#include <stdio.h>

int * find_bigger(const int a, const int b) {
    int bigger;
    bigger = (a > b) ? a : b;
    return &bigger;
}

int main () {
    int first, second, max;

    printf("Enter two integers to compare.\n");
    scanf("%d%d", &first, &second);

    max = *(find_bigger(first, second));
    printf("The bigger one is, %d.\n", max);

    return 0;
}

```

```

ch11.c:243:13: warning: address of stack memory associated with local variable 'bigger' returned [-Wreturn-stack-address]
    return &bigger;
           ^~~~~~
1 warning generated.

```

## 동적 변수

- Example 11-10 실습 및 해설
  - **사라질 배열**의 시작 주소를 리턴하고 있음
  - 해결 방법
    - 정적 변수 (Static Variable): static int arr[5];
    - 전역 변수 (Global Variable): 함수 외부에 int arr[5];
    - **동적 변수 (Dynamic Variables)**: 힙 메모리에 변수를 만듦

### example\_11-10.c :

```

#include <stdio.h>

int * num_array() {
    int i, arr[5];
    for (i = 0; i < 5; i++)
        arr[i] = i;
    return arr;
}

```

```

int main() {
    int i, *p = num_array();

    for (i = 0; i < 5; i++)
        printf("%d ", p[i]);
    printf("\n");

    return 0;
}

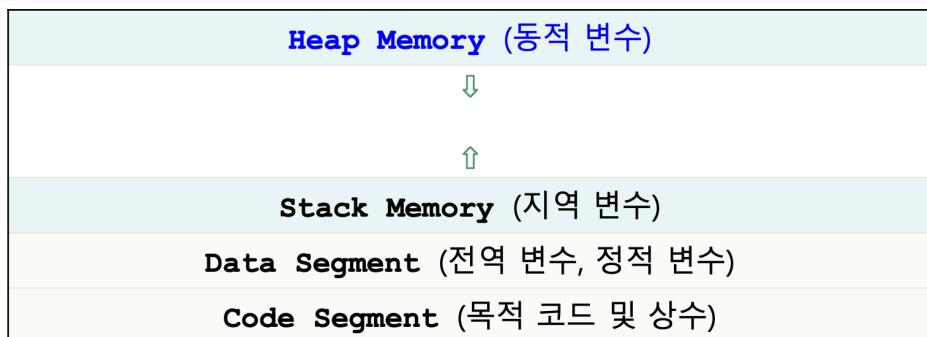
```

```

ch11.c:267:9: warning: address of stack memory associated with local variable 'arr' returned [-Wreturn-stack-address]
    return arr;
          ^
1 warning generated.
(base) minsung@iminsseong-ui-MacBookAir ~/Documents/3학년_2학기/시스템프로그래밍기초/dev/ch11 (main*?) $ ./ch11
0 -2004344924 -1619722239 3 4

```

## 동적 변수와 힙 메모리



- 스택 메모리: 정적 메모리 (Static Memory)
  - 컴파일 타임에 크기 확정 (지역 변수의 크기를 참고로 스택 프레임의 크기를 확정)
  - 생성과 소멸이 자동으로 관리됨
- 힙 메모리: 동적 메모리 (Dynamic Memory)
  - 런 타임에 크기 확정
  - 생성과 소멸을 프로그래머가 관리
    - malloc으로 만은 후, free하기 전까지는 그대로 유지됨

## 동적 변수 만들기

```
#include <stdlib.h>
(void *)malloc(size_t size);
#define size_t unsigned int
```

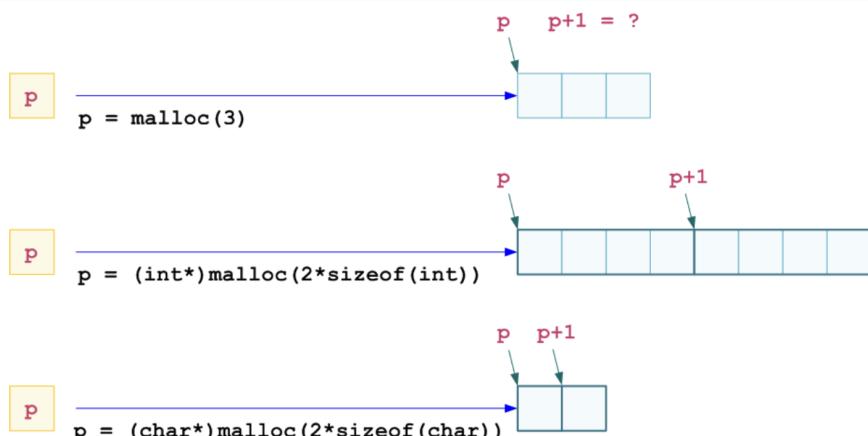
- malloc(memory allocate) 함수
  - size 크기의 힙 메모리를 할당한 후 시작 주소를 리턴
  - size\_t는 변수나 자료형의 **크기를 표현**하는 자료형
  - void\*는 **범용 포인터**(이후 어떤 자료형이라도 가리킬 수 있음)

```
int* p;
*p = 3;           // Error

int num,
int *p = &num;
*p = 3;           // O.K.

int* p;
p = (int*)malloc(sizeof(int));    // (int*)는 형 변환 연산자
*p = 3;           // O.K.
```

## malloc 함수의 형 변환



- 가리키는 대상을 **몇 바이트 단위로 읽을지를** 천명

```
int num = 10;
void *p = &num;
p++;           // Error
printf("%d", *p);        // Error
printf("%d", *(int*)p);   // O.K.
```

● Example 11-11 실습 및 해설

```
int* get_number() {
    int* p;
    p = (int *)malloc(sizeof(int));
    if (p == NULL) {
        printf("No more memory available.\n");
        exit(1);
    }
    *p = 20;
    return p;
}
```

- 가용한 힙 메모리가 없으면 malloc은 NULL을 리턴
  - NULL Test에 의해 오류 처리
  - exit로 프로그램을 빠져 나옴

**example\_11-11.c :**

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

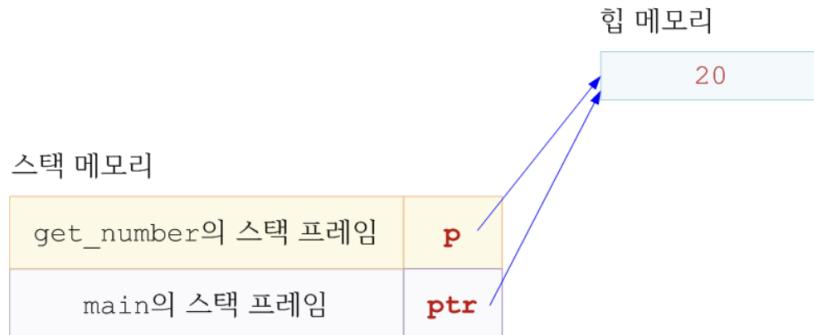
int * get_number() {
    int * p;
    p = (int *)malloc(sizeof(int));
    if (p != NULL)
        *p = 20;
    return p;
}

int main() {
    int i, *ptr;
    ptr = get_number();
    printf("ptr is %d.\n", *ptr);
    free(ptr);
    ptr = NULL;
}
```

**ptr is 20.**

## p versus \*p

- ```
int* get_number() {
    int* p;
    p = (int *)malloc(sizeof(int));
```



- `p`는 스택 프레임에 존재하는 지역 변수
- `*p`는 힙 메모리에 존재하는 동적 변수
  - 힙 메모리에는 변수 명이 없음
  - 변수 명 대신 포인터로 접근
- 힙 메모리는 변수명 대신 포인터로 접근한다.
- 예에서 유의해야 할 점은 두 가지다.
  - 첫째, `malloc`에 의해 확보된 변수 공간에는 별도의 변수명이 없다. 변수 공간을 가리키는 포인터를 따라가서 20이라는 값을 넣었을 뿐이다. `malloc`에 의해 만들어지는 공간은 기본적으로 프로그램 실행 도중 확보한 공간이다. 따라서 프로그램 실행 도중에 변수명을 부여할 수는 없는 노릇이다.
  - 둘째, 포인터 `p`가 가리키는 변수 공간은 스택 메모리가 아니라 힙 메모리에 만들어진다. 따라서 이 공간은 `get_number` 함수를 빠져나와 호출 함수로 되돌아가도 그대로 유지된다.
- `p = (int*)malloc(sizeof(int));`에서 포인터 변수 `p`는 여전히 `get_number` 함수의 스택 프레임에 존재하는 지역 변수라는 점에도 유의해야 한다. 즉, `int * p;`에 의해 만들어진 `p` 자체는 `get_number` 함수의 스택 프레임에 존재하기 때문에 함수를 빠져나오면 사라진다. 그럼에도 불구하고 `main`에서 `p`를 따라갈 수 있는 이유는 `return p;`에서 리턴한 `p`의 값이 복사되어 `ptr = get_number`의 `ptr`로 들어갔기 때문이다. **힙 메모리에 만들어지는 것은 p가 아니라 p가 가리키는 변수다.** 그 변수 공간이 사라지지 않았고, 또 그 변수 공간을 가리키는 포인터가 있으면 얼마든지 접근이 가능하다.
- `malloc`으로 할당받은 메모리는 `free`를 가하기 전까지는 그대로 살아 있다.

## 1. 힙 메모리는 변수 이름이 없고, 오직 포인터로 접근한다.

- `malloc`에 의해 할당된 메모리는 특정 주소를 가리키는 포인터(`p`)로만 접근할 수 있습니다.
- 따라서, 힙 메모리에 접근하려면 반드시 그 메모리를 가리키는 포인터가 있어야 합니다.

## 2. 스택 메모리와 힙 메모리의 차이를 이해해야 한다.

- `get_number` 함수가 종료되면, 스택 메모리에 있던 변수(`p`)는 사라집니다.
- 하지만, `p` 가 가리키던 힙 메모리 공간은 여전히 존재합니다.
- 따라서, `get_number`에서 힙 메모리 주소를 반환(return)하면, 메인 함수에서도 해당 메모리에 접근 가능합니다.

### free 함수

```
void free(void *ptr);
```

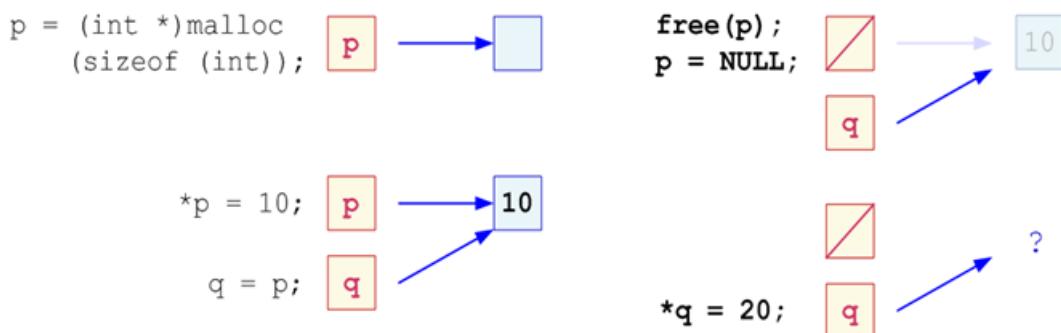
- `malloc`으로 할당 받은 메모리를 반납
- **포인터를 인자로 넘겨줌**
- `free`를 가하기 전까지는 그대로 살아 있음
- cf. Java 언어는 가비지 콜렉터 (Garbage Collector)가 자동으로 불필요한 메모리를 수거

```
free(ptr);
ptr = NULL;           ①
...
if (ptr != NULL)     ②
    *ptr = 40;
```

- `free` 직후에는 항상 `NULL`을 대입
- 이후에, 이미 `free` 시킨 포인터를 따라가지 않도록 **널 테스트**

## 댕글링 포인터

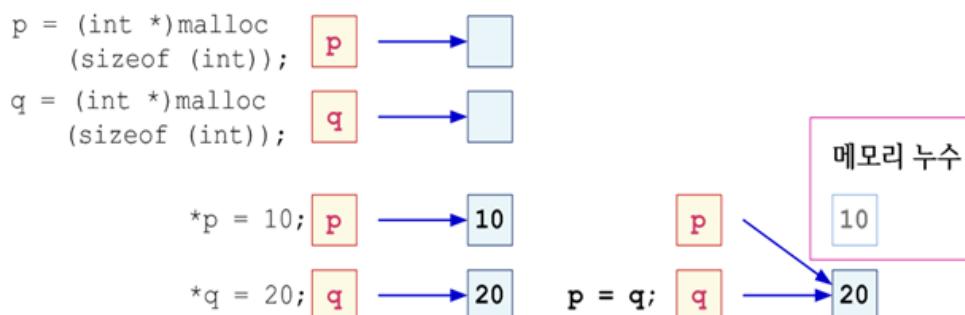
```
int *p, *q;  
p = (int*)malloc(sizeof(int));  
*p = 10;  
  
q = p;  
free(p);  
p = NULL;  
*q = 20; // Error. q is a dangling pointer.
```



- 동적 메모리를 반납할 때는 이른바 댕글링 포인터에 유의해야 한다.  
 $p = q;$ 에 의해  $q$ 도  $p$ 와 동일한 메모리 공간을 가리키게 된다.  
이후  $free(p);$ 에서  $p$ 가 가리키는 공간을 반납하고  $p$ 를  $NULL$ 로 만들었다고 가정하자. 그러나  $q$ 는 여전히 이미 반납한 이전 메모리 공간을 가리키고 있다. 결국  $*q = 20;$ 처럼  $q$ 를 따라가는 것은 오류다. 이처럼 이미 반납한 공간을 가리키는 포인터를 댕글링 포인터라고 한다.

## 메모리 누수

```
int* p = (int *)malloc(sizeof(int));  
int* q = (int *)malloc(sizeof(int));  
*p = 10; *q = 20;  
p = q;
```



### • Memory Leakage

- $p = q;$   $p$ 가 가리키던 공간을 반납하지도 않았고, 접근도 불가
- 낙동강 □□□

● Example 11-12 실습 및 해설

- 동적 메모리 활용은 프로그래머의 자유

- `int* p = (int*)malloc(40); // 10개의 정수로 활용`
- `char* p = (char*)malloc(40); // 40개의 문자로 활용`

```
int name[40];
char *name = (char*)malloc(sizeof(char) * sizeof(char));
```

- 정적 배열 크기는 컴파일 타임에 확정
  - 스택 프레임의 크기를 미리 확정하기 위함
  - 메모리 낭비 우려
- 동적 배열 크기는 런 타임에 확정
  - 메모리 낭비 방지
  - 정적 배열에 비해 메모리 할당 속도는 느림(힙 메모리 검색)

example\_11-12.c :

```
#define MAX 10
#include <stdio.h>
#include <stdlib.h>

int * square_array() {
    int i;
    int * p = NULL;

    p = (int*)malloc(MAX * sizeof(int));
    if (p != NULL) {
        for (i = 0; i < MAX; i++)
            p[i] = i * i;
    }
    return p;
}

int main() {
    int i, * arr = square_array();
    for (i = 0; i < MAX; i++)
        printf("%d ", *(arr + i));
    printf("\n");
    free(arr);
    arr = NULL;
```

```

    return 0;
}

```

0 1 4 9 16 25 36 49 64 81

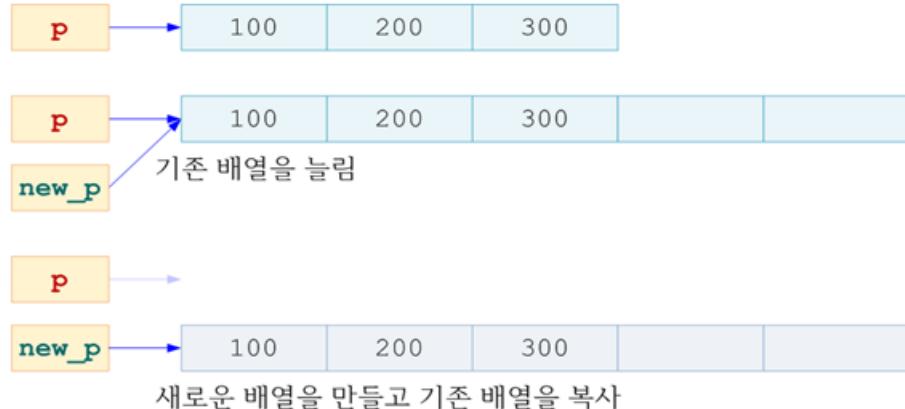
### calloc, realloc

```
int *p = (int*)calloc(3 * sizeof(int));
```

- malloc과 동일. 단, 메모리를 0으로 초기화

```
int *p = (int *)malloc(3 * sizeof(int));
```

```
int *new_p = (int *)realloc(p, 5 * sizeof(int));
```



- 공간이 없을 경우 이전 포인터 p가 가리키는 공간을 반납
- 정적 배열에 대해서는 realloc 사용 불가

- realloc(re-allocate) 함수는 문자 그대로 다시 할당하라는 것이다. `int * p = (int *)malloc(3 * sizeof(int));`에서 malloc 함수로 정수 세 개를 넣을 공간을 할당했는데 프로그램 실행 도중에 이 공간으로는 부족해서 추가로 두 개를 더 넣을 공간이 필요할 때도 있다.  
그 경우 `int * new_p = (int *)realloc(p, 5 * sizeof(int));`;처럼 realloc 함수를 써서 모두 다섯 개의 공간을 만들어달라고 요구할 수 있다. 단, 이전에 사용하던 공간을 가리키는 포인터 p를 첫 인자로 넘겨주어야 한다. 만약 p를 NULL로 주면 이전에 사용하던 공간이 없다는 뜻이므로 그 경우 realloc은 malloc과 동일한 명령이 된다. 따라서 루프를 돌 경우에 p = NULL로 초기화한 후 루프 내부에서 realloc을 쓰면 처음에는 malloc으로 먹고 이후로는 realloc으로 먹는다.
- realloc은 일단 현재 포인터 p가 가리키는 공간 끝에 정수 두 개를 추가할 만한 빈 공간이 있는지를 확인한다. 만약 그러한 공간이 있으면 공간을 넓혀 원래의 p를 그대로 돌려준다.  
그러나 만약 그 공간을 이미 다른 프로그램이 사용하고 있으면 그렇게 할 수가 없다. 그 경우 realloc은 다른 빈 곳에 한꺼번에 다섯 개를 넣을 공간이 있는지를 확인한다. 이후 원래 p가 가리키던 정수 세 개를 그대로 복사한 다음에 새로 마련한 공간의 시작 주소를 돌려준다.
- realloc 함수를 사용할 때 유의할 점은 두 가지다.
  - 첫째, 충분한 인접 공간이 없어 다른 곳으로 옮겨갈 때의 realloc은 이전 포인터 p가 가리키던 공간을 운영체제에 반납한다. 따라서 이전 포인터인 p를 따라가면 안 된다. 물론 인접 공간이 충분하다면 그렇게 하지 않는다. 그러나 프로그래머로서는 그러한 속성을 알 수가 없다. 따라서 예에서 보듯이 realloc에 의해 새로운 포인터를 할당받은 이후에는 이전에 사용하던 포인터를 쓰지 말아야 한다.

- 둘째, 정적 배열에 대해서는 realloc 함수를 쓸 수 없다. 정적 배열은 힙이 아니라 스택에 존재하기 때문이다. 예를 들어 `int arr[3]; int * p = arr;`이라고 해놓고 `p`가 가리키는 배열을 `realloc`으로 늘리려고 하면 안 된다.

## realloc 함수 사용 예시

- Example 11-13, 11-14 실습 및 해설
  - realloc 함수 사용

**example\_11-13.c :**

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

int main() {
    int* p = (int*)malloc(3 * sizeof(int));
    int i;

    p[0] = 100; p[1] = 200; p[2] = 300;
    p = (int*)realloc(p, 5 * sizeof(int));
    if (p != NULL) {
        p[3] = 400;
        p[4] = 500;
    }
    for (i = 0; i < 5; i++)
        printf("%d ", *(p + i));
    printf("\n");

    return 0;
}
```

100 200 300 400 500

**example\_11-14.c :**

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

int main() {
    int last = 0; char ch;
    char * p;

    p = (char*)malloc(sizeof(char));
    if (p == NULL)
        exit(1);
```

```

printf("Enter a character string.\n");
while (1) {
    if ((ch = getchar()) == '\n')
        break;
    else {
        *(p + last) = ch;
        last++;
        p = (char*)realloc(p, (last + 1) * sizeof(char));
        if (p == NULL)
            exit(1);
    }
}
*(p + last) = '\0';
printf("%s\n", p);

return 0;
}

```

```

Enter a character string.
The beauty of programming.
The beauty of programming.

```

#### 11-4. 2차원 배열과 포인터 배열

##### 2차원 배열

```

int arr[2][3] = {
{1, 2, 3},           // 0 행
{4, 5, 6}           // 1 행
};
int arr[2][3] = {1, 2, 3, 4, 5, 6};

```

arr[0]			arr[1]		
arr[0][0]	arr[0][1]	arr[0][2]	arr[1][0]	arr[1][1]	arr[1][2]
1	2	3	4	5	6

- 2차원 배열의 정체는 1차원 배열
  - 관점의 차이 (행과 열로 간주)
  - 배열의 배열 (Array of Array).
  - arr[0] 행이 여러 열로 이루어진 배열

##### ● Example 11-15 실습 및 해설

example\_11-15.c :

```

#define ROW 2 // 행(Row)의 수
#define COL 3 // 열(Column)의 수
#include <stdio.h>

int main() {
    int i, j;
    int first[ROW][COL], second[ROW][COL], add[ROW][COL];

    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COL; j++) {
            printf("Enter first matrix [%d][%d] :", i, j);
            scanf("%d", &first[i][j]);
        }
    }

    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COL; j++) {
            printf("Enter second matrix [%d][%d] :", i, j);
            scanf("%d", &second[i][j]);
        }
    }

    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COL; j++)
            add[i][j] = first[i][j] + second[i][j];
    }

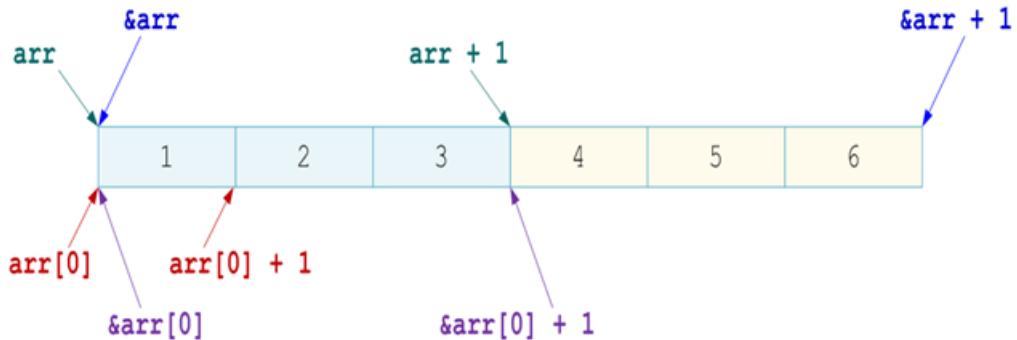
    printf("Sum of the two matrices is,\n");
    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COL; j++)
            printf("%d ", add[i][j]);
        printf("\n");
    }

    return 0;
}

```

## 2차원 배열의 포인터 산술 연산

### ● Example 11-16 실습 및 해설



- `arr[0]`도 그 자체로 배열 명. 행의 첫 요소를 가리킴
- `&arr[0]`은 행 전체를 가리킴
- `&arr`은 2차원 배열 전체를 가리킴

example\_11-16.c :

```
#include <stdio.h>

int main() {
    int * p, * q, arr[2][3] = { 1, 2, 3, 4, 5, 6 };

    printf("arr is %p.\n", arr);
    printf("arr + 1 is %p.\n", arr + 1);
    printf("&arr is %p.\n", &arr);
    printf("&arr + 1 is %p.\n", &arr + 1);

    printf("arr[0] is %p.\n", arr[0]);
    printf("arr[0] + 1 is %p.\n", arr[0] + 1);
    printf("&arr[0] is %p.\n", &arr[0]);
    printf("&arr[0] + 1 is %p.\n", &arr[0] + 1);

    p = arr;
    printf("p is %p, p + 1 is %p.\n", p, p + 1);

    q = arr[0];
    printf("q is %p, q + 1 is %p.\n", q, q + 1);

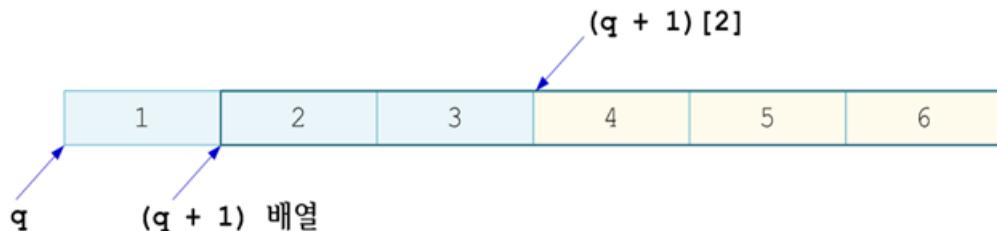
    printf("(q+1)[2] is %d.\n", (q+1)[2]);
}
```

```
    return 0;  
}
```

```
arr is 00000078968FFB98.  
arr + 1 is 00000078968FFBA4.  
&arr is 00000078968FFB98.  
&arr + 1 is 00000078968FFBB0.  
arr[0] is 00000078968FFB98.  
arr[0] + 1 is 00000078968FFB9C.  
&arr[0] is 00000078968FFB98.  
&arr[0] + 1 is 00000078968FFBA4.  
p is 00000078968FFB98, p + 1 is 00000078968FFB9C.  
q is 00000078968FFB98, q + 1 is 00000078968FFB9C.  
(q+1)[2] is 4.
```

## 2차원 배열과 포인터

- `int *p, *q, arr[2][3] = { 1, 2, 3, 4, 5, 6 };`
- `p = arr;`
- `q = arr[0];`

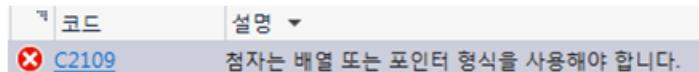


- `p, q`는 정수 포인터
  - 항상 4 바이트 단위로 움직임
  - `(q + 1)[2]`는 `q + 1` 배열의 세번째 요소

## 2차원 배열과 포인터

### ● Example 11-17 실습 및 해설

- ```
int i, j, arr[2][3];
int *p = arr;
p[i][j] = i * i + j * j; // 오류
```



- 2차원 배열을 포인터로 받으면 행과 열의 수에 관한 정보가 없음

```
p[i][j] = *(p + (i * COL) + j)
```

- 한 행에 몇 열이 있는지를 알아야 컴파일러가 p[i][j]의 주소 계산 가능

#### example\_11-17.c :

```
#include <stdio.h>

int main() {
    int i, j, arr[2][3];
    int* p = arr;

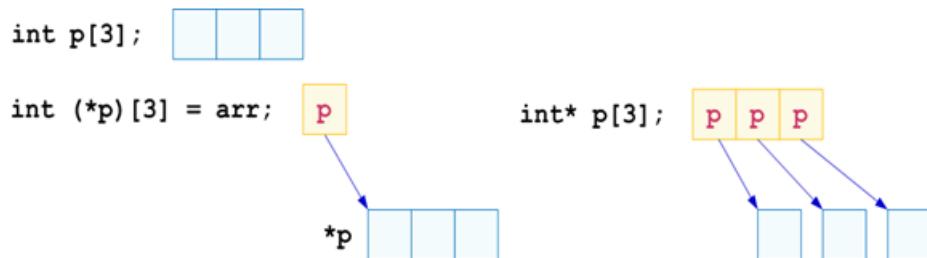
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            p[i][j] = i * i + j * j; // 컴파일 오류
            printf("%d ", p[i][j]); // 컴파일 오류
        }
        printf("\n");
    }

    return 0;
}
```

- 2차원 배열에서도 배열과 포인터가 자유로이 호환되는지에 대해 생각해 보자. 2차원 배열에서 p[i][j]의 주소는  $p[i][j] = *(p[0] + (i * \text{COL}) + j)$ 처럼 계산할 수 있다. 여기서 COL은 한 행이 몇 열로 구성되어 있는 가를 말한다. 예를 들어 p[1][2]는  $*(p[0] + (1 * 3) + 2)$ 로 나타낼 수 있다. 하나의 행이 3열로 구성되므로 행의 수인 1에 3을 곱한 다음에 거기에 열의 수인 2를 더해야 비로소 p[1][2]의 주소를 구할 수 있기 때문이다. 물론 이는 컴파일러에 의해  $*(p[0] + (1 * 3) + 2) * \text{sizeof(int)}$ 처럼 자료형을 감안한 구체적인 주소로 바뀐다.

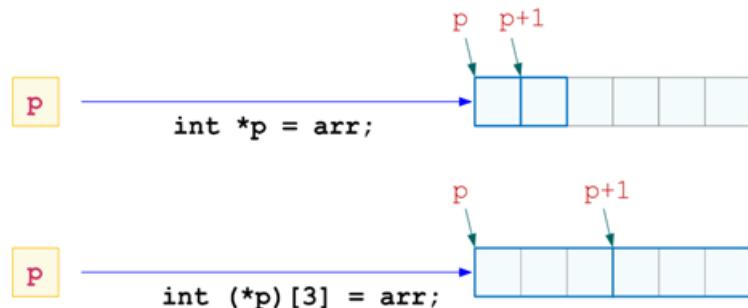
- 2차원 배열을 포인터로 받으면 거기에는 몇 행 몇 열이라는 정보가 없다.
- 2차원 배열을 포인터로 처리할 때는 행과 열에 관한 정보에 유의해야 한다. `int* p = arr;`에 의해 `arr` 배열의 시작 주소가 `p`로 들어간다. 그러나 `p`에는 단순히 시작 주소만 들어갈 뿐 그것이 가리키는 `arr` 배열의 차원이 몇 행 몇 열인지 모른다. 그런데 `p[i][j] = i * i + j * j;`, `printf("%d ", p[i][j]);`에서 `p[i][j]`의 주소를 계산하려면 한 행에 3열이 있다는 사실을 알고 있어야 비로소 `(p + (i * 3) + j)`라는 공식을 적용할 수 있다. 그러나 `p`에는 그런 정보가 없다. 결국 컴파일 오류가 뜬다. 물론 `p` 대신 `arr`을 직접 쓰면 이런 문제가 생기지 않는다. 또, `p[i][j]` 대신 `*(p[0] + (i * 3) + j)`라고 프로그래머가 직접 주소를 계산해주어도 문제가 생기지 않는다. 그러나 프로그램이 복잡해지면 그렇게 하기가 어려워진다.

## 2차원 배열과 포인터



- `int p[ ]`
  - 배열 `p`의 요소가 정수
- `int* p[ ]`
  - 배열 `p`의 요소가 포인터
- `int (*p)[ ]`
  - 포인터 `p`가 가리키는 것이 배열

## 2차원 배열과 포인터



- 해결책 1
  - `int (*p)[3]`에 의해 한 행이 3 열이라는 정보 제공

### ● Example 11-18 실습 및 해설

- 해결책 2
  - 매개변수 선언에 의해 행이 몇 열이라는 정보 제공

**example\_11-18.c :**

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

void print_it(int p[2][3]) {
    int i, j;
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            p[i][j] = i * i + j * j;
            printf("%d ", p[i][j]);
        }
        printf("\n");
    }
}

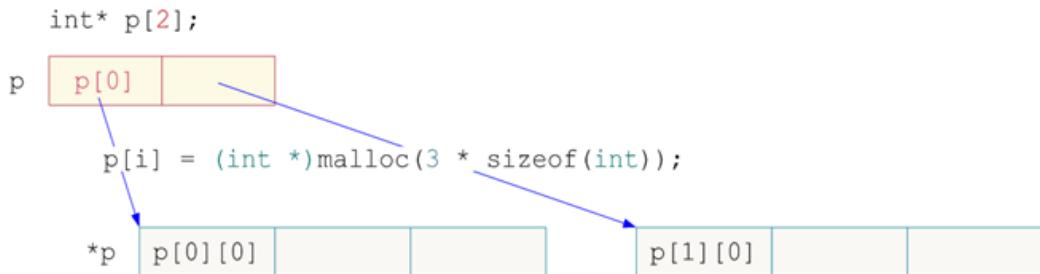
int main() {
    int* arr = (int*)malloc(2 * 3 * sizeof(int));
    if (arr != NULL)
        print_it(arr);
    free(arr);
    arr = NULL;

    return 0;
}
```

|   |   |   |
|---|---|---|
| 0 | 1 | 4 |
| 1 | 2 | 5 |

## 포인터 배열

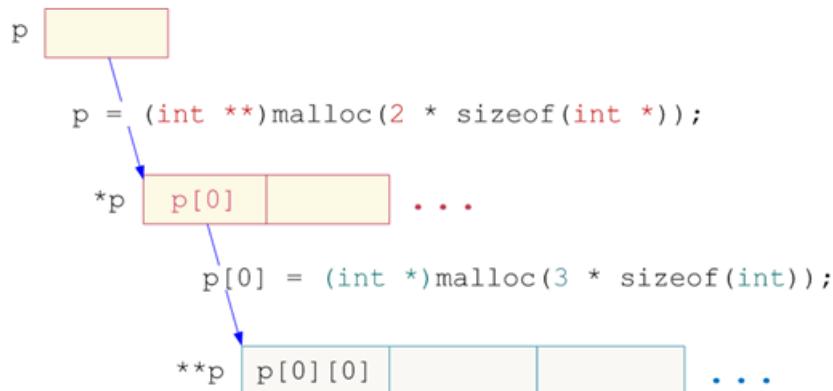
```
int* p[2];
for (i = 0; i < 2; i++)
    p[i] = (int*)malloc(3 * sizeof(int));
```



- 배열 요소가 포인터인 배열 (Array of Pointers)
  - 예의 경우 행은 정적 메모리, 열은 동적 메모리

## 2차원 포인터 배열

```
int** p = (int**)malloc(2 * sizeof(int*));
for (i = 0; i < 2; i++)
    p[i] = (int*)malloc(3 * sizeof(int));
```



- 행과 열이 모두 동적 메모리
  - 2차원 배열로 선언
  - 행은 포인터이므로 `sizeof(int*)` 크기
  - 행이 가리키는 것이 가리키는 것이 정수이므로 `int **` 타입

## 정적 2차원 배열과 동적 2차원 배열

```
int arr[2][3];
arr = &arr[0]           // arr은 첫 행의 시작 주소
arr[0] = &arr[0][0]     // arr[0]은 첫 열의 시작 주소
arr = &&arr[0][0]       // 따라서 배열 명은 첫 열의 주소의 주소
```

- 정적 2차원 배열 명도 2차원 포인터

```
int arr[m][n];
int **arr = (int**)malloc(m * n * sizeof(int));
```

- 정적 2차원 배열 선언 versus 동적 2차원 배열 선언

- 정적 배열이든 동적 배열이든 2차원 배열명은 2중 포인터다.
- 정적 배열인 경우에도 2차원 배열명은 사실상 2중 포인터다. 예에서 arr이라는 배열명은 배열의 시작 주소다. 그러나 2차원 배열의 경우 arr이 가리키는 것은 arr[0], 즉 0행 전체이므로, arr = &arr[0]처럼 쓸 수 있다.  
그런데 arr[0] 역시 배열로 이루어져 있기 때문에 arr[0]이 가리키는 것은 arr[0][0]이므로 arr[0] = &arr[0][0]처럼 쓸 수 있다.  
두 가지 사실을 종합하면 arr이 가리키는 것은 0행이고 0행이 가리키는 것은 0행 0열이다라고 할 수 있다.  
이는 arr = &arr[0]과 arr[0] = &arr[0][0]으로부터 arr = &&arr[0][0]을 유도한 것과 같다. 결국 arr[0][0]의 주소의 주소가 arr이라는 점에서 2차원 배열명은 2중 포인터라 할 수 있다.

## 정적 포인터 배열, 동적 포인터 배열

### ● Example 11-19 실습 및 해설

- 동적 2차원 행렬
- 정적 포인터 배열
  - int \*p[100];로 선언
  - 행의 수가 컴파일 타임에 결정
  - 열의 수는 런 타임에 결정
- 동적 포인터 배열
  - int \*\*p;로 선언
  - 행의 수, 열의 수 모두 런 타임에 결정

example\_11-19.c :

```

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

int ** create_matrix(int, int);
void destroy_matrix(int**, int);

int main() {
    int row, column, i, j, **matrix;

    printf("Enter row and column.\n");
    scanf("%d%d", &row, &column);

    matrix = create_matrix(row, column);
    for (i = 0; i < row; i++) {
        for (j = 0; j < column; j++) {
            matrix[i][j] = i * i + j * j;
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
    destroy_matrix(matrix, row);

    return 0;
}

int ** create_matrix(int row, int col) {
    int i, **p;
    p = (int**)malloc(row * sizeof(int *));
    if (p == NULL)
        exit(1);
    for (i = 0; i < row; i++)
        p[i] = (int*)malloc(col * sizeof(int));
    if (p[i] == NULL)
        exit(1);
}
return p;
}

void destroy_matrix(int** p, int row) {
    int i;
    for (i = 0; i < row; i++)
        free(p[i]);
    free(p);
    p = NULL;
}

```

```

Enter row and column.
4 5
0      1      4      9      16
1      2      5      10     17
4      5      8      13     20
9      10     13     18     25

```

▼ 11/28 목 - 12장

## 12장. 문자열

### 12-1. 문자열 표현

#### 문자열

```
char *str;
```

- 첫 문자를 가리키는 포인터
- 이후 문자는 포인터 산술 연산에 의해 접근 가능

```

char str[10] = "apple";           // 문자열
char str[10] = {'a', 'p', 'p', 'l', 'e'}; // 문자열이 아님

```



- 큰 따옴표로 둘러싸면 문자열로 간주
- 컴파일러가 자동으로 '\0'을 추가
- '\0' = 널 문자 = 문자열 끝을 나타내는 구분자 = 아스키 값 0

## 널 문자, 널 포인터

| 명칭             | 아스키 값 | 자료형          |
|----------------|-------|--------------|
| Null 문자 ('\0') | 0     | char         |
| NULL 포인터       | None  | int or void* |
| Zero 문자 ('0')  | 48    | char         |
| 줄 바꿈 문자 ('\n') | 10    | char         |
| Space 문자 (' ') | 32    | char         |

- 널 문자, 널 포인터 모두가 0
- 널 문자는 1 바이트 문자. 널 포인터는 4 바이트 정수 또는 포인터

```
char *p = "apple";
```

- "apple"의 값  
= 첫 문자 'a'를 가리키는 포인터  
= 문자열 시작 주소

## 문자열 길이, 문자열 크기

### ● Example 12-1, 12-2 실습 및 해설

- char str[20];  
`scanf("%s", str); printf("%s", str);`
  - %s 형식으로 읽으면 자동으로 '\0'을 추가
  - scanf, printf가 문자열을 다룰 때에는 포인터를 인자로 전달
- "apple"
  - 문자열 길이(Length)는 5
    - strlen은 문자열 길이를 구하는 함수
  - 문자열 크기(Size)는 6.
    - 크기를 따질 때에는 '\0'까지 포함

### ● Example 12-3 실습 및 해설

example\_12-1.c :

```
#include <stdio.h>

int main() {
```

```

char* p = "apple";
char str[20];
int i;

printf("*p is %c.\n", *p);
printf("Last character of apple is %c.\n", *(("apple" + 4)));
// "apple"은 'a'를 가리키는 포인터를 돌려줌.
// 이에 4를 더하고 있으므로 이는 포인터 산술 연산임.

printf("Enter a string.\n");
scanf("%s", str);

printf("%s\n", str);
for (i = 0; str[i] != '\0'; i++)
    printf("%c", str[i]);
printf("\n");

return 0;
}

```

```

*p is a.
Last character of apple is e.
Enter a string.
hanyang
hanyang
hanyang

```

### **example\_12-2.c :**

```

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

int main() {
    char * str1 = "Okay Buddy";
    char str2[15] = "Bless you.";

    printf("strlen(str1) is %d.\n", strlen(str1));
    printf("strlen(str2) is %d.\n", strlen(str2));
    printf("sizeof(str1) is %d.\n", sizeof(str1));
    printf("sizeof(str2) is %d.\n", sizeof(str2));

    return 0;
}

```

```

strlen(str1) is 10.
strlen(str2) is 10.
sizeof(str1) is 8.
sizeof(str2) is 15.

```

- 문자열 길이(Length)와 문자열 크기(Size)는 다른 의미다. "apple"라고 할 때 문자열의 길이는 5이지만 크기는 6이다. 크기를 따질 때에는 '\0'까지 포함해야 하기 때문이다. 따라서 문자열의 길이에 1을 더하면 문자열의 크기가 된다. `strlen`은 문자열의 길이를 구하는 함수로서 `string.h` 헤더 파일에 선언되어 있다. 이 함수에 문자열을 가리키는 포인터를 건네주면 '\0'을 만나기 전까지의 문자 개수를 돌려준다.
- 그러나 `sizeof` 연산자를 써서 문자열 크기를 구해서는 안 된다. `sizeof(str1)`의 경우 포인터 자료형의 크기를 돌려준다. `str1`이 포인터이기 때문이다. `sizeof(str2)`의 경우 배열 전체의 크기를 돌려준다.

**example\_12-3.c :**

```
#include <stdio.h>

int main() {
    char str1[10] = "apple";
    char* str2 = "apple";

    str1[4] = 'y';
    printf("%s", str1);
    printf("\n");

    str2 = "pie";
    printf("%s", str2);
    printf("\n");

    // str2[4] = 'y'; // 실행 오류 (크래시)
    // str1 = "peach"; // 컴파일 오류 (좌변이 상수)

    return 0;
}
```

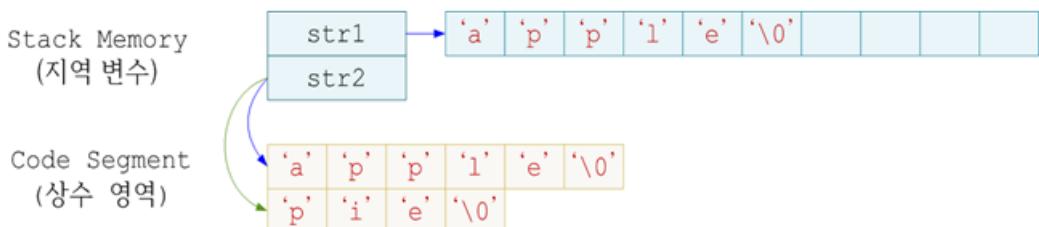
```
apply
pie
```

## 읽기 전용 문자열

- `char str1[10] = "apple"; // str1은 상수 포인터.`  
`char *str2 = "apple"; // str2는 변수 포인터.`
  
- `str1[4] = 'y'; // O.K. str1[10]은 스택 세그먼트에 존재`  
`str2[4] = 'y'; // 오류. "apple"는 코드 세그먼트에 존재`  
`str1 = "peach"; // 오류. 좌변이 상수 포인터`  
`str2 = "pie"; // str2의 포인터 값만 바뀜`

### • 코드 세그먼트 문자열

- 배열로 선언하지 않는 한, 큰 따옴표로 둘러싸인 것은 모두 코드 세그먼트의 상수 영역에 저장. 읽기 전용 메모리
- cf. 배열은 스택 세그먼트 문자열(읽고 쓰기가 가능)



## 문자열

### ● Example 12-4 실습 및 해설

- 코드 세그먼트에 저장된 문자열은 정적 변수의 특성을 지님

### ● Example 12-5 실습 및 해설

- 입력 문자열을 거꾸로 출력

### ● Example 12-6 실습 및 해설

- 입력 문장의 소문자를 모두 대문자로 바꿈

**example\_12-4.c :**

```
#include <stdio.h>

char* get_apple() {
    char* p = "apple";
    return p;
}

int main() {
    char* str = get_apple();
    printf("%s", str);
```

```
    return 0;  
}
```

```
apple
```

- 코드 세그먼트에 저장된 문자열은 정적 변수의 특성을 지닌다는 점에도 유의해야 한다. 따라서 문자열 상수는 프로그램이 시작할 때 저장되어 프로그램이 끝날 때까지 그 값을 그대로 유지한다.  
예의 get\_apple 함수 내부에 문자열 상수 "apple"은 get\_apple 함수를 빠져나가 호출 함수로 되돌아가도 사라지지 않는다. main 함수에서 포인터를 따라가 문자열을 찍을 수 있는 것은 이 때문이다.

#### example\_12-5.c :

```
#define MAX 15  
#include <stdio.h>  
  
int main() {  
    char ch, str[MAX];  
    int i;  
  
    printf("Enter a sentence to reverse.\n");  
    for (i = 0; (ch = getchar()) != '\n'; i++)  
        str[i] = ch;  
    str[i] = '\0';  
  
    for (--i; i >= 0; i--)  
        putchar(str[i]);  
  
    return 0;  
}
```

```
Enter a sentence to reverse.  
hyu minsung  
gnusnim uyh
```

#### example\_12-6.c :

```
#define MAXCHAR 20  
#include <stdio.h>  
#include <ctype.h> // for islower, toupper  
  
char* read_line() { // 입력 문장을 text 배열에 넣고 배열 시작 주소를 돌려줌  
    char ch;  
    int i;  
    static char text[MAXCHAR]; // 함수를 빠져나가도 사라지지 않게 하기 위해 정적  
    변수로 선언  
  
    printf("Enter a sentence.\n");  
    for (i = 0; (ch = getchar()) != '\n'; i++)
```

```

        text[i] = ch;
        text[i] = '\0';

    return text;
}

int main() {
    int i;
    char* p = read_line();

    for (i = 0; p[i] != '\0'; i++) {
        if (islower(p[i]))
            p[i] = toupper(p[i]);
        putchar(p[i]);
    }
    printf("\n");

    return 0;
}

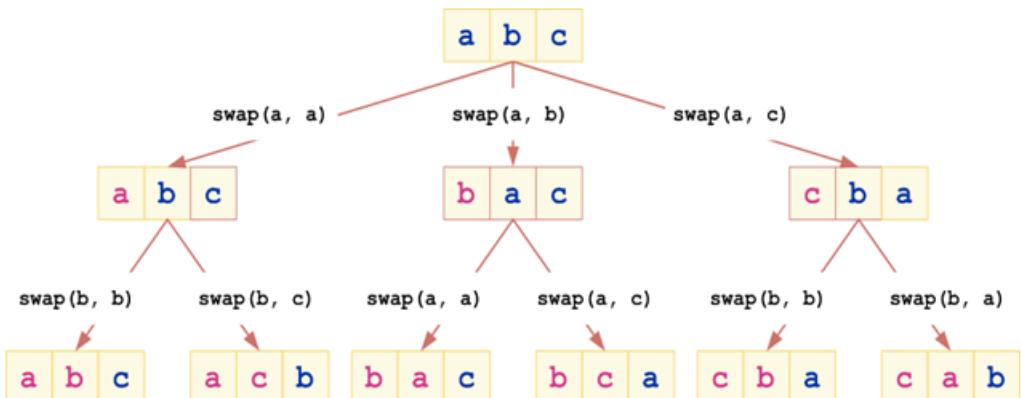
```

```

Enter a sentence.
peaceful world.
PEACEFUL WORLD.

```

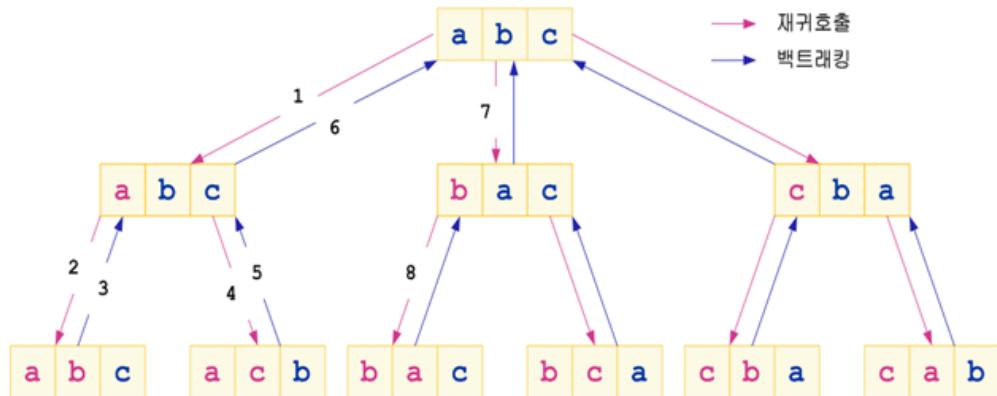
## 문자 순열



- 가능한 a, b, c 순서를 모두 나열한 문자 순열
- 가능한 모든 상태를 나열한 상태 공간 트리 (State Space Tree)
- 자리마다 구하는 방법이 같기 때문에 재귀 호출이 필요

## 문자 순열의 재귀 호출

### Example 12-7 실습 및 해설



example\_12-7.c :

```
#include <stdio.h>

void swap(char*, char*);
void permute(char*, int, int);

int main() {
    char str[] = "abc";
    permute(str, 0, strlen(str) - 1); // 1

    return 0;
}

void swap(char* x, char* y) {
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void permute(char* s, int left, int right) { // 2
    int i;
    if (left == right) // 3
        printf("%s\n", s);
    else {
        for (i = left; i <= right; i++) { // 4
            swap(s + left, s + i); // 5 -> 스왑
            permute(s, left + 1, right); // 6 -> 재귀로 순열 구함
        }
    }
}
```

```

        swap(s + left, s + i); // 7 -> 백트래킹
    }
}

```

```

abc
acb
bac
bca
cba
cab

```

- 재귀 호출에 의해 문자 순열을 구하는 프로그램이다. //1에 의해 //2의 left에는 0이, right에는 2가 들어간다. 여기서 left와 right는 permute 함수가 처리해야 할 인덱스의 범위를 말한다. //3이 베이스 케이스이다. left와 right가 일치하면 문자 하나이므로 더 이상 순열을 구할 필요가 없기 때문에 배열 내용을 출력하고 호출 함수로 되돌아간다. 여기서 베이스 케이스는 상태 공간 트리의 하단까지 내려온 상태라고 할 수 있다.
- //4의 루프를 통해 i 인덱스를 left에서 right까지 증가시킨다. 따라서 처음 이 함수를 호출했을 때 i는 0에서 2까지 간다. //5에서 가장 왼쪽에 있는 문자와 거기서 오른쪽으로 i만큼 떨어져있는 문자를 교환한다. 처음 이 함수를 호출했을 때는 i가 0에서 2까지 가면서 a와 a, a와 b, a와 c를 교환한다. swap(a, a)처럼 자기 자신과 스왑하는 것을 셀프 스왑이라고 부른다. 이렇게 함으로써 첫째 자리를 세 가지 문자로 고정시킨다. 이후 각각의 경우에서 //6의 재귀 호출에 의해 left 인덱스를 1만큼 증가시킴으로써 둘째 자리를 고정시킨다.
- 그림에서 위에서 아래로 향하는 화살표가 재귀 호출이 일어나는 과정이다. 아래에서 위로 향하는 화살표는 함수 실행이 끝나고 직전에 자신을 호출한 함수로 되돌아가는 과정이다. 직전에 지나왔던 곳으로 되돌아간다는 의미에서 이를 백트래킹이라고 부른다. 그림의 번호가 example\_12-7.c에서 재귀 호출과 백트래킹이 일어나는 순서다.
- 여기서 중요한 것은 //7이다. 이는 //5와 동일한 명령문으로 이처럼 동일한 스왑을 반복하면 스왑을 하기 이전 상태로 간다. 이 명령문이 필요한 이유는 배열을 //5의 스왑 이전 상태로 되돌려놓을 필요가 있기 때문이다. 그림을 예로 들자면 일단 1, 2순으로 재귀 호출이 진행되어 베이스 케이스를 만나면 좌하단의 "abc"를 출력한다. 이는 a를 고정시킨 상태에서 swap(b, b)에 의해 b를 고정시킨 모습니다. 이후 3의 백트래킹에 의해 호출 함수로 되돌아갈 때에는 //7에 의해 다시 한 번 swap(b, b)를 호출하여 원래의 "abc"를 복원해야 한다. 물론 swap(b, b)는 셀프 스왑이므로 스왑 결과의 배열 내용이 바뀌지는 않는다. 그러나 모든 것이 셀프 스왑은 아니다. 예를 들어 swap(b, c)에 의해 4로 내려가 "acb"를 출력한 다음에는 반드시 다시 한 번 swap(c, b)를 해야 5의 백트래킹에 의해 호출 함수로 되돌아갔을 때 "abc"를 복원할 수 있다.
- 백트래킹을 할 때마다 원본을 복원해야 하는 이유는 //5의 swap 함수가 참조 호출에 의해 원본 배열 자체를 변경했기 때문이다. 만약 값 호출이라면 피호출 함수 실행이 끝나 호출 함수로 되돌아가면 호출 당시의 값을 자동으로 회복한다. 그러나 여기서는 swap 함수가 포인터를 건네받아서 원본 배열 자체를 변경하고 있기 때문에 //7에서 프로그래머가 직접 swap 함수를 다시 호출해야 하만 원본 배열을 복원할 수 있다.

## 12-2. 문자열 입출력

```

char str[15];
printf("Enter a string.\n");
scanf("%s", str);

```

- %s 형식으로 문자열을 읽는 경우 scanf는 빈칸을 읽으면 더 이상 읽기를 멈춘다. 예의 경우 "Good Day"라고 입력하면 str 배열에는 "Good"까지만 입력되고 '\0'이 삽입된다. 따라서 빈칸 이후의 나머지 문자열인 "Day"는 입력 버퍼에 그대로 남는다.

## gets

### ● Example 12-8 실습 및 해설

- scanf 함수의 scan set 기능
  - "%[A-Za-z0-9]s"이면 영문과 숫자만 읽음
  - "%[^0-9]s"이면 숫자를 만나기 전까지 모든 문자를 읽음
  - "%[^\\n]s" 이면 '\n'을 만나기 전까지 모든 문자를 읽음
    - gets 함수와 동일

```
char *gets(char *str);
```

- get string (한 줄 읽기 함수)
- '\n'을 만나거나 파일 끝에 도달하면 읽기를 멈춤
- 문자열 끝에 '\0'을 추가
- 성공적이면 str을, 그렇지 않으면 NULL을 리턴

#### example\_12-8.c :

```
#include <stdio.h>

int main() {
    char str[128];

    printf("Enter a string.\n");
    scanf("%[A-Z]s", str);
    printf("String before lower case is %s\n", str);

    while (getchar() != '\n');
    printf("Enter a string.\n");
    scanf("%[^e]s", str);
    printf("String before e is %s\n", str);

    while (getchar() != '\n');
    printf("Enter a string with spaces.\n");
    scanf("%[^\\n]s", str);
    printf("You entered %s\n", str);

    while (getchar() != '\n');
    printf("Enter a string with spaces.\n");
```

```

    gets(str);
    printf("You entered %s\n", str);

    return 0;
}

```

```

Enter a string.
Patience
String before lower case is P
Enter a string.
The Letter
String before e is Th
Enter a string with spaces.
Sizzling Day
You entered Sizzling Day
Enter a string with spaces.
Sizzling Day
You entered Sizzling Day

```

### gets, puts

#### ● Example 12-9 실습 및 해설

- gets(get string)는 한 줄 읽기 함수
  - '\n'을 '\0'으로 대체
- puts(put string)는 한 줄 쓰기 함수
  - '\0'을 '\n'으로 대체
  - printf("%s\n", str); 대신 puts(str);

#### example\_12-9.c :

```

#include <stdio.h>

int main() {
    char str[10];
    printf("Enter a string.\n");
    gets(str);
    puts(str);

    do {
        printf("Enter another string.\n");
        gets(str);
        puts(str);
    } while (*str != '\0');

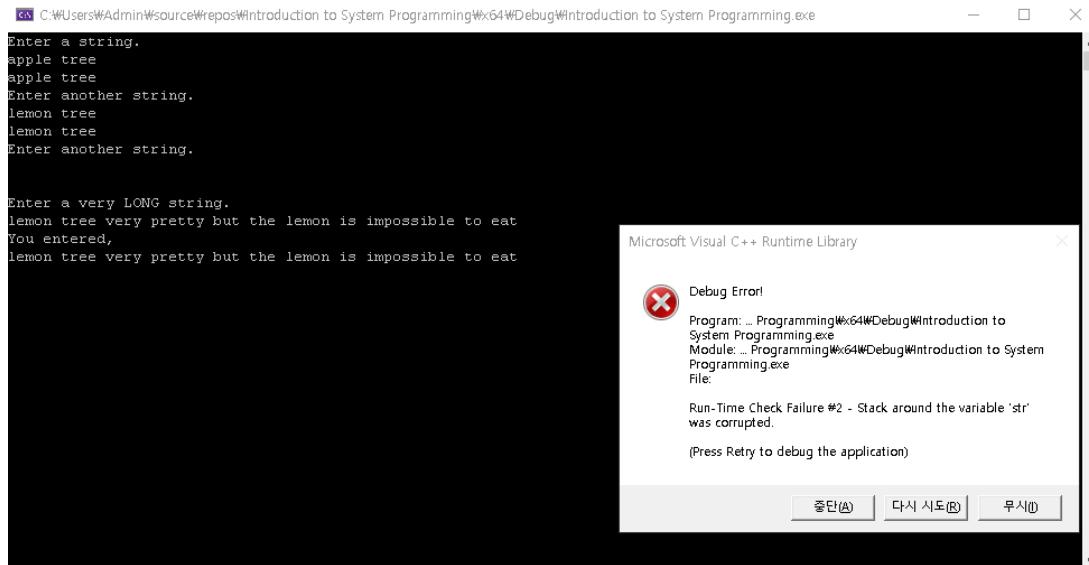
    printf("Enter a very LONG string.\n");
    gets(str);
    printf("You entered,\n");
    puts(str);
}

```

```

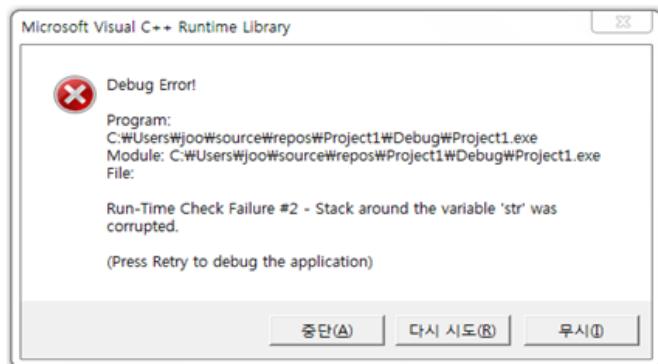
    return 0;
}

```



### gets and buffer overflow

- char str[10];  
printf("Enter a string.\n"); gets(str);  
  - gets는 '\n'이 들어 오는지만 확인
  - 배열 경계선을 넘어 문자를 저장
  - 버퍼 오버플로우(Buffer Overflow, Buffer Overrun)
  - 허용되지 않은 메모리 영역을 침범: 메모리 오염(Memory Corruption)



버퍼 오버플로우



## fgets

```
char str[15];
char *fgets(char *str, int n, FILE* fp);

fgets(str, sizeof(str), stdin); // fp =stdin
```

- fgets

- 배열 경계선을 넘지 않아 버퍼 오버플로우를 방지
- 배열 크기를 초과하거나, '\n'을 읽거나, 파일 끝에 도달했을 때 중 어느 하나라도 먼저 일어나면 읽기를 멈춤
- 둘째 인자 n을 참고로 문자가 들어올 때마다 배열 크기를 초과하는지 확인하면서 읽음
- 표준 입력에서 이 함수를 쓰려면 세째 인자를 stdin으로

## 입력 버퍼 청소

- Example 12-10, 12-11 실습 및 해설

- gets는 '\n'을 '\0'으로 대체
- fgets는 '\n'을 유지한 채 '\0'을 추가
- '\n'과 '\0'이 자리를 다투면, fgets는 '\0'을 먼저 넣는다.

```
while (getchar() != '\n');
```

- 입력 버퍼 클리어링
- fflush(stdin)은 비 표준 함수

### example\_12-10.c :

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

int main() {
    char text[10];

    printf("Enter a text.\n"); // 1
    fgets(text, sizeof(text), stdin); // 2
    printf("You entered %s.", text); // 3
    printf(" Its length is %d.\n", strlen(text)); // 4

    text[strlen(text) - 1] = '\0'; // 5
```

```

printf("You entered %s.", text);
printf(" It's length is %d.\n", strlen(text));

return 0;
}

```

```

Enter a text.
apple
You entered apple
. It's length is 6.
You entered apple. It's length is 5.

```

- gets가 '\n'을 제거하는 것과 달리 fgets는 '\n'까지 읽어들인다. 예를 들어 //1에서 apple(Enter)이라고 입력하면 //2의 text에는 'a', 'p', 'p', 'l', 'e', '\n', '\0'이 들어간다. '\n'까지도 읽어들인 다음에 문자열 끝을 나타내는 '\0'를 붙이기 때문이다. 따라서 //3에서 줄 바꿈 문자까지도 출력한다. 또, //4에서 '\n'까지 문자열로 간주하여 문자열 길이를 6으로 출력한다. 물론 이는 잘못된 결과다. 이 문제를 해결하려면 //5처럼 '\n' 자리에 '\0'을 덧씌워야 한다. 즉, text[5]의 위치에 '\0'을 넣어야 제대로 된 결과를 출력할 수 있다.

#### example\_12-11.c :

```

#include <stdio.h>

int main() {
    char first[6], last[6]; // 1

    printf("Enter first name.\n"); // 2
    fgets(first, sizeof(first), stdin); // 3

    // while (getchar() != '\n');

    printf("Enter last name.\n"); // 4
    fgets(last, sizeof(last), stdin); // 5

    printf("Full name is,\n");
    puts(first);
    puts(last); // 6

    return 0;
}

```

```

Enter first name.
Danny
Enter last name.
Full name is,
Danny

```

- '\n'과 '\0'이 자리를 다투면, fgets는 '\0'을 먼저 넣는다.
- first name과 last name을 입력받아 full name을 출력하는 프로그램이다. //1에 의해 first에는 다섯 문자까지 입력이 가능하다. 마지막에 '\0'을 붙여야 하기 때문이다. 그런데 //2에 대해 "Danny(Enter)"라고 입력하면 first에 'D', 'a', 'n', 'n', 'y', '\0'이 들어간다. 배열 마지막에 '\n'이 아닌 '\0'이 들어간 이유는

fgets가 다섯 문자를 읽자마자 더 이상 읽다가는 문자열 끝에 '\0'을 넣을 수 없다고 판단하여 곧바로 '\0'을 삽입하고 읽기를 멈추었기 때문이다. fgets는 항상 '\0'이 들어갈 자리를 염두에 주고 있기 때문에 배열 크기가 여섯 개라면 다섯 개를 입력한 상태에서 읽기를 멈춘다.

- 그러나 이렇게 되면 아직 입력 버퍼에는 '\n', 즉 Enter가 읽히지 않고 남아 있는 상태다. 이 상태에서는 //4를 출력하기는 하지만 입력 버퍼에 '\n'이 남아 있으므로 last name이 입력되기를 기다리지 않고 곧바로 //5가 실행되어 '\n'을 읽어오면 더 이상 읽기를 멈추고 결국 last에는 '\n', '\0'이 저장된다. 따라서 //6에서 줄 바꿈만 두 번 하게 된다. 한 번은 '\n'을, 또 한번은 '\0'을 '\n'으로 대체하여 찍기 때문이다.
- 버퍼를 클리어하려면 while(getchar() != '\n');
- 이 문제를 해결하는 방법은 간단하다. 항상 입력 버퍼에 남아 있던 문자를 모조리 제거한 다음에 새로운 입력을 받아들이는 것이다. 이를 버퍼 청소라 부른다. 버퍼 청소를 하려면 첫 입력이 끝난 //3 다음에 위 while 문을 삽입하면 된다. 이 명령문은 while 루프이기는 하지만 조건식 다음에 아무런 명령문이 없기 때문에 조건식만을 테스트한다. 또 그 과정에서 계속해서 getchar 함수를 실행한다. 이렇게 되면 '\n'을 만나기 전까지 입력 버퍼에 남아있던 문자는 물론 '\n'까지도 읽어버린다. 루프를 빠져나왔다는 것은 이미 '\n'까지 읽었다는 뜻이기 때문이다. 물론 fflush(file flush) 함수를 써서 fflush(stdin)처럼 해 볼 수도 있으나 이 함수는 비표준 함수로서 실행 결과가 정의되어 있지 않기 때문에 위 명령문을 쓰는 편이 훨씬 안전하다.

### 12-3. 문자열 처리 함수

strlen(string length)

```
size_t strlen(const char *str);
```

- 문자열의 길이를 리턴
- size\_t는 unsigned int를 의미



Example 12-12 실습 및 해설

**example\_12-12.c :**

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

int main() {
    char * str1 = "pine", * str2 = "apple";

    if (strlen(str1) - strlen(str2) >= 0) // 1
        printf("yes.\n");
    else
        printf("no.\n");

    if (strlen(str1) > strlen(str2)) // 2
        printf("yes.\n");
    else
        printf("no.\n");
}
```

```

        printf("yes.\n");
    else
        printf("no.\n");

    if (((int)strlen(str1) - (int)strlen(str2)) >= 0) // 3
        printf("yes.\n");
    else
        printf("no.\n");

    return 0;
}

```

yes.  
 no.  
 no.

- `strlen` 함수는 리턴 타입이 `size_t` 즉, `unsigned int`라는 점에 유의해야 한다. //1의 경우 “no”라고 찍어여 하지만 “yes”로 찍힌다. 0 이상의 정수를 나타내는 자료형인 `unsigned int` 사이에 산술 연산을 할 경우 자동으로 모듈로 연산이 이루어져 //1의 팔호 앙이  $(4 - 5) \% 2^{32}$ 이 되어 양수가 되어 버리기 때문이다. 이를 방지하려면 //2처럼 산술 연산을 제거해야 한다. 아니면 //3처럼 `unsigned int`를 `int`로 형 변환을 한 다음에 산술 연산을 해야 한다.

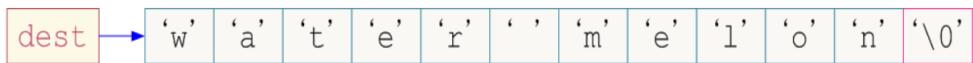
## 문자열 복사

```

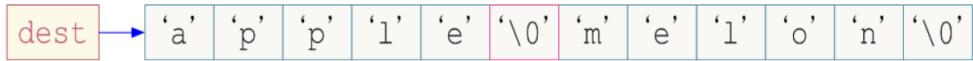
char dest[10], src[10];
dest = src;                                // Error. 배열 복사 불가
for (i = 0; i <= strlen(src); i++) // O.K. 요소 단위 복사
  dest[i] = src[i];
strcpy(dest, src);                         // O.K. 문자열 복사 함수

```

```
char* strcpy(char *dest, char *src);
```



```
strcpy(dest, src);
```



- `strcpy` 함수는 `src`가 가리키는 문자열을 `dest`가 가리키는 문자열로 복사한 다음 `dest`를 돌려준다. `src` 문자열 끝의 '\0'까지도 `dest` 문자열로 복사한다. 그림에서 보듯 `strcpy` 함수 실행 결과 `dest` 문자열은 '\0'을 처음 만나는 “apple”에서 끝나기 때문에 이후의 나머지 문자열은 모두 무시된다. 이 함수를 쓸 때에

는 인자의 순서에 특히 유의해야 한다. 뒤의 것이 앞의 것으로 복사된다. 다시 말해 나중 인자인 `src`가 가리키는 것을 처음 인자인 `dest`가 가리키는 곳으로 복사한다.

### strcpy(string copy)

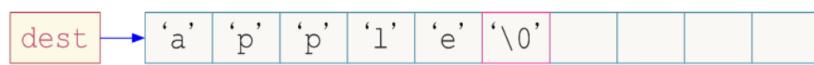
```
char *dest = "water melon", *src = "apple";  
  
strcpy(dest, src); // Error. *dest는 상수 영역  
  
dest = (char*)malloc(sizeof(char) * (strlen(src) + 1));  
strcpy(dest, src); // O.K.  
// strlen(src) + 1 WHY?
```

```
char *dest = (char*)malloc(6 * sizeof(char));  
char *src = (char*)malloc(11 * sizeof(char));  
  
strcpy(dest, "apple");  
strcpy(src, "strawberry");  
  
strcpy(dest, src); // Error. dest 크기가 작음  
strcpy(src + 1, src); // Error. 주소 중복 결과는 undefined
```

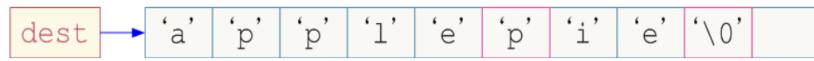
- `strlen(src) + 1` WHY?
  - '\0'까지 복사하기 위해 `strlen(src)`의 길이에 1을 더해줘야 한다.

### strcat(string concatenate)

```
char* strcat(char *dest, char *src)
```



```
strcat(dest, src);
```



- `strcat` 함수는 문자열을 이어붙이는 함수로서 `src`가 가리키는 문자열을 `dest`가 가리키는 문자열 끝에 이어붙인 다음에 `dest`를 돌려준다. 그림에서 보듯 `dest` 문자열 끝 '\0'에서부터 `src` 문자열의 첫 문자가 덧씌워진다는 점에 유의해야 한다. 또, `dest`가 가리키는 배열이 `src`가 가리키는 문자열을 이어 붙이기에 충분한지도 확인해야 한다.

## strcmp (string compare)

```
int strcmp(const char *str1, const char *str2);
```

- str1이 가리키는 문자열과 str2가 가리키는 문자열 비교
  - str1 > str2이면 양수, str1 < str2이면 음수
  - str1 == str2이면 0을 리턴
- A < B < ... < Z < a < b < ... < z
  - strcmp("a", "A")는 양수
- strcmp("abc", "abcd")는 음수(사전에 먼저 나오는 단어가 작음)

```
char str1[] = "apple";  char str2[] = "pie";
if (str1 == str2)      // 오류. 문자열이 아니라 주소를 비교
char* str = (char*)malloc(6 * sizeof(char));
strcpy(str, "apple");
if (str == "apple")
    // 오류. 힙 메모리의 "apple" 주소와 스택 메모리의 str 값을 비교
```

- if (str == "apple")은 오류다. 스택에 있는 str이라는 포인터와 코드 세그먼트에 있는 "apple"의 시작 주소를 비교할 뿐, 포인터가 가리키는 내용을 비교하는 것이 아니다.  
더구나 char \* str = (char\*)malloc(6 \* sizeof(char));에서 할당한 힙 메모리의 주소와 "apple"이라는 상수가 저장된 코드 세그먼트 주소가 같을 수는 없다. 이 경우에도 문자열을 비교하려면 strcmp 함수를 써야한다.

## 문자열 관련 함수

- Example 12-13 실습 및 해설
  - 문자열 관련 함수 테스트
- Example 12-14 실습 및 해설
  - strlen 함수 직접 구현
- Example 12-15 실습 및 해설
  - 재귀 호출에 의한 strlen 함수 구현
- Example 12-16 실습 및 해설
  - strcpy 함수 직접 구현
- Example 12-17 실습 및 해설
  - strcpy 함수 포인터 버전

```
char* my_strcpy(char* dest, const char* src){  
    char* backup = dest;  
    while (*dest++ = *src++); // *(dest++) = *(src++)  
    return backup;           // 널 문자까지 복사하면 false  
}
```

**example\_12-13.c :**

```
#include <stdio.h>  
#include <string.h>  
  
int main() {  
    char str1[30], str2[10];  
  
    printf("Enter the first string.\n");  
    gets(str1);  
    printf("Enter the second string.\n");  
    gets(str2);  
    // char * gets(char * str) : 한 줄 읽기 함수  
    // '\n'을 만나거나 파일 끝에 도달하면 읽기를 멈춤  
    // 문자열 끝에 '\0'을 추가  
    // 성공적이면 str을, 그렇지 않으면 NULL을 리턴  
  
    printf("strlen(str1) is %d.\n", strlen(str1));  
    printf("strlen(str2) is %d.\n", strlen(str2));  
    // size_t strlen(const char * str) : 문자열의 길이를 리턴 ('\0' 제외)  
  
    if (strcmp(str1, str2) == 0)  
        printf("%s and %s are equal.\n", str1, str2);  
    else if (strcmp(str1, str2) < 0)  
        printf("%s is smaller than %s.\n", str1, str2);  
    else
```

```

    printf("%s is bigger than %s.\n", str1, str2);
    // int strcmp(const char *str1, const char *str2);
    // str1이 가리키는 문자열과 str2가 가리키는 문자열 비교
    // str1 > str2이면 양수, str1 < str2이면 음수, str1 == str2이면 0을 리턴

    printf("Before strcpy, str1 = %s, str2 = %s.\n", str1, str2);
    strcpy(str1, str2);
    printf("After strcpy, str1 = %s, str2 = %s.\n", str1, str2);
    // char* strcpy(char *dest, char *src);
    // strcpy 함수는 src가 가리키는 문자열을 dest가 가리키는 문자열로 복사한 다음 dest를 돌려준다.
    // src 문자열 끝의 '\0'까지도 dest 문자열로 복사한다.
    // strcpy 함수 실행 결과 dest 문자열은 '\0'을 처음 만나는 "apple"(src의 문자열)에서 끝나기 때문에 이후의 나머지 문자열은 모두 무시된다.
    // 이 함수를 쓸 때에는 인자의 순서에 특히 유의해야 한다.
    // 뒤의 것이 앞의 것으로 복사된다. 다시 말해 나중 인자인 src가 가리키는 것을 처음 인자인 dest가 가리키는 곳으로 복사한다.

    printf("Before strcat, str1 = %s, str2 = %s.\n", str1, str2);
    strcat(str1, str2);
    printf("After strcat, str1 = %s, str2 = %s.\n", str1, str2);
    // char* strcat(char *dest, char *src)
    // strcat 함수는 문자열을 이어붙이는 함수로서 src가 가리키는 문자열을 dest가 가리키는 문자열 끝에 이어붙인 다음에 dest를 돌려준다.
    // dest 문자열 끝 '\0'에서부터 src 문자열의 첫 문자가 덧씌워진다는 점에 유의해야 한다.
    // 또, dest가 가리키는 배열이 src가 가리키는 문자열을 이어 붙이기에 충분한지도 확인해야 한다.

    printf("%s.\n", strcat(str1, "OMG!"));

    return 0;
}

```

```

Enter the first string.
warning: this program uses gets(), which is unsafe.
lemon
Enter the second string.
tree
strlen(str1) is 5.
strlen(str2) is 4.
lemon is smaller than tree.
Before strcpy, str1 = lemon, str2 = tree.
After strcpy, str1 = tree, str2 = tree.
Before strcat, str1 = tree, str2 = tree.
After strcat, str1 = treetree, str2 = tree.
treetreeOMG!.

```

**example\_12-14.c :**

```
#include <stdio.h>

int my_strlen(const char * str) {
    int i;
    for (i = 0; str[i] != '\0'; i++);
    return i;
}

int main() {
    char text[30];
    printf("Enter a text.\n");
    gets(text);
    printf("Length of the text is %d.\n", my_strlen(text));

    return 0;
}
```

```
Enter a text.
warning: this program uses gets(), which is unsafe.
apple
Length of the text is 5.
```

#### **example\_12-15.c :**

```
#include <stdio.h>

int recursive_strlen(const char * str) {
    if (*str == '\0')
        return 0;
    else
        return (1 + recursive_strlen(++str));
}

int main() {
    char text[30];
    printf("Enter a string.\n");
    gets(text);
    printf("Length is %d.\n", recursive_strlen(text));

    return 0;
}
```

```
Enter a string.
warning: this program uses gets(), which is unsafe.
apple
Length is 5.
```

#### **example\_12-16.c :**

```

#include <stdio.h>

char * my_strcpy(char * dest, const char * src) {
    int i = 0;
    while ((dest[i] = src[i]) != '\0') // dest[i] = src[i]로 dest[i]에 sr
c의 개별 요소 대입
        i++;
    return dest;
}

int main() {
    char dest[30], src[10];

    printf("Enter destination string.\n");
    gets(dest);
    printf("Enter source string.\n");
    gets(src);

    printf("On strcpy(dest, src), dest became %s.\n", my_strcpy(dest, sr
c));

    return 0;
}

```

```

Enter destination string.
warning: this program uses gets(), which is unsafe.
minsung
Enter source string.
apple
On strcpy(dest, src), dest became apple.

```

### example\_12-17.c :

```

#include <stdio.h>

char * my_strcpy(char * dest, const char * src) {
    char * backup = dest;
    while (*src != '\0') {
        *dest = *src;
        dest++; src++;
    }
    *dest = '\0';
    return backup;
}

int main() {
    char dest[30], src[10];

```

```

printf("Enter destination string.\n");
gets(dest);
printf("Enter source string.\n");
gets(src);

printf("On strcpy(dest, src), dest became %s.\n", my_strcpy(dest, sr
c));

return 0;
}

```

### strchr, strstr

```

char *strchr(const char *str, int c);
char *strstr(const char *str1, const char *str2);

```

- **strchr(string character)**
  - 문자 c를 찾아 포인터를 리턴
- **strstr(string string)**
  - str1이 가리키는 문자열에서 처음 만나는 문자열 str2를 찾아 첫 문자를 가리키는 포인터를 리턴
  
- **디폴트 프로모션 (Default Promotion)**
  - char, short는 int로, float는 double로 변환하여 전달
  - strchr 함수 원형에서 c를 int로 선언한 것은 이 때문
  - 피 호출 함수는 전달받은 인자를 다시 원하는 타입으로 변환

## strncpy, sprintf, strtok

### ● Example 12-18 실습 및 해설

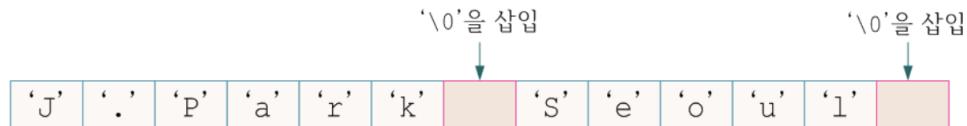
- strchr, strstr,
- strncpy(string n copy) 정확히 n 개의 문자만 copy

### ● Example 12-19 실습 및 해설

- sprintf(string printf)
- printf 함수의 출력을 문자열 형태로 배열에 저장

### ● Example 12-20 실습 및 해설

- char \*strtok(char \*str, const char \*delim);
- token이 끝나는 곳에 '\0'을 삽입



#### example\_12-18.c :

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

int main() {
    char * here, * there;
    char text[] = "This is first. This is second. This is third";
    const char ch = '.';
    here = strchr(text, ch);
    printf("Text after the first peroid is, %s\n", (here + 2));

    strcpy(text, "It is a right answer.");
    there = strstr(text, "right");
    strncpy(there, "wrong", 5);
    puts(text);

    return 0;
}
```

```
Text after the first peroid is, This is second. This is third
It is a wrong answer.
```

#### example\_12-19.c :

```

#include <stdio.h>

int main() {
    char str[100];

    char * name = "Lee eun";
    int age = 19;
    double weight = 58.5;

    char * first = "First line of a long string.";
    char * second = "Second line of a long string.";

    sprintf(str, "Name: %s, Age: %d, Weight: %lf.", name, age, weight);
    puts(str);

    sprintf(str, "%s %s", first, second);
    puts(str);

    return 0;
}

```

```

Name: Lee eun, Age: 19, Weight: 58.500000.
First line of a long string. Second line of a long string.

```

#### example\_12-20.c :

```

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

int main() {
    char str[] = "J.Park Seoul 010-2222-3456"; // 1

    char * p = strtok(str, " "); // 2
    while (p != NULL) { // 3
        printf("%s\n", p);
        p = strtok(NULL, " "); // 4
    }
    return 0;
}

```

```

J.Park
Seoul
010-2222-3456

```

- strtok 함수는 문자열을 토큰 단위로 분리하기 위한 것이다. //1의 문자열에 대해 //2의 형태로 strtok 함수를 호출하면 처음 빈칸을 만나기 전까지의 토큰만 분리하여 그 문자열의 시작 주소를 돌려준다. 따라서 좌

변 p에는 "J.Park"의 시작 주소 즉, 'J'를 가리키는 포인터가 들어간다. //3의 널 테스트는 문자열이 끝났는지 확인하기 위해서다.

- strtok 함수의 첫 인자로 처음에는 문자열의 시작 주소를 넘겨주어야 한다. //2의 str이 그것이다. 그런데 이 함수는 토큰이 끝나는 곳에 항상 '\0'(Null 문자)을 삽입하여 문자열 끝임을 표시한다. 또, 이후에 이 함수를 호출할 때는 //4처럼 NULL을 첫 인자로 넘겨주어야 그 다음 문자부터 시작해서 다음 토큰을 분리한다. 여기서 대문자 NULL은 NULL 포인터가 아니라 Null 문자를 의미한다. 문자를 삽입하기 때문에 이 함수는 원본을 변경한다는 점에도 유의해야 한다. 둘째 인자는 구분자(Delimiter)다. 예를 들어 strtok(NULL, "-");라고 하면 "-"가 나오기 직전까지만 분리하고 "-"의 위치에 '\0'을 삽입한다. strtok(NULL, ",;")처럼 둘 이상의 문자를 구분자로 쓸 수도 있기 때문에 이 인자는 항상 큰따옴표를 써서 문자열로 나타내야 한다는 점에도 유의해야 한다.

## 한글 처리

- 한글
    - 완성형은 한글을 음절 단위로 표현
    - 조합형은 초성, 중성, 종성을 구분하여 표현
  - ISO 표준 유니코드 (Unicode, ISO 10646)
    - 영문이든 한글이든 모든 문자를 2 바이트로 표현
  - ANSI 표준 아스키 코드
    - 영 문자를 1 바이트로 표현
    - 영문자는 1 바이트로, 한글은 2 바이트로 처리하는 것은 운영체제가 관리
- Example 12-21 실습 및 해설
  - 영문과 한글이 섞인 문장을 거꾸로 출력
- Example 12-22 실습 및 해설
  - wcslen(wide character set length) 함수

### example\_12-21.c :

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

int main() {
    int i;
    unsigned char str[40];

    printf("Enter a mixture of Korean and English.\n");
    gets(str);
    puts(str);

    for (i = 0; i < (int)strlen(str); i++)
        printf("%c", str[i]);
```

```

printf("\n");

/*
// 한글은 2바이트 단위로 하나의 문자를 나타내므로 2바이트 중 둘째 바이트를 먼저 찍
고 이어서 첫째 바이트를 찍어서는 제대로 된 출력이 나오지 않음.
for (i = (int)strlen(str) - 1; i >= 0; i--)
    printf("%c", str[i]);
printf("\n");
*/

for (i = (int)strlen(str) - 1; i >= 0; ) {
    if (str[i] >= 0 && str[i] <= 127) {
        printf("%c", str[i]);
        i--;
    }
    else {
        printf("%c", str[i-1]);
        printf("%c", str[i]);
        i -= 2;
    }
}
printf("\n");

return 0;
}

```

**output:**

```

Enter a mixture of Korean and English.
한글 font A4 용지
한글 font A4 용지
한글 font A4 용지
지용 4A tnof 글한

```

#### example\_12-22.c :

```

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

int main() {
    char str[] = "ABC언어"; // 1
    wchar_t uni_str[] = L"ABC언어"; // 2

    printf("sizeof(str) is %d.\n", sizeof(str)); // 3
    printf("sizeof(uni_str) is %d.\n", sizeof(uni_str)); // 4

    printf("strlen(str) is %d.\n", strlen(str)); // 5
    printf("wcslen(uni_str) is %d.\n", wcslen(uni_str)); // 6

```

```
    return 0;  
}
```

```
output:  
sizeof(str) is 8.  
sizeof(uni_str) is 12.  
strlen(str) is 7.  
wcslen(uni_str) is 5.
```

- 문자 표현 방식은 크게 SBCS(Single Byte Character Set), WBCS(Wide Byte Character Set), MBCS(Multi Byte Character Set)로 나누어 볼 수 있다. SBCS는 1바이트로 문자를 표현하는 방식으로 아스키코드가 대표적이다. WBCS는 2바이트로 문자를 표현하는 방식으로, 유니코드가 대표적이다. MBCS는 1바이트와 2바이트를 섞어서 표현하는 방식이다.
- //1의 우변은 2바이트 한글과 1바이트 영문으로 처리되므로 MBCS라 할 수 있다. //2의 우변 문자열 상수 앞의 L은 이어지는 상수가 유니코드 상수임을 의미한다. 또, wchar\_t는 유니코드의 char 타입임을 의미한다. 따라서 //2는 WBCS라고 할 수 있다.
- sizeof는 함수 호출이 아니라 연산자다. 이 연산자는 팔호 안의 변수가 정확히 몇 바이트를 차지하는지만 계산한다. //3은 8바이트다. 영문 3바이트, 한글 두 글자 4바이트, 그리고 '\0'이 1바이트를 먹기 때문이다. 그러나 //4는 12바이트다. 유니코드는 영문, 한글, 특수 문자를 가리지 않고 모두 2바이트로 처리하기 때문이다. 따라서 문자 다섯 개에 10바이트, 거기에 '\0'까지 2바이트를 먹기 때문에 12바이트다.
- 한글에 대해서는 문자열 길이도 다른 방법으로 정의할 수 있다. //5의 strlen은 영문 3바이트에 한글 두 글자 4바이트를 더해 7바이트를 출력한다. 이는 우리가 예상했던 바다. 그러나 //6은 다르다. ANSI C의 strlen 함수에 해당하는 것이 유니코드의 wcslen 함수다. 이 함수는 바이트 수가 아니라 문자의 개수를 돌려준다. 유니코드는 영문, 한글을 분간하지 않으므로 //2의 우변 문자열 길이는 5다. 이 함수 외에도 유니코드에서 사용할 수 있는 문자열 함수로는 strcpy에 해당하는 wcscpy, strcmp에 해당하는 wcscmp 등이 있다. 또, 입출력을 위한 함수로는 wscanf, wprintf, fgetws, fputws 등이 있다.

## 12-4. 버퍼 오버플로우와 컴퓨터 보안

## 버퍼 오버플로우와 컴퓨터 보안

```
void try_this( ){
    char str[4];
    strcpy(str, "ABCDEF");
    return;
}
```

- str 배열은 널 문자를 포함하여 3개의 문자를 수용.
- strcpy 함수는 '\0'을 만날 때까지 무조건 복사
- 배열 경계선을 넘어가는지 확인은 프로그래머 책임
- 할당 받지 않은 영역을 침범하는 것은 보안(Security)과 직결
  - 쓰기와 읽기 모두가 보안에 위배
  - C, C++, C#, JAVA, Python 언어
    - 명령문 실행 도중 문자 하나를 읽고 쓸 때마다 매번 배열 경계선을 넘는지 확인하면 상당한 속도 저하를 초래
    - 속도를 이유로 보안을 등한시하는 설계를 채택

## 버퍼 오버플로우

| do_this( )<br>의 스택 프레임 | 매개변수                                                  |                |
|------------------------|-------------------------------------------------------|----------------|
|                        | 지역 변수                                                 | char str[4]; ↓ |
|                        | 리턴 주소                                                 | 1004 → 2004 ↓  |
|                        | 리턴 값                                                  |                |
| 코드 세그먼트                | int main( ){<br>do_this( );<br>add_all();<br>...<br>} |                |
|                        | 2004      void format_disk( ){<br>fdisk( );<br>}      |                |

- str[4]의 오버플로우
  - 입력 내용 조작하여 리턴 주소를 2004번지로 변경
  - do\_this가 끝나면 1004번지 대신 2004번지로
  - 2004번지에 악성 코드를 미리 저장
  - 리턴 주소가 오버플로우 자체에 심어놓은 악성 코드를 가리키게 할 수도 있음

## 버퍼 오버플로우

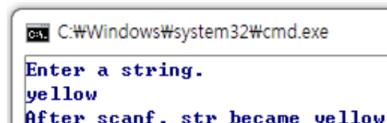
```
void do_this( ){
    int i, j;
    char username[8];
    int disk_quota = 200;
    int super_user = 0;
    ...
}
```

- `username`을 입력 받는 도중 버퍼 오버플로우
  - `disk_quota`나 `super_user` 값을 바꿀 수 있음
  - 어떤 순서로 메모리를 할당하는지는 구현에 따라 다름
- 리턴 주소만 오염되는 것은 아니다. 변수를 선언한 순서대로 메모리를 할당한다고 가정할 경우 예의 `username`을 입력받는 도중에 버퍼 오버플로우가 일어나면 그 아래 변수가 오염된다. 예의 경우 사용자에게 할당된 디스크 양(disk quota)을 바꾸거나 사용자 권한을 관리자(super user) 권한으로 바꿀 수 있다. 오버플로우가 어느 방향으로 일어나는지, 즉 `username` 이전에 선언한 변수가 영향을 받는지 이후에 선언한 변수가 영향을 받는지는 컴파일러 구현 방법에 따라 달라진다.

## scanf\_s

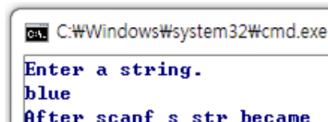
### Example 12-23 실습 및 해설

- ```
char str[4];
printf("Enter a string.\n");
scanf("%s", str);
```



```
C:\Windows\system32\cmd.exe
Enter a string.
yellow
After scanf, str became yellow
```

### Example 12-24 실습 및 해설



```
C:\Windows\system32\cmd.exe
Enter a string.
blue
After scanf_s str became
```

```
scanf_s("%s", str, sizeof(str)); // _s means security
```

- `scanf`의 버퍼 오버플로우를 방지하기 위한 함수
- `sizeof(str) - 1` 개까지의 문자만 읽음 ('\`\0`' 추가)
- 문자 하나를 읽을 때마다 매 순간 배열 경계선을 넘는지 확인 (`scanf`에 비해 느림)
- ANSI C 표준 C 함수가 아님 (이식성 저하)

### 3. `scanf` 와 `gets` 의 차이점

특성	<code>scanf</code>	<code>gets</code>
버퍼 오버플로우 가능성	제한하지 않으면 발생 ( <code>%s</code> 사용 시)	무조건 발생 (버퍼 크기를 고려하지 않음)
사용 시 대처 방법	입력 크기 제한 필요 ( <code>%9s</code> 등)	사용 금지, 대안으로 <code>fgets</code> 사용

#### example\_12-23.c :

```
#include <stdio.h>

int main() {
    char str[4];

    printf("Enter a string.\n");
    scanf("%s", str);
    printf("After scanf, str became %s\n\n", str);

    return 0;
}
```

```
Enter a string.
yellow
After scanf, str became yellow
```

- 이는 오류 메시지나 크래시 창이 뜨느냐 마느냐에 무관하게 실제로 버퍼 오버플로우가 있어난 것임.

#### example\_12-24.c :

```
#include <stdio.h>

int main() {
    char str[4];

    printf("Enter a string.\n");
    scanf_s("%s", str, sizeof(str));
    printf("After scanf_s str became %s\n", str);

    return 0;
}
```

```
output:
Enter a string.
blue
After scanf_s str became
```

- ANSI C 표준 C 함수가 아니라 비주얼 C 컴파일러에서만 쓸 수 있음 (이식성 저하)

### strncpy

- 버퍼 오버플로우를 일으키는 함수
  - scanf, gets, strcpy, strcat, sprintf 등

```
char* strncpy (char* dest, const char* src, size_t num);
```

- string n copy
- strcpy와 동일하지만 셋째 인자인 num 만큼만 복사하라는 의미
- 버퍼 오버플로우를 방지

#### Example 12-25 실습 및 해설

```
strcpy(dest, src)
≠ strncpy(dest, src, sizeof(dest)) // Error.

strcpy(dest, src)
= strncpy(dest, src, sizeof(dest) - 1); // O.K.
dest[sizeof(dest) - 1] = '\0';
```

#### example\_12-25.c :

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

int main() {
    char dest[9]; // 1
    char * src = "mju.ac.kr"; // 2
    char str[9];

    strncpy(dest, src, sizeof(dest)); // 3
    printf("%s\n", strcpy(str, dest)); // 4

    return 0;
}
```

[6] 89641 trace trap ./ch12

- 단순해 보이지만 오류가 있다. //1에서 dest 배열의 크기는 9다. 그런데 //2의 src 길이는 '\0'을 포함하면 10이다. 따라서 //3처럼 복사하면 sizeof(dest)는 9이기 때문에 dest에는 '\0'이 들어가지 않는다. 이 경우 //4가 문제를 일으킨다. strcpy 함수는 dest 문자열에서 '\0'이 나올 때까지 dest를 src로 복사하지만 dest에는 '\0'이 없기 때문에 dest 배열의 경계선을 넘어 읽기를 계속하기 때문이다. 또, 그처럼 읽은 문자를 계속해서 str 배열에 저장함으로 인해 str 배열도 오버플로우를 일으킨다.

- 따라서 strcpy를 strncpy로 바꾸려면 dest 배열이 수용할 수 있는 크기보다 하나 적은 개수를 복사해야 한다. 그리고 dest 배열의 마지막에 '\0'을 삽입해야 한다.

### memset, memcpy

#### ● Example 12-26 실습 및 해설

- 힙 메모리를 사용
- 복사에 충분한 메모리를 할당하여 오버플로우를 방지

#### ● Example 12-27 실습 및 해설

- `memset(dest, 0, sizeof(char) * (strlen(src) + 1));`
  - 셋째 인자 만큼, 둘째 인자로 초기화
- `memcpy(dest, src, sizeof(char) * (strlen(src) + 1));`
  - 무조건 인자에 지정한 바이트 수만큼 복사
  - '\0'을 만났는지 확인하지 않음

**example\_12-26.c :**

```
#include <stdio.h>
#include <stdlib.h> // to use NULL
#include <string.h> // to use strcpy

char * safe_strcpy(char *, const char *);

int main() {
    char source[100];
    char * destination = NULL;

    printf("Enter source string.\n");
    gets(source);

    destination = safe_strcpy(destination, source);
    printf("After safe string copy, destination points to\n");
    puts(destination);
    free(destination);

    return 0;
}

char * safe_strcpy(char * dest, const char * src) {
    char * backup;
```

```

dest = (char*)malloc(sizeof(char) * (strlen(src) + 1));
if (dest == NULL) {
    printf("No more memory.\n");
    exit(1);
}
backup = dest;
while (*dest++ = *src++);

return backup;
}

```

```

Enter source string.
warning: this program uses gets(), which is unsafe.
rain drops composed by chopin
After safe string copy, destination points to
rain drops composed by chopin

```

#### **example\_12-27.c :**

```

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

char * my_strcpy(char *, const char *);

int main() {
    char source[30], *destination = NULL;

    printf("Enter source string.\n");
    gets(source);
    destination = my_strcpy(destination, source);
    printf("After string copy, destination points to\n");
    puts(destination);
    free(destination);

    return 0;
}

char * my_strcpy(char * dest, const char * src) {
    dest = (char*)malloc(sizeof(char) * strlen(src) + 1);
    if (dest == NULL) {
        printf("No more memory.\n");
        exit(1);
    }

    memset(dest, 0, sizeof(char) * (strlen(src) + 1)); // 1
    memcpy(dest, src, sizeof(char) * (strlen(src) + 1)); // 2
}

```

```
    return dest;  
}
```

```
Enter source string.  
warning: this program uses gets(), which is unsafe.  
a nightmare  
After string copy, destination points to  
a nightmare
```

- //1의 memset(memory set) 함수는 dest가 가리키는 공간을 0으로 초기화한다. 둘째 인자는 초기화 값을, 셋째 인자는 dest가 가리키는 주소로부터 몇 바이트를 초기화할 것인가를 지정한다.
- //2의 memcpy 함수는 strcpy 함수와 유사하지만 약간의 차이가 있다. 이 함수는 src가 가리키는 내용을 dest가 가리키는 곳으로 복사하되 무조건 인자에 지정한 바이트 수만큼 복사한다. strcpy처럼 '\0'을 만났는지 확인하지도 않는다.

## 12-5. 문자열 배열과 문자열 포인터 배열

### Array of Array

#### Example 12-28 실습 및 해설

```
char list[3][5] = { "Kim", "Lee", "Park" };
```

'K'	'i'	'm'	'\0'		'L'	'e'	'e'	'\0'		'P'	'a'	'r'	'k'	'\0'
-----	-----	-----	------	--	-----	-----	-----	------	--	-----	-----	-----	-----	------

- 배열의 배열 (Array of Array)
  - 연속 배열 (Contiguous Array)

```
char s[2][4]; // 1  
strcpy(s[0], "Kim"); strcpy(s[1], "Lee"); // 2
```

- 문자열 역시 배열이기 때문에 2차원 배열의 각 행마다 문자열 배열을 담을 수 있다. 2차원 배열에 문자를 저장하려면 일단 //1처럼 선언해야 한다.  
이후 //2의 strcpy 명령에 의해 문자열을 넣을 수 있다. 여기서 strcpy의 첫 인자인 s[0]은 0행 전체를 가리키는 포인터이고 둘째 인자인 문자열은 그 자체로 첫 문자를 가리키는 포인터다.

세 사람의 이름을 입력받아 정렬한 후 화면에 디스플레이하는 프로그램을 작성해보자.

```
#define FRIENDS 3  
#define MAX 10  
  
int main() {  
    char name[MAX];  
    char list[FRIENDS][MAX]; // 1
```

```

int i;

for (i = 0; i < FRIENDS; i++) {
    printf("Enter your friend's name.\n"); // 2
    gets(name);
    strcpy(list[i], name); // 3
}
bubble_sort(list); // 4
print_array(list); // 5

return 0;
}

```

- 하향식 접근으로 일단 main을 작성해 보자. //1에서 2차원 정적 배열 list를 선언한다. //2에서 루프를 돌면서 세 사람 이름을 입력받아 //3에서 list[i]로 복사한다. 이후 //4에서 list를 정렬하고 //5에서 출력한다.

```

void print_array(char arr[FRIENDS][MAX]) { // 1
    int i;
    printf("\n");
    for (i = 0; i < FRIENDS; i++)
        puts(arr[i]); // 2
}

```

- 간단해 보이는 print\_array 함수부터 작성해 보자. 2차원 배열이므로 //1의 매개변수를 char \*\* arr이라고 해도 되지만 차원을 알릴 필요가 있을 때에 대비하여 예처럼 행과 열의 수를 명시하기로 한다. //2의 puts에서 arr[i]는 한 행을 가리키는 포인터다.

```

void bubble_sort(char arr[FRIENDS][MAX]) {
    int pass, current, sorted = 0;
    for (pass = 1; (pass < FRIENDS) && (!sorted); pass++) {
        sorted = 1;
        for (current = 0; current < (FRIENDS - pass); current++) {
            if (strcmp(arr[current], arr[current + 1]) > 0) // 1
                swap(arr[current], arr[current + 1]); // 2
        }
    }
}

```

- 버블 정렬 함수다. 문자열을 비교해야 하므로 //1처럼 strcmp 함수를 써야 한다. 또, 오름차순으로 정렬할 경우, 이전 것이 이후 것보다 크면 //2의 함수를 불러 교환한다. main이 이 함수를 호출할 때 bubble\_sort(list);라는 식으로 list하는 배열명 즉 포인터를 넘겨주었기 때문에 list 배열 원본을 변경할 수 있다. 또, //2에서 swap 함수를 호출할 때에는 arr[i], 즉 행 전체를 가리키는 포인터를 넘김으로써 원본 배열을 행 단위로 변경할 수 있다.

```

void swap(char * p, char * q) {
    char temp[MAX];
    strcpy(temp, p);
    strcpy(p, q);
    strcpy(q, temp);
}

```

```
    strcp(q, temp);
}
```

- 문자열을 스와핑하는 함수다. 숫자를 스와핑하는 함수와 유사하지만 문자열을 스와핑해야 하기 때문에 strcpy 함수를 써야 한다.

#### example\_12-28.c :

```
#define FRIENDS 3
#define MAX 10

#include <stdlib.h>
#include <string.h>

void swap(char * p, char * q) {
    char temp[MAX];
    strcpy(temp, p);
    strcpy(p, q);
    strcp(q, temp);
}

void bubble_sort(char arr[FRIENDS][MAX]) {
    int pass, current, sorted = 0;
    for (pass = 1; (pass < FRIENDS) && (!sorted); pass++) {
        sorted = 1;
        for (current = 0; current < (FRIENDS - pass); current++) {
            if (strcmp(arr[current], arr[current + 1]) > 0)
                swap(arr[current], arr[current + 1]);
        }
    }
}

void print_array(char arr[FRIENDS][MAX]) {
    int i;
    printf("\n");
    for (i = 0; i < FRIENDS; i++)
        puts(arr[i]);
}

int main() {
    char name[MAX];
    char list[FRIENDS][MAX];
    int i;

    for (i = 0; i < FRIENDS; i++) {
        printf("Enter your friend's name.\n");
        gets(name);
        strcpy(list[i], name);
    }
}
```

```

bubble_sort(list);
print_array(list);

return 0;
}

```

```

Enter your friend's name.
warning: this program uses gets(), which is unsafe.
이 국 종
Enter your friend's name.
강 영 희
Enter your friend's name.
차 희 진

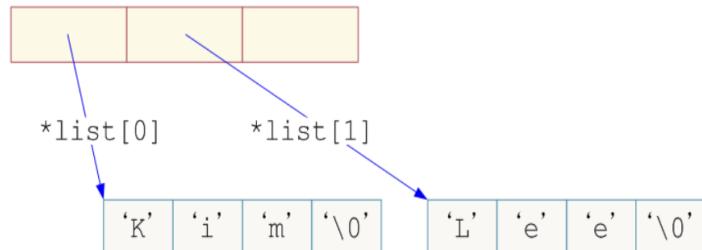
강 영 희
이 국 종
차 희 진

```

### Array of Pointers

```
char *list[3] = {"Kim", "Lee", "Park"};
```

```
char* list[3];
```



- 문자열 포인터 배열 (Array of Pointers to Strings)
  - 불연속 배열 (Ragged Array)

### Example 12-29 실습 및 해설

#### example\_12-29.c :

```

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

int main() {
    int i;
    char conti[3][5] = {"Kim", "Lee", "Park"};
    char temp[5];

    char * ragged[3] = {"Kim", "Lee", "Park"};

```

```

char * temp_ptr;

strcpy(temp, conti[2]);
strcpy(conti[2], conti[0]);
strcpy(conti[0], temp);
for (i = 0; i < 3; i++)
    printf("%s ", conti[i]);
printf("\n");
temp_ptr = ragged[2];
ragged[2] = ragged[0];
ragged[0] = temp_ptr;
for (i = 0; i < 3; i++)
    printf("%s ", ragged[i]);
printf("\n");

return 0;
}

```

Park Lee Kim  
Park Lee Kim

## 배열의 배열 vs. 포인터 배열

- Example 12-30 실습 및 해설
  - 연속 배열
- Example 12-31 실습 및 해설
  - 불연속 배열

```

char wish_list[MAX][100]; // 연속 배열
char* wish_list[MAX]; // 불연속 배열

```

- 배열의 배열(연속 배열)
  - 공간 낭비 우려
  - 배열 자체가 스택 프레임 안에 존재
- 포인터 배열(불연속 배열)
  - malloc에 의해 필요한 문자열 크기만큼 공간을 할당
  - 포인터만 스택 프레임 안에 존재. 배열은 힙 메모리에 존재

### example\_12-30.c :

```

#define MAX 10
#include <stdio.h>

```

```

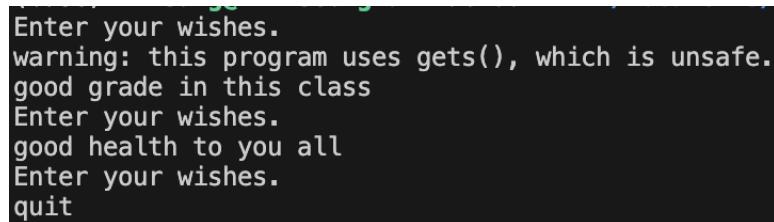
#include <string.h>

int main() {
    char wish_list[MAX][100];
    char temp[100];
    int i = 0, j;

    while (1) {
        printf("Enter your wishes.\n");
        gets(temp);
        if (strcmp(temp, "quit") == 0)
            break;
        strcpy(wish_list[i], temp);
        i++;
    }
    for (j = 0; j < i; j++)
        puts(wish_list[j]);

    return 0;
}

```



The terminal window displays the following interaction:

```

Enter your wishes.
warning: this program uses gets(), which is unsafe.
good grade in this class
Enter your wishes.
good health to you all
Enter your wishes.
quit

```

#### **example\_12-31.c :**

```

#define MAX 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
    char * wish_list[MAX];
    char temp[100];
    int i = 0, j;

    while (1) {
        printf("Enter your wishes.\n");
        gets(temp);
        if (strcmp(temp, "quit") == 0)
            break;
        wish_list[i] = (char*)malloc(sizeof(char)*(strlen(temp)+1));
        if (wish_list[i] == NULL) {
            fprintf(stderr, "No more memory.\n");

```

```

        exit(1);
    }
    strcpy(wish_list[i], temp);
    i++;
}

for (j = 0; j < i; j++)
    puts(wish_list[j]);
for (j = 0; j < i; j++) {
    free(wish_list[j]);
    wish_list[j] = NULL;
}

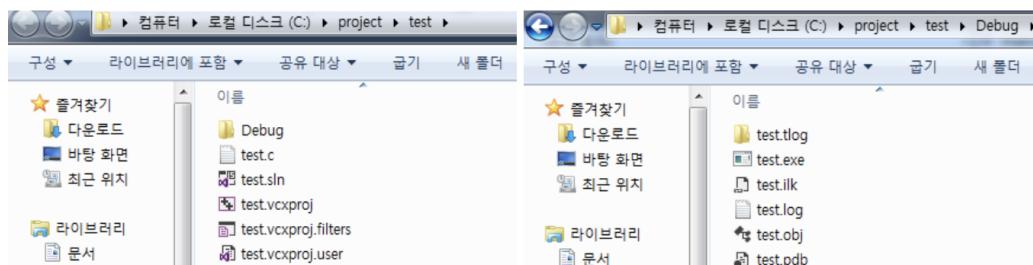
return 0;
}

```

## 명령어 라인 인자

### Example 12-32 실습 및 해설

- "솔루션 탐색기" 창 -> 프로젝트 폴더 명에 마우스 우 클릭. "파일 탐색기에서 폴더 열기"를 선택하여 실행파일 위치확인



- Debug 폴더 안에 test.exe가 실행파일
- 실행 파일 명은 **프로젝트 명과 일치**

### example\_12-32.c :

```

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

int main(int argc, char * argv[]) {
    int i;
    char * name = argv[1];
    int age = atoi(argv[2]);

    printf("argc is %d\n", argc);
    printf("%s is %d years old\n", name, age);
}

```

```

    for (i = 1; i < argc; i++)
        printf("%s ", argv[i]);
    printf("\n");

    return 0;
}

```

- 다른 함수와 마찬가지로 main 함수에도 인자를 전달할 수 있다.  
지금까지 우리가 그 기능을 사용하지 않았을 뿐이다. 예의 소스코드를 입력한 상태에서 실행이 안된다.  
main 함수에 인자가 선언되어 있지만 그것을 전달하지 않았기 때문이다.  
이 프로그램을 실행하려면 [빌드] 메뉴에서 [솔루션 빌드] 또는 [프로젝트 빌드]를 선택하여 일단 실행 파일을 만들어야 한다. 만약 프로젝트 명을 test라고 했다면 빌드 결과 test.exe라는 이름의 실행 파일이 만들어진다.
- 이후 명령 프롬포트 창을 띄우고 이 디렉터리로 이동하여 `test kim 20`이라고 하면 실행된다.

## 명령어 라인 인자

### ● 디렉터리 이동 및 실행 명령

```

C:\>cd project
C:\project>cd test
C:\project\test>dir/w
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호: BE3D-9503

C:\project\test>dir/w
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호: BE3D-9503

C:\project\test>dir/w
[.]          [..]          [Debug]
test.c       test.sln      test.vcxproj
test.vcxproj.filters test.vcxproj.user
      5개 파일           8,789 바이트
      3개 디렉터리  96,356,958,208 바이트 남음

C:\project\test>cd debug
C:\project\test\Debug>dir/w
[.]          [..]          [Debug]
test.pdb     [test.tlog] vc141.idb  vc141.pdb
              7개 파일           875,042 바이트
              3개 디렉터리  96,356,573,184 바이트 남음

C:\project\test\Debug>test kim 20
argc is 3
kim is 20 years old
kim 20

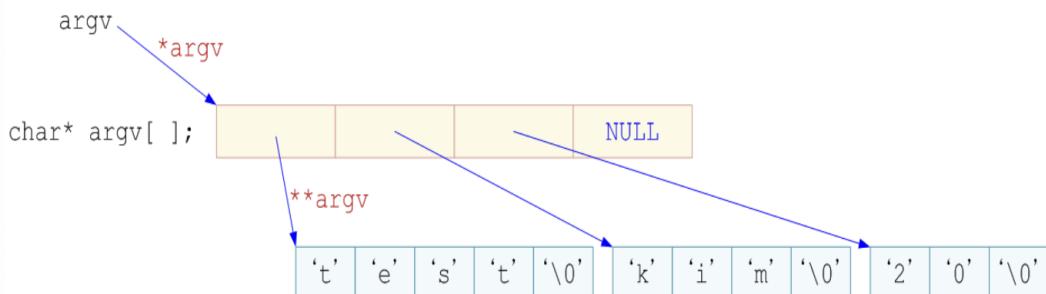
```

- MS DOS (Microsoft Disk Operating System) 명령어
  - `cd`(change directory)
  - `dir`(directory). `dir/w`(directory wide)
  - 모든 명령어를 보려면 `help`

## 명령어 라인 인자

```
> test kim 20(Enter)
```

- 명령어 라인 인자 (Command Line Argument)
  - 콘솔 명령어 형태로 메인 함수에게 전달하는 인자
  - test는 실행파일 명으로서 프로젝트 명과 일치
- int main(int argc, char\* argv[ ])
  - argc(argument count)는 인자의 개수
  - argv(argument vector)는 개별 인자를 가리키는 포인터 배열로서 2중 포인터 (char\*\* argv)



## 명령어 라인 인자

### Example 12-33 실습 및 해설

- 명령어 라인 인자를 사용한 난수 발생 프로그램

**example\_12-33.c :**

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

int main(int argc, char ** argv) {
    int i, max;
    int * rand_arr;

    srand(time(NULL));
    max = atoi(argv[1]);

    rand_arr = (int*)malloc(max * sizeof(int));
    if (rand_arr == NULL) {
        fprintf(stderr, "No more memory.\n");
        exit(1);
    }
```

```

        for (i = 0; i < max; i++)
            *(rand_arr + i) = 1 + rand() % 45;
        for (i = 0; i < max; i++)
            printf("%d ", *(rand_arr + i));
        printf("\n");

        free(rand_arr);

        return 0;
    }

```

```

● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch12 (main*?) $ ./ch12 6
34 9 44 24 20 23
● (base) minsung@iminseong-ui-MacBookAir ~/Documents/3학년 2학기 /시스템 프로그래밍 기초 /dev/ch12 (main*?) $ ./ch12 8
31 41 42 6 22 43 20 33

```

- 현재 코드에서는 argc를 검증하지 않는다.
- argv[1]이라고 하는 이유 : argv[0]은 ./ch12와 같은 프로그램 경로이다. argv[1]이 사용자가 입력한 첫 번째 명령줄 인수이다.

▼ 복잡한 선언식의 해석

## 무엇을 선언한 것인가?

```
void ( * signal ( int sig, void (* func ) (int) ) ) ( int );
```

## 복잡한 선언의 해석

void (\* signal ( int sig, void (\* func ) (int) ) ) ( int );



이름 signal을 포함하는 괄호 ()를 기준으로 원쪽으로 대칭이동

void (\* signal ( int sig, void (\* func ) (int) ) ) ( int );

void ( int ) ( \* ( int sig, void (\* func ) (int) ) signal );

void

반환하며 int를 매개변수로 갖는 함수를

가리키는 포인터를

반환하며 int sig와 void (\*func)(int)를 매개변수로 갖는 함수

signal

signal은 함수이다 !!!

void ( int ) ( \* ( int sig, void (\* func ) (int) ) signal );



void ( int ) \*

( int sig, void (\* func ) (int) )

signal

반환형

매개변수

함수이름

int sig, void (\* func ) (int)



void (\* func ) (int)



void ( int ) ( \* func )

void를

반환하며 int를 매개변수로 갖는 함수를

가리키는 포인터

func

## 13장. 구조체

### 13-1. 구조체 정의

#### 구조체

- **기본 자료형** (Primitive Data Type, Built-in Data Type)
  - 정수, 부동소수, 문자
- **파생 자료형** (Derived Data Type)
  - 배열, 포인터
  - 구조체, 열거체
  - **사용자 정의 자료형** (User-defined Data Type)
- 배열과 구조체
  - 동종 자료형을 묶은 것이 배열
  - 이종 자료형을 묶은 것이 구조체

#### 구조체 정의

```
타입 명
struct employee {
    char name[9];          ①
    int age;                ②
    int dependents;         ③
};                         세미콜론   멤버 변수 명
```

- employee: 이름 표 (Name Tag)
- Struct employee 까지가 새로운 타입 명
- 내부의 name, age, dependents는 변수 명

## 구조체 타입, 구조체 변수

```
struct employee buyer;
```

- 구조체 타입 정의와 구조체 변수 정의는 별개
- int n;
  - int에 해당하는 것이 struct employee
  - n에 해당하는 것이 buyer

```
struct employee{  
    char name[9];  
    int age;  
    int dependents;  
} buyer;
```

- 타입 정의와 변수 정의를 한꺼번에 할 수도 있음
- 변수를 선언할 때마다 타입 정의를 반복해야 하는 불편이 따름

## 데이터 베이스 용어

- 멤버 (Member) 또는 멤버 변수 (Member Variable)
  - 구조체를 이루는 요소
  - 멤버가 모여 구조체를 이룸
- 데이터베이스
  - 레코드 (Record) : 구조체
  - 필드 (Field) : 멤버 변수
  - 필드가 모여 레코드를 이룸
  - 키 필드 (Key Field) : 검색을 위한 필드
- 데이터 계층 구조
  - Bit -> Byte -> Field -> Record -> File 순으로 커짐

## 도트 연산자

```
struct employee buyer;
strcpy(buyer.name, "John Lee");
buyer.age = 34;
buyer.dependents = 3;
```

struct employee buyer;	
name[9]	'J' 'o' 'h' 'n' ' ' 'L' 'e' 'e' '\0'
age	34
dependents	3

- 도트 연산자 (Dot Operator, Member Access Operator, '.')
  - 구조체 멤버 변수에 접근

## 헤더 파일

### Example 13-1 실습 및 해설

- 자료형 정의는 대개 별도의 헤더 파일에
  - "헤더 파일" 폴더에 우 클릭. 이후 "추가 → 새 항목"
  - 소스 파일과 같은 폴더에 만들어짐
- 
- &mycar.mileage = &(mycar.mileage)
    - 도트 연산자가 주소 연산자보다 우선순위가 높음
    - 괄호와 같은 최 상위 등급

### example\_13-1.h :

```
struct car {
    char name[8];
    int year;
    int mileage;
};
```

### example\_13-1.c :

```

#include "ch13.h"
#include <stdio.h>

int main() {
    struct car mycar = {"sonata", 2015, 120000};

    printf("Name is %s.\n", mycar.name);
    printf("Year is %d.\n", mycar.year);
    printf("Mileage is %d.\n", mycar.mileage);

    printf("&mycar is %p.\n", &mycar);
    printf("&(mycar.year) is %p.\n", &(mycar.year));

    printf("Enter the mileage updates.\n");
    scanf("%d", &mycar.mileage);
    printf("The new mileage is %d.\n", mycar.mileage);

    if (mycar.mileage > 150000)
        printf("It's old.\n");
    else
        printf("It's not that old.\n");

    return 0;
}

```

```

Name is sonata.
Year is 2015.
Mileage is 120000.
&mycar is 0x16d736f40.
&(mycar.year) is 0x16d736f48.
Enter the mileage updates.
200000
The new mileage is 200000.
It's old.

```

## 동적 구조체

```
struct car mycar;
struct car *p = &mycar;           // 정적 구조체 포인터
p = (struct car *)malloc(2 * sizeof(struct car)); // 동적 구조체 포인터
```

`p = &mycar;`



`p = (struct car *)malloc(2 * sizeof(struct car));`



- (`struct car`) 크기. (`struct car*`)로 형 변환.
- 구조체가 커질수록 포인터가 유리
  - 멤버 변수 복사에 걸리는 시간적, 공간적 부담
  - 함수 호출시 배열과 마찬가지로 구조체도 포인터만 넘길 수 있음
  - 원본 변경을 막으려면 `const struct car *p`로 받음

## 화살표 연산자

```
struct car mycar;
struct car *p = &mycar;
*p.year = 2018;           // Error (WHY?)
(*p).year = 2018;         // O.K.
p->year = 2018;          // O.K. Same as above.
```

- 화살표 연산자 (Arrow Operator)
  - C 언어는 짧은 기호를 선호
  - Arrow 연산자의 왼편에는 포인터 변수가 들어가야 한다.
  - Arrow 연산자의 오른편에는 구조체 멤버 변수가 들어가야 한다.

## 중첩 구조체(Nested Structure)

```
struct point{
    int x;
    int y;
};

struct circle{
    struct point center;
    int radius;
};

struct circle cr;
cr.radius = 10;
cr.center.x = 2;
```

```
struct circle cr;
```

struct point center	x ← 2	y
radius	10	

## 중첩 구조체

```
struct circle{
    struct point{
        int x;
        int y;
    } *cptr;           // 원의 중심점을 가리키는 포인터
    int radius;
};

struct circle *cr; // 원을 가리키는 포인터
cr->cptr->x = 2; // cr이 가리키는 구조체 멤버 중,
                    // cptr이 가리키는 구조체 멤버 x에 2를 대입
```

### ● Example 13-2 실습 및 해설

#### example\_13-2.h :

```
struct point {
    int x;
    int y;
```

```

};

struct rectangle {
    struct point tl; // top left. 사각형의 좌 상단
    struct point br; // bottom right. 사각형의 우 하단
};

```

**example\_13-2.c :**

```

#include "ch13.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
    struct rectangle * p;
    int x_diff, y_diff;

    p = (struct ractangle *)malloc(sizeof(struct rectangle));
    if (p == NULL) {
        printf("No more memory.\n");
        exit(1);
    }

    printf("Enter x, y of top left.\n");
    scanf("%d%d", &(p->tl).x, &(p->tl).y);
    printf("Enter x, y of bottom right.\n");
    scanf("%d%d", &(p->br).x, &(p->br).y);

    x_diff = abs((p->br).x - (p->tl).x); // x 값의 차이
    y_diff = abs((p->br).y - (p->tl).y); // y 값의 차이
    printf("The area of rectangle is, %d.\n", x_diff * y_diff);

    return 0;
}

```

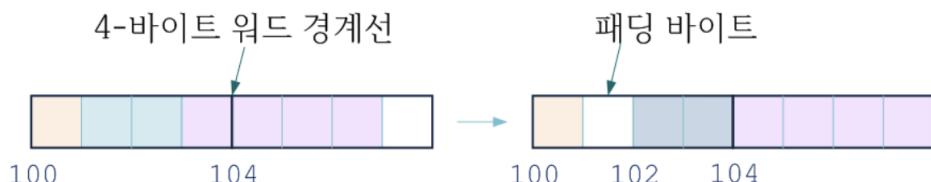
```

Enter x, y of top left.
0 0
Enter x, y of bottom right.
10 10
The area of rectangle is, 100.

```

## 변수 정렬(Variable Alignment)

```
struct this {
    char a;
    short b;
    int c;
}
```



- CPU는 워드 단위로 메모리에 접근
  - 4 바이트 워드라면, int c는 두 번 접근을 요함 (속도 저하)
  - 패딩 바이트(Padding Bytes) 삽입에 의해 변수 정렬
  - char는 1의 배수 주소에서, short는 2의 배수 주소에서 시작
  - int는 4의 배수 즉, 워드 경계선(Word Boundaries)에서 시작
- 변수 정렬(Variable Alignment)이 되어 있을 때 컴퓨터는 가장 빠르게 변수를 읽고 쓸 수 있다.  
예의 멤버 변수를 다닥다닥 붙여서 저장한다면 100번지에서 시작할 경우 왼쪽 그림처럼 char a는 1바이트, short b는 2바이트, int c는 4바이트를 차지한다.  
그런데 CPU가 메모리에 접근할 때는 항상 워드 단위로 읽고 쓴다. 따라서 만약 워드 크기가 4바이트라면 마지막 int c는 두 번에 걸쳐서 메모리에 접근해야만 한다. 100번지에서 시작하는 첫 워드를 가져와서 마지막 1바이트를 떼어낸 다음 다시 104번지에서 시작하는 둘째 워드를 가져와서 처음 3바이트를 떼어내어 이전의 1바이트와 결합해야 하기 때문이다. 메모리 접근 횟수가 늘어난다는 것은 프로그램 실행 속도가 그만큼 느려짐을 의미한다.
- 이 문제를 해결하기 위한 방법이 패딩(Padding)이다. 이는 오른쪽 그림과 같이 char a 다음에 1바이트를 비워놓는 방법이다. 이후 102번지부터 2바이트 자료형인 short b를 시작하면 마지막 int c는 워드 경계선에서 시작하기 때문에 4바이트를 한 번에 읽을 수 있다. 이렇게 하기 위해 컴파일러는 변수의 주소에 일정한 원칙을 적용한다. char 타입은 1바이트 단위이므로 1의 배수 주소에서 시작한다. 즉, 아무 주소에나 들어갈 수 있다. 그러나 short는 2의 배수 주소(예에서는 102번지)에서 시작한다. 같은 맥락에서 int는 4의 배수 주소에서 시작한다. float는 4의 배수에서 시작하지만 double은 원도우에서는 8의 배수, 유닉스에서는 4의 배수 주소에서 시작한다. 이 원칙은 구조체가 아닌 일반 변수일 경우에도 마찬가지로 적용된다.

## 구조체 실습

- Example 13-3 실습 및 해설
  - 구조체의 크기는 멤버 중 가장 큰 자료형 크기의 배수다.
- Example 13-4 실습 및 해설
  - 구조체 멤버 변수로 사용된 포인터

### example\_13-3.c :

```
#include <stdio.h>

int main() {
    struct first {
        short x;
        short y;
        short z;
    } a;

    struct second {
        float x;
        char y;
    } b;

    struct third {
        char x;
        double y;
        char z;
    } c;

    struct fourth {
        char x;
        char z;
        double y;
    } d;

    printf("sizeof(a) is %d.\n", sizeof(a));
    printf("sizeof(b) is %d.\n", sizeof(b));
    printf("sizeof(c) is %d.\n", sizeof(c));
    printf("sizeof(d) is %d.\n", sizeof(d));

    return 0;
}
```

```
sizeof(a) is 6.
sizeof(b) is 8.
sizeof(c) is 24.
sizeof(d) is 16.
```

- 구조체의 크기는 멤버 중 가장 큰 자료형 크기의 배수다.
- 패딩에는 구조체의 크기가 가장 큰 멤버 크기의 배수여야 한다는 원칙이 있다.
  - 이 원칙에 의하면 변수 a의 크기는 6바이트다. 모두 short 타입의 2바이트 정수이기 때문이다.
  - 그러나 변수 b의 크기는 8바이트다. b에서 가장 큰 멤버 변수는 float x이므로 구조체 전체의 크기도 4의 배수여야 하기 때문이다. 따라서 float x가 4바이트지만 char y이후에 3바이트를 패딩 바이트로

처리한다.

- 변수 c의 크기는 24바이트다. char x 다음에 7바이트를 패딩해야 double y가 8바이트 경계선에서 시작할 수 있기 때문이다. 마지막 char z 다음에도 7바이트를 패딩한다. 구조체의 크기는 가장 큰 멤버 변수의 크기인 8의 배수여야 하기 때문이다.
- 사실상 c와 d는 동일한 선언이다. 그러나 c가 24바이트인 데 비해 d는 16바이트다. char x, char z가 각각 1바이트를 차지한 후 6바이트가 패딩되고 이후 double y가 8바이트이기 때문이다. 이처럼 프로그래머가 직접 멤버 변수의 순서를 바꿈으로써 패딩에 소요되는 메모리 공간을 줄일 수 있다.

#### example\_13-4.c :

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

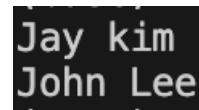
struct employee {
    char * name; // 구조체 멤버 변수로 사용된 포인터
    int resident_no;
};

int main() {
    struct employee seller, * p;
    p = &seller;
    p->name = "Jay kim";
    puts(p->name);

    p->name = (char*)malloc(sizeof(char) * 40);
    if (p->name == NULL) {
        printf("No more memory.\n");
        exit(1);
    }

    strcpy(p->name, "John Lee");
    puts(p->name);
    free(p->name);

    return 0;
}
```



```
Jay kim
John Lee
```

- 포인터 변수를 구조체 멤버로 거느릴 수도 있다. 그 경우 그 포인터는 일반적인 포인터와 완전히 동일한 방법으로 사용된다.
  - p->name = "Jay kim";에서 우변 문자열이 코드 세그먼트 상수 영역에 저장되고 시작 주소가 좌변의 name으로 들어간다.

- `p->name = (char*)malloc(sizeof(char) * 40);` 에서 힙 영역에 40바이트를 확보하고 시작 주소가 `p->name` 으로 들어간다.
- `strcpy(p->name, "John Lee");` 에 의해 `p->name`이 가리키는 곳으로 문자열이 복사된다.

## 13-2. 구조체 복사

### 구조체 대입 연산

#### ● Example 13-5 실습 및 해설

- `struct point increment(struct point s);`
- `struct point a = {1, 1}, b, c;`
  
`b = a; // 구조체 단위로 대입이 가능`  
`c = increment(a); // 인자 전달은 일종의 대입 연산`  
`if (a == b) // 비교 불가`

	비교	대입	인자 전달
배열	불가능	불가능	배열 시작 주소 복사(참조 호출 효과)
구조체	불가능	가능	구조체 멤버 복사(값 호출 효과)

#### ● Example 13-6 실습 및 해설

- 배열을 멤버로 거느린 구조체를 복사하면 배열도 복사됨

**example\_13-5.c :**

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

struct point increment(struct point s) { // 1
    s.x += 1;
    s.y += 1;
    return s; // 2
}

int main() {
    struct point a = {1, 1}, b, c;

    b = a; // 3
    printf("b is (%d, %d).\n", b.x, b.y);
}
```

```

c = increment(a); // 4
printf("c is (%d, %d).\n", c.x, c.y); // 5

if (a.x == b.x && a.y == b.y) // 6
    printf("a and b have same values.\n");
if (&a == &b) // 7
    printf("a and b are same points.\n");

return 0;
}

```

```

b is (1, 1).
c is (2, 2).
a and b have same values.

```

- 구조체 단위로 대입하거나 복사할 수 있다.
  - //3처럼 구조체 단위로 통째로 대입할 수 있다. 이는 배열 단위로 대입할 수 없다는 것과는 대조를 이룬다. 구조체 단위로 대입할 수 있다는 것은 구조체 멤버 변수 하나하나가 **자동으로** 복사된다는 의미이다. 따라서 //3에 의해 a의 멤버 변수인 x, y가 각각 b의 멤버 변수인 x, y로 복사되어 들어간다.
- 인자 전달은 일종의 대입 연산이다.
  - 구조체 단위로 대입할 수 있다는 것은 함수 호출 시에 //4에서 전달한 a가 //1의 s로 구조체 단위로 복사되어 들어감을 의미한다. 다시 말해, 함수 호출 시 구조체를 넘기면 **자동으로** 구조체 멤버 하나하나가 매개변수 구조체의 멤버로 복사되어 들어간다. 물론 이는 당연히 값 호출이다. 그러나 //2에서 변경한 구조체를 리턴 값으로 돌려주었기 때문에 //5에서는 1을 더한 값이 출력된다. 일반적으로 피호출 함수는 호출 함수에 단 하나의 값만 돌려줄 수 있다. 그러나 이처럼 멤버 변수가 두 개인 구조체 하나를 돌려주면 실제로는 두 개의 값을 돌려준 것이 된다.
- 그러나 구조체 단위의 관계 연산이나 논리 연산은 허용되지 않는다. 예를 들어 (a == b)나 (a ≠ b)처럼 비교해서는 안된다. 두 점의 좌표 값이 같은지를 비교하려면 //6처럼 멤버 변수를 하나하나 비교해야 한다. 특히 두 점의 좌표 값이 같은 것과 두 점이 같은 점이라는 것은 별개의 사실이라는 점에 유의해야 한다. 좌표 값이 같은지를 알아내려면 //6처럼 하면 된다. 그러나 두 점이 같은 점인지를 알아내려면 //7처럼 변수의 주소를 비교해야 한다.
- 배열과 구조체의 공통점과 차이점도 구분할 필요가 있다. 배열 단위로 서로 비교할 수 없으며 구조체 단위로도 서로 비교할 수 없다는 것이 공통점이다. 그러나 차이점도 있다. 배열 단위로는 대입이 불가능하기 때문에 함수 호출 시 배열을 넘기면 배열의 시작 주소만 복사되어 들어갈 뿐, 배열 요소가 복사되지는 않는다. 반면, 구조체 단위로는 대입이 가능하기 때문에 함수 호출 시 구조체를 넘기면 구조체 요소가 하나하나 자동으로 복사된다.

#### example\_13-6.c :

```

#include <stdio.h>

struct person { // 1
    char name[10];
    int age;
}

```

```

};

int over_twenty(const struct person *p) { // 2
    if (p->age > 20)
        return 1;
    else
        return 0;
}

int main() {
    struct person s1, s2 = {"Jay", 24};

    s1 = s2; // 3
    printf("s1's name is %s.\n", s1.name); // 4

    if (over_twenty(&s1))
        printf("%s is over twenty.\n", s1.name);

    return 0;
}

```

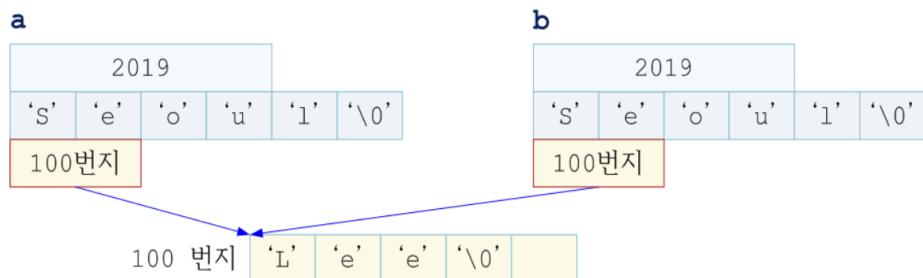
**s1's name is Jay.  
Jay is over twenty.**

- 구조체를 복사하면 멤버 변수인 배열도 자동으로 복사된다.
- 구조체끼리는 대입이 가능하지만 배열끼리는 대입이 불가능하다. //1의 구조체 내부에는 name이라는 배열이 있다. 그렇다면 //3처럼 s2 구조체를 s1 구조체에 대입할 경우 구조체 내부의 name 배열도 복사가 될까? 결론부터 말하자면 복사가 된다. 구조체를 대입할 때는 배열을 포함한 모든 멤버 변수가 복사되어 들어간다. 다시 말해 배열이 구조체 멤버일 경우에는 배열 단위로도 복사가 된다. //4에서 그 점을 확인할 수 있다. 이는 일반적인 배열 사이에서는 불가능한 일이다.

## 얕은 복사(Shallow Copy)

### Example 13-7 실습 및 해설

- a.name = (char\*)malloc(5 \* sizeof(char));  
strcpy(a.name, "Lee");  
b = a; // 얕은 복사  
strcpy(b.name, "Park");



- 구조체끼리 복사하면 **디폴트로 얕은 복사**가 이루어짐
- 포인터를 복사**하므로 a.name이 가리키는 공간이 공유됨
- strcpy(b.name, "Park");에 의해 "Lee"가 "Park"로 **덧 씌워짐**

example\_13-7.c :

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

struct employee {
    int entry_year; // 입사 연도
    char residence[6]; // 거주지
    char * name; // 사원 이름
};

int main() {
    struct employee a = {2019, "Seoul", NULL};
    struct employee b;

    a.name = (char*)malloc(5*sizeof(char));
    strcpy(a.name, "Lee");

    b = a;
    strcpy(b.name, "Park");

    printf("%d %s %s\n", a.entry_year, a.residence, a.name);
    printf("%d %s %s\n", b.entry_year, b.residence, b.name);
}
```

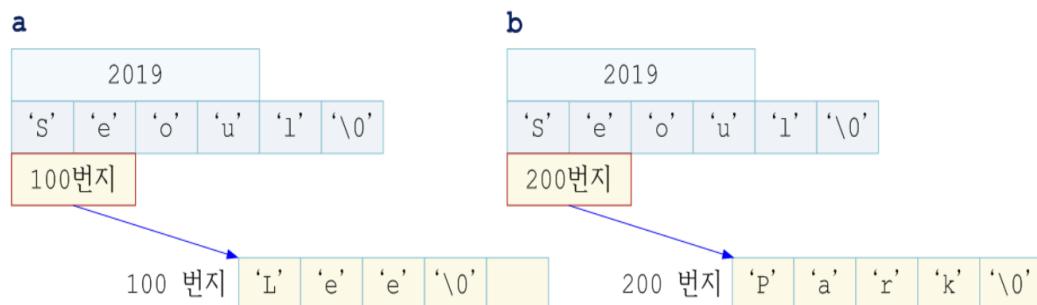
```
    return 0;  
}
```

2019 Seoul Park  
2019 Seoul Park

### 깊은 복사(Deep Copy)

#### ● Example 13-8 실습 및 해설

```
b = a;  
b.name = (char*)malloc(5 * sizeof(char)); // 별도 메모리 생성  
strcpy(b.name, "Park");
```



#### example\_13-8.c :

```
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
struct employee {  
    int entry_year;  
    char residence[6];  
    char * name;  
};  
  
void show_name(struct employee her) {  
    printf("%s\n", her.name); // 1  
    strcpy(her.name, "홍두루미"); // 2  
    // strcpy(a.name, "홍두루미"); // 값 호출이기 때문에 피호출 함수가 원본 구조체  
    //에는 접근 불가  
}  
  
int main() {
```

```

struct employee a = {2019, "Seoul", NULL};
a.name = (char*)malloc(9*sizeof(char));
strcpy(a.name, "박하영");

show_name(a); // 3
printf("%s\n", a.name);
return 0;
}

```

박 하 영  
홍 두 르 미

- 함수 호출 시 구조체를 전달하면 자동으로 얇은 복사가 이루어진다.
  - 함수 호출 시 구조체 자체를 전달하면 자동으로 얇은 복사가 진행된다. 따라서 멤버 변수 중 포인터가 있으면 주소 값만 복사되어 넘어간다. 예에서는 //3에 의해 a.name 포인터가 show\_name 함수의 her.name 포인터로 복사되어 들어간다. 그러나 이전 예와는 다를 예의 경우에는 프로그래머가 그것을 직접 깊은 복사로 바꿀 수 없다. 인자를 던달하는 메커니즘은 완전히 컴파일러 소관이기 때문이다.
  - 구조체 자체를 전달하면 값 호출이기 때문에 피호출 함수가 원본 구조체에는 접근할 수 없다. 그러나 여기서 원본 구조체에 접근할 수 없다는 말은 스택 메모리에 관한 이야기다. 즉, 피호출 함수의 스택 프레임 아래에 깔린 호출 함수의 스택 프레임에 접근할 수 없다는 것일 뿐, 힙 메모리에 접근할 수 없다는 말은 아니다. 주소만 있으면 힙 메모리에는 어떤 함수든 접근할 수 있다. a.name은 malloc에 의해 만들어진 힙 메모리를 가리키는 포인터다. 따라서 show\_name 함수는 //1에 의해 힙 메모리에 있는 문자열을 읽을 수는 있다. 그러나 여기서 her.name은 a.name이라는 포인터 값만 복사해서 받은 얇은 복사이기 때문에 a.name이 가리키는 곳과 her.name이 가리키는 힙 메모리는 공유된다. 따라서 show\_name 함수 내부에서 //2처럼 her.name이 가리키는 내용을 바꾸면 원본 구조체인 a.name이 가리키는 내용도 바뀐다는 점에 유의해야 한다.

### 구조체 포인터 전달

```

do_this(&big);                                ①
void do_this(const struct employee *p); ②

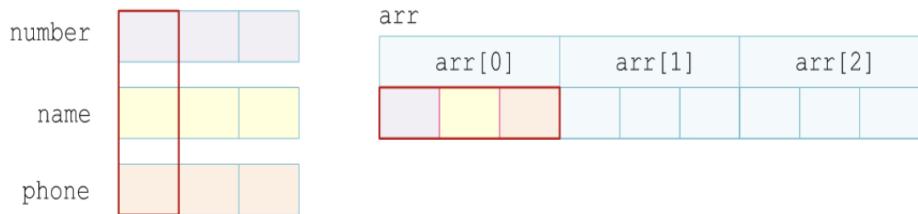
```

- 읽기만 할 때에도 구조체 포인터를 전달
  - 시간적, 공간적 효율 때문
  - 피 호출 함수는 \*p를 const로 선언

## 12-3. 구조체 배열과 구조체 포인터 배열

## 구조체 배열(Array of Structures)

- Example 13-9 실습 및 해설
  - 3개의 배열
- Example 13-10 실습 및 해설
  - 하나의 구조체 배열



- 구조체 배열의 필요성
  - 3 개의 분리된 배열보다는 하나의 구조체 배열

### example\_13-9.c :

```
#include <stdio.h>

int main() {
    int i;
    int entry[3] = {201504, 202011, 202502}; // 입사 연월
    char name[3][20] = {"Kim", "Lee", "Park"}; // 이름
    char phone[3][15] = {"02-1234", "031-1234", "051-1234"}; // 전번

    for (i = 0; i < 3; i++)
        printf("%d %s %s.\n", entry[i], name[i], phone[i]);

    return 0;
}
```

```
201504 Kim 02-1234.
202011 Lee 031-1234.
202502 Park 051-1234.
```

### example\_13-10.c :

```
#include <stdio.h>

struct employee {
    int entry;
```

```
char name[20];
char phone[15];
};

int main() {
    int i;
    struct employee arr[3] =
    {
        {201504, "Kim", "02-1234"},
        {202011, "Lee", "031-1234"},
        {202502, "Park", "051-1234"}
    };

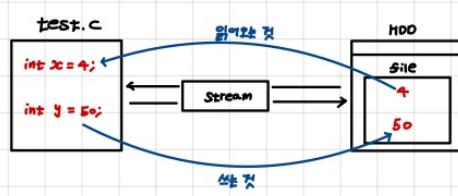
    for (i = 0; i < 3; i++)
        printf("%d %s %s.\n", arr[i].entry, arr[i].name, arr[i].phone);

    return 0;
}
```

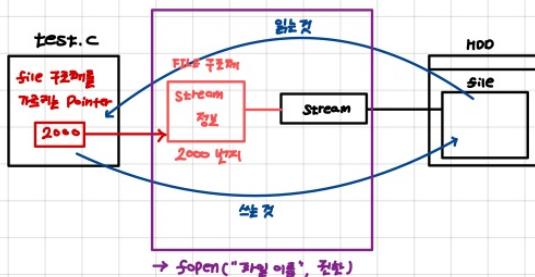
```
201504 Kim 02-1234.
202011 Lee 031-1234.
202502 Park 051-1234.
```

▼ 영상 보강 - 14장

## 14장. 파일 입출력



파일을 읽고 쓰는 과정에서 Stream이라는 것을 만들.  
Stream은 물체의 데이터를 읽어오기도 하고, 데이터를 Stream에 쓰기도 함.  
Stream은 우리가 직접 C에서 할 수 있는 것 X



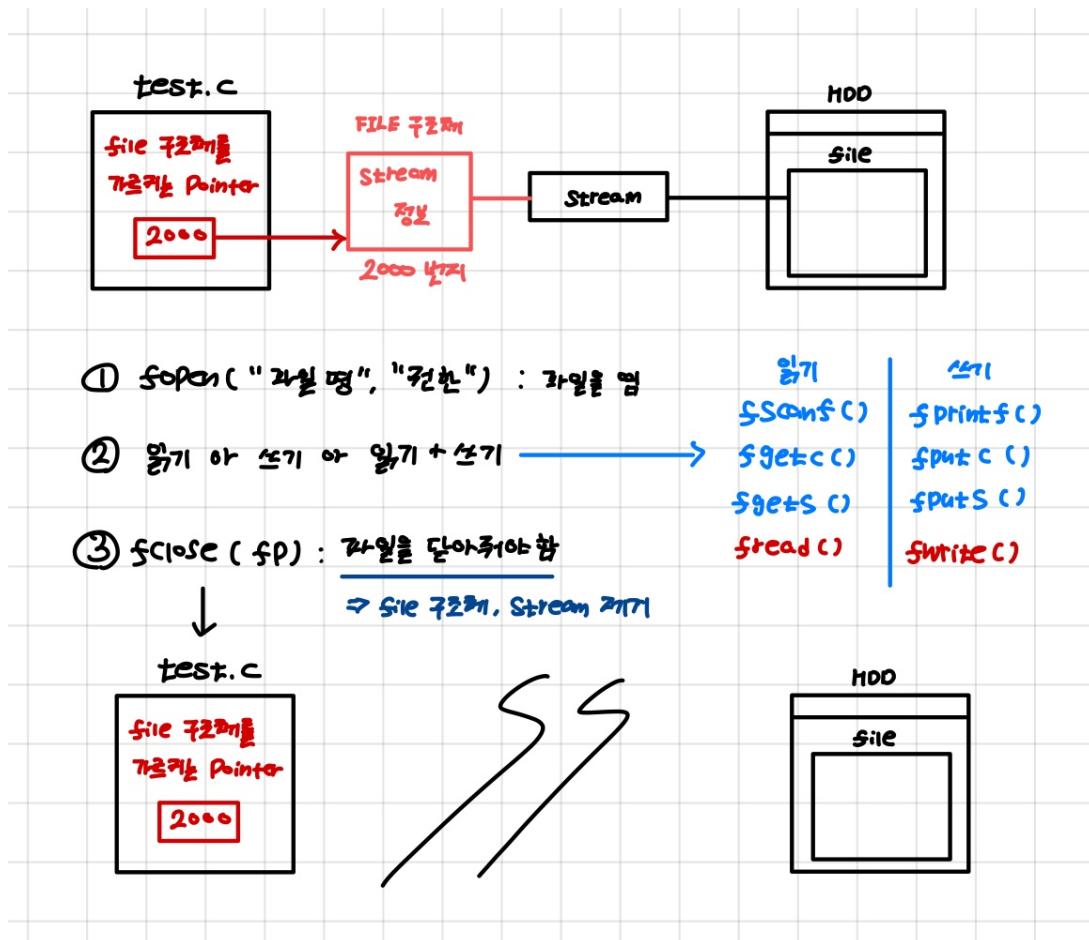
Stream을 생성하여 다루는 File과 동일하.  
Stream은 직접 Control할 수 있는 것이 아님. → Stream에 대한 정보를 갖지 않은 FILE 구조체를 만듬.  
`test.c`가 FILE 구조체를 가질때는 Pointer (FILE \*fp)를 만듬.  
FILE 구조체를 가르키는 Pointer의 값(2000 번址)에 가서 Stream에 대한 정보를 이용해 우리가 FILE에 내용을 쓰거나 읽어올 수 있음.

우리가 할 것 : FILE 구조체를 가르키는 Pointer를 받아온 것 → `fopen` 함수 사용.

이제, HDD에 있는 어떤 파일을 Control하는 Stream을 만들것지 정해야함.

`fopen` 예시 : File을 열어서 그 File하고 연결되는 Stream을 만들고, 그 Stream에 대한 정보를 갖는 FILE 구조체를 만들어서 주소를 return  
만약 File이 없거나 File을 열 수 없으면 NULL을 반환

`fopen ("file 이름", "접두")`  
접두 설명 (   
    읽기  
    쓰기  
    읽기 + 쓰기 )



## 파일 열기

`FILE* fp = fopen("C:/data.txt", "wt");` 문자열 중간 유의할 것!

- data.txt는 물리적 파일의 이름
- FILE은 논리적 파일을 정의한 구조체 자료형
- 프로그램과 입출력 장치 사이에 스트림 (Stream)을 형성
- 스트림에 관한 모든 정보를 FILE 구조체에 저장
- 운영체제는 물리적 파일을 열어 FILE 구조체로 사상
- 이후 이 함수가 리턴한 구조체 포인터를 통해 물리적 파일에 접근

**모드:**

- w : 쓰기 모드
- r : 읽기 모드
- t : 텍스트 모드 (w+ : 쓰기 및 읽기 모드, r+ : 읽기 및 쓰기 모드)
- b : 바이너리 모드

쓰기 + text mode : wt

읽기 + binary mode : rb

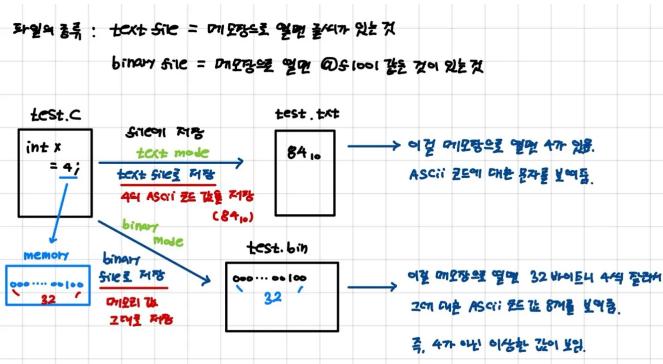
## fopen( )

```
FILE* fopen(const char* name, const char* mode)
FILE* fp = fopen("C:/data.txt", "wt");
```

mode: "wt", "wb", "rt", "rb", "at", "ab"  
=w                   =r                   =a  
**주의! t가 기분이 좋다**

- 파일 열기 모드 (Mode) **주의! t가 기분이 좋다**

- "wt" (write text) 는 텍스트 파일 형태로 쓰기
- "rb" (read binary) 는 2진 파일 형태로 읽기
- "at" (append text) 는 파일 끝에 새로운 텍스트를 추가
- "r" 모드에서 파일이 없으면 NULL 포인터를 리턴
- "w", "a" 모드에서 파일이 없으면 파일을 만들고 포인터를 리턴
- '+' (업데이트 지시자)는 읽기와 쓰기를 같이 해야 할 때 사용
- fopen 이후에는 NULL 테스트를 통해 제대로 열렸는지 확인



## 텍스트 파일, 2진 파일

- 텍스트 파일은 아스키 문자로 번역하여 저장
  - 10진수 36을 문자 '3' '6'으로 변환
  - 00110011 00110110으로 저장
  - 메모장 파일
- 2진 파일은 메모리 내용 그대로를 파일에 저장
  - 10진수 36이라면 00100100을 저장
- 대용량 파일을 읽거나 쓰려면 2진 파일이 유리
  - 2진수나 문자로 변환할 일이 없기 때문에 속도가 빠름
  - 숫자 32817은 2진 파일로는 2 바이트. 텍스트 파일로는 5 바이트. 2진 파일이 작은 공간을 차지

### Example 14-1 실습 및 해설

- from.txt 파일을 읽어 to.txt 파일에 복사

 `getc()` : 키보드에서 들어오는 것을 한 문자를 읽어옴

`fgetc(src)` : 파일 `src`에서 한 문자를 읽어옴

`putc()` : 화면에 한 문자를 출력함

`fputc(ch, dest)` : 파일 `dest`에 `ch`을 씁

`EOF` : 파일의 끝

`feof(src)` : `src`의 끝까지 갔는지 확인

### example\_14-1.c :

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

int main() {
    int ch;

    FILE * src = fopen("file/from.txt", "rt"); // 읽기 모드로 열
    FILE * dest = fopen("file/to.txt", "wt"); // 쓰기 모드로 열

    if (src == NULL || dest == NULL) {
        printf("Error opening file.\n");
        exit(1);
    }

    while ((ch = fgetc(src)) != EOF) // fgetc로 src에서 한 문자를 읽어와서 ch
    //에 대입. 이것이 파일의 끝(EOF)에 도달하지 않았다면 (즉, 정상적으로 읽어왔다면)
        fputc(ch, dest);

    if (feof(src))
        printf("Copy complete.\n");
    else
        printf("Error while copying.\n");

    fclose(src);
    fclose(dest);

    return 0;
}
```

## EOF, feof

- `while ((ch = fgetc(src)) != EOF)`  
    `fputc(ch, dest);`
- `if (feof(src))`  
    `printf("Copy complete");`  
`else`  
    `printf("Error while copying");`
- EOF
  - 파일 끝에 도달했을 때
  - from.txt 파일이 없을 때
  - 하드 디스크 용량이 부족해서 새로운 파일을 만들 수 없을 때
  - 하드웨어 장애로 파일을 열 수 없을 때
  - 동시에 열 수 있는 파일의 수를 넘어갈 때
- feof
  - 실제로 파일 끝에 도달했는지 확인하는 함수

## fgetc, getc

파일 입출력	<code>fgetc(FILE* fp)</code> = <code>getc(FILE* fp)</code>	<code>fputc(int c, FILE* fp)</code> = <code>putc(int c, FILE* fp)</code>
표준 입출력	<code>fgetc(stdin)</code> = <code>getchar( )</code>	<code>fputc(int c, stdout)</code> = <code>putchar( )</code>

- `fgetc(file get character)`
  - 문자 단위로 파일 읽기
- `getc(get character)`
  - `fgetc`와 동일한 함수
  - 단, 속도를 높이기 위해 **매크로 함수**로 구현
  - 표준 입출력에도 `fgetc`와 `fputc` 사용 가능
    - `fp` 위치에 `stdin`, `stdout`을 삽입

### ● Example 14-2 실습 및 해설

- 123abc에서 숫자가 아님을 판단하는 순간은 이미 'a'라는 문자를 읽은 다음임
- 문자열을 제대로 찍으려면 'a'를 다시 스트림에 삽입해서 **읽기 이전 상태로 되돌려야 함**

### example\_14-2.c :

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

int main() {
    int ch;
    unsigned int num = 0;

    FILE * fp = fopen("file/from.txt", "rt");
    if (fp == NULL) {
        printf("Error opening file.\n");
        exit(1);
    }

    ch = fgetc(fp);
    while(isdigit(ch)) {
        // isdigit(ch) : ch가 숫자면 true, 아니면 false
        num = num * 10 + ch - '0'; // ch == 82 (문자 : 2, 0의 아스키 코드 값 : 80) -> 82 - '0' == 82 - 80 == 2
        ch = fgetc(fp);
    }
    printf("Number is %d.\n", num);

    fclose(fp);

    return 0;
}
```



gets() : buffer에서 문자열(string)을 읽어옴

puts() : 문자열(string)을 화면에 출력

fgets() : 파일에서 문자열(string)을 읽어옴 → \n까지 읽어서 하나의 문자열로 처리

fputs() : 파일에 문자열(string)을 씀

char\* fgets(char \* buff, int max, FILE\* fp);

: 파일 fp에서 max-1만큼 읽어서 buff에 저장 → 파일 끝까지 읽었다면 NULL 리턴

int\* fputs(const char\* str, FILE\* fp);

: str을 파일 fp에 씀

## fgets, fputs

```
char* fgets(char* buff, int max, FILE* fp);
int* fputs(const char* str, FILE* fp);
```

- 한 줄 단위로 문자열을 읽고 쓰는 함수
  - 표준 입출력: gets, puts
  - 파일 입출력: fgets, fputs
- fgets(file get string)
  - 최대(max - 1)개의 문자를 읽어 buff에 저장
    - 버퍼 오버플로우 방지에 유리
  - '\n'을 만나거나, EOF가 돌아오거나, 버퍼가 차면 멈춤
  - '\n'을 저장하고 그 다음에 '\0'을 삽입
    - cf. gets는 '\n'을 없앰

### ● Example 14-3 실습 및 해설

12

#### example\_14-3.c :

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

int main() {
    char buff[100];
    FILE * src = fopen("file/from.txt", "rt");
    FILE * dest = fopen("file/to.txt", "wt");

    if (src == NULL || dest == NULL) {
        printf("Error opening file.\n");
        exit(1);
    }

    while (fgets(buff, sizeof(buff), src) != NULL) // 파일 끝까지 읽었다면 NULL이 나옴
        fputs(buff, dest);

    if (feof(src))
        printf("Copy complete.\n");
    else
        printf("Error while copying.\n");

    fclose(src);
    fclose(dest);
}
```

```
    return 0;  
}
```

### fwrite, fread binary mode

```
size_t fwrite(const void* ptr, size_t size, size_t numelts,  
             FILE* stream); 파일 내용을 Stream에 size numelts 만큼 쓰기  
size_t fread(void* ptr, size_t size, size_t numelts,  
             FILE* stream); Stream에 있는 size numelts 만큼 읽어와 ptr에 저장
```

- **fwrite(file write)**
  - 2진 파일 쓰기
  - `fwrite((void *)&this, sizeof(this), 1, fp);`
  - `this`라는 구조체 변수 하나를 2진 파일에 기록
  - 성공적으로 출력한 구조체의 개수를 리턴. 이는 인자로 전달한 `numelts`의 개수와 일치해야 함
  
- **fread(file read)**
  - 2진 파일 읽기
  - `fread((void *)&this, sizeof(this), 1, fp);`
  - 2진 파일에 있던 구조체 하나를 `this`라는 구조체로 읽음

#### fwrite, fread 예제 :

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
typedef struct {  
    int id;  
    char name[20];  
} studentType ;  
  
int main() {  
    int i, count;  
  
    studentType stdnt, stdnt2;  
  
    FILE * dest = fopen("file/student.bin", "wb");  
  
    stdnt.id = 2023;  
    strcpy(stdnt.name, "James"); // 배열이라 strcpy 사용  
  
    count = fwrite(&stdnt, sizeof(studentType), 1, dest); // 출력한 구조체
```

의 개수를 리턴. 이는 인자로 전달한 `numlets`의 개수(여기서는 1)와 일치해야함

```
fclose(dest);

// dest에 작성하였으니, 그것을 읽어와서 stdnt2에 대입해보자

FILE * src = fopen("file/student.bin", "rb");

fread(&stdnt2, sizeof(studentType), 1, src);
printf("%d %s\n", stdnt2.id, stdnt2.name);

fclose(src);

return 0;
}
```

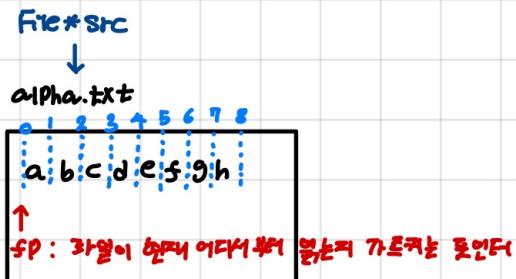
#### **textmode** 입출력 :

읽기	쓰기
fgetc	fputc
fgets	fputs
fscanf	fprintf

- `fscanf` : 파일에서 읽어오는 것 - `scanf`하고 사용하는 건 똑같음. 단지 파일을 하나 지정해줘야 함.
- `fprintf` : 파일에 쓰는 것

#### **binary(이진) mode** 입출력 :

읽기	쓰기
fread	fwrite



`fgetc (Src) = a` → 이후 `SPtr` 1로 이동

`ftell (Src)` : 현재 `SP`가 어디있는지 return

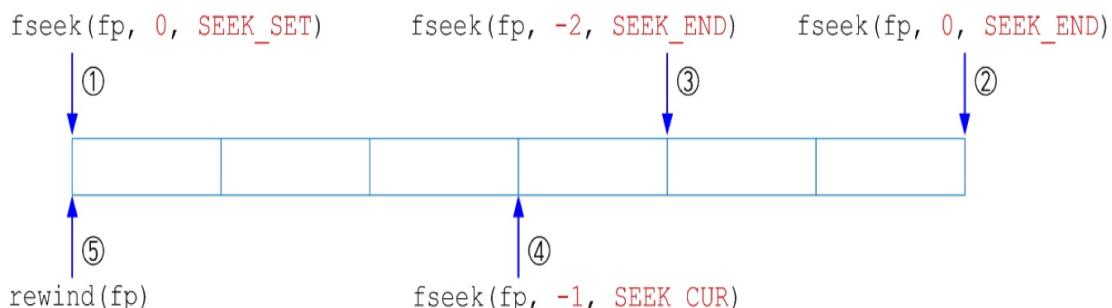
위 상황에서 `ftell (Src) = j`

`rewind (Src)` : `SP`가 어디있든 간에 처음 위치로 보내줌

`fseek` 함수 : `SP`를 임의적 위치로 박을 수 있음.

### fseek

```
int fseek(fp, 0, SEEK_SET); // 위치 포인터를 파일 처음으로
int fseek(fp, 0, SEEK_END); // 파일 끝으로
int fseek(fp, -2, SEEK_END); // 끝에서 두 칸 이전으로
int fseek(fp, -1, SEEK_CUR); // 현 위치에서 한 칸 이전으로
void rewind(fp); // 위치 포인터를 파일 처음으로
long ftell(fp); // 현 위치 이전에 있는 바이트 수
```



- 임의 접근 (Random Access) 함수

- cf. 순차 접근 (Sequential Access) : scanf, fgetc



SEEK\_SET : 처음 위치를 나타냄

SEEK\_END : 마지막 위치를 나타냄

`fseek(fp, 0, SEEK_SET)` : 처음 위치로

`fseek(fp, 1, SEEK_SET)` : 처음 위치에서 1바이트 이동

`fseek` 함수의 2번째 인자 : 기준점

기준점 :

- 0 : 그 자리
- 양수 : 파일 끝 방향으로
- 음수 : 파일 시작 방향으로