

전산 SMP 6주차

2015. 11. 03

김범수

bskim45@gmail.com

Special thanks to 박기석 (kisuk0521@gmail.com)

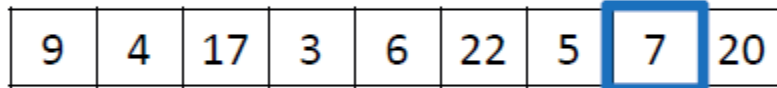
지난 내용 복습

Array, Sorting

자료구조

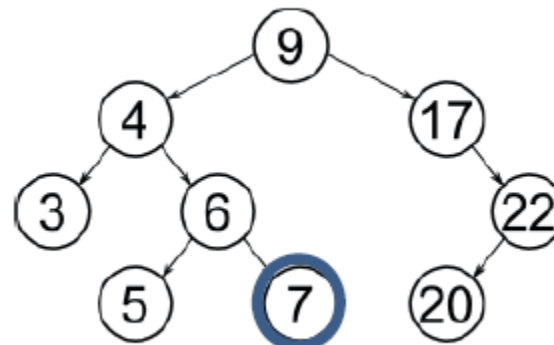
- 많은 데이터를 예쁘게 잘 저장해서
 - 쉽고 빠르게 꺼내 쓰려고
 - 데이터가 많아지면 찾는데도 오래 걸린다.
-
- Example: finding a number from a set of numbers
 - How many comparisons do we need to retrieve 7?

In linear array



8 comparisons

In binary search tree



4 comparisons

선언, 초기화, 접근

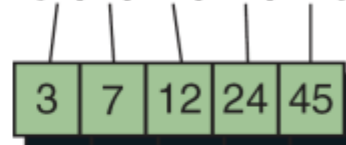
(a) Basic Initialization

```
int numbers[5] = {3, 7, 12, 24, 45};
```



(b) Initialization without Size

```
int numbers[ ] = {3, 7, 12, 24, 45};
```



(c) Partial Initialization

```
int numbers[5] = {3, 7};
```



The rest are
filled with 0s

(d) Initialization to All Zeros

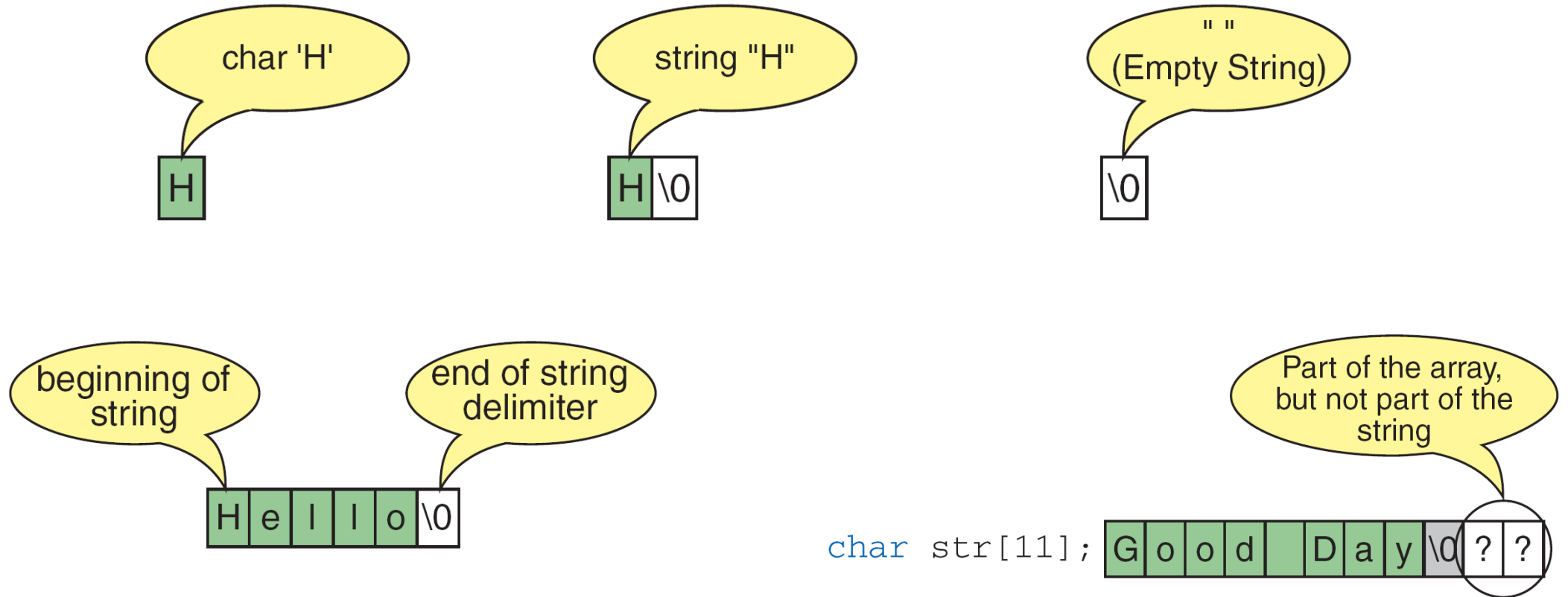
```
int lotsOfNumbers [1000] = {0};
```



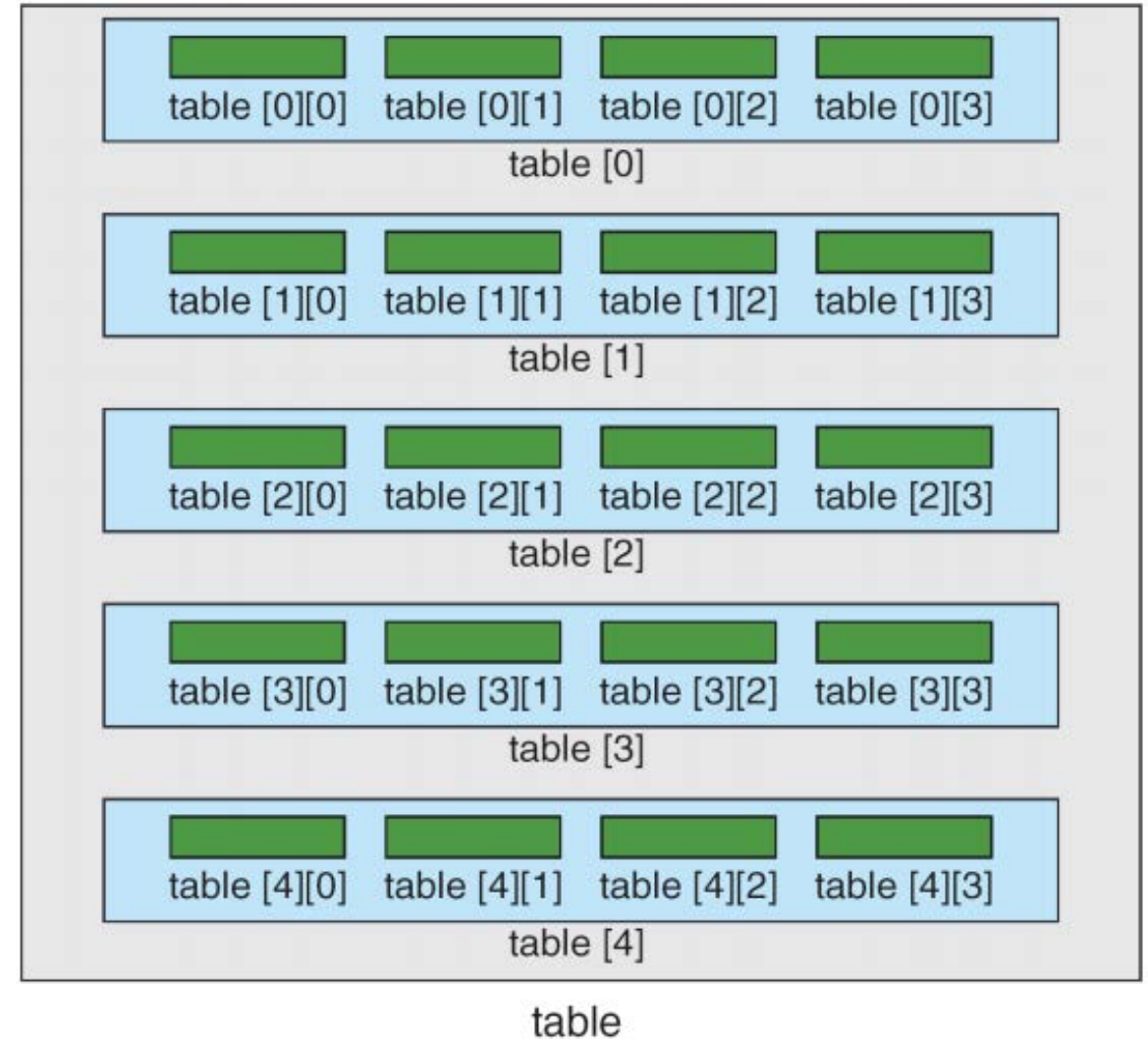
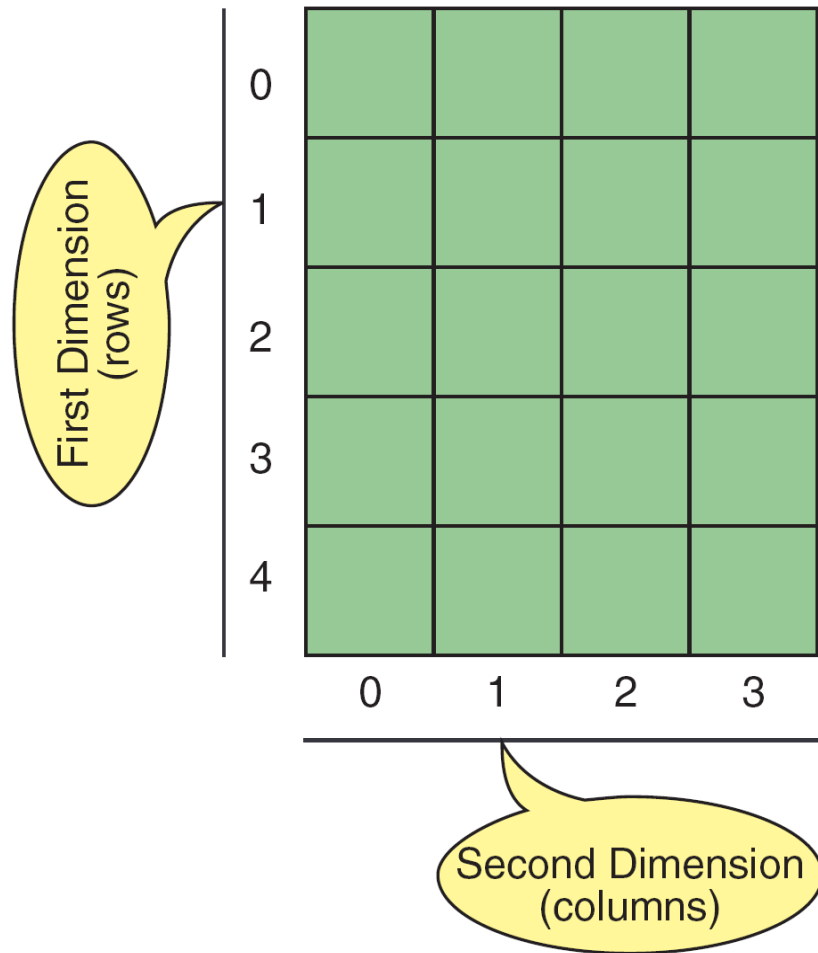
All filled with 0s

문자열 (string)도 결국은 배열이다

- char 형의 배열 & 끝에 널문자 '\0'
- 배열 & 포인터를 완전하게 배우고 string에 대해서 더 자세히 합니다.



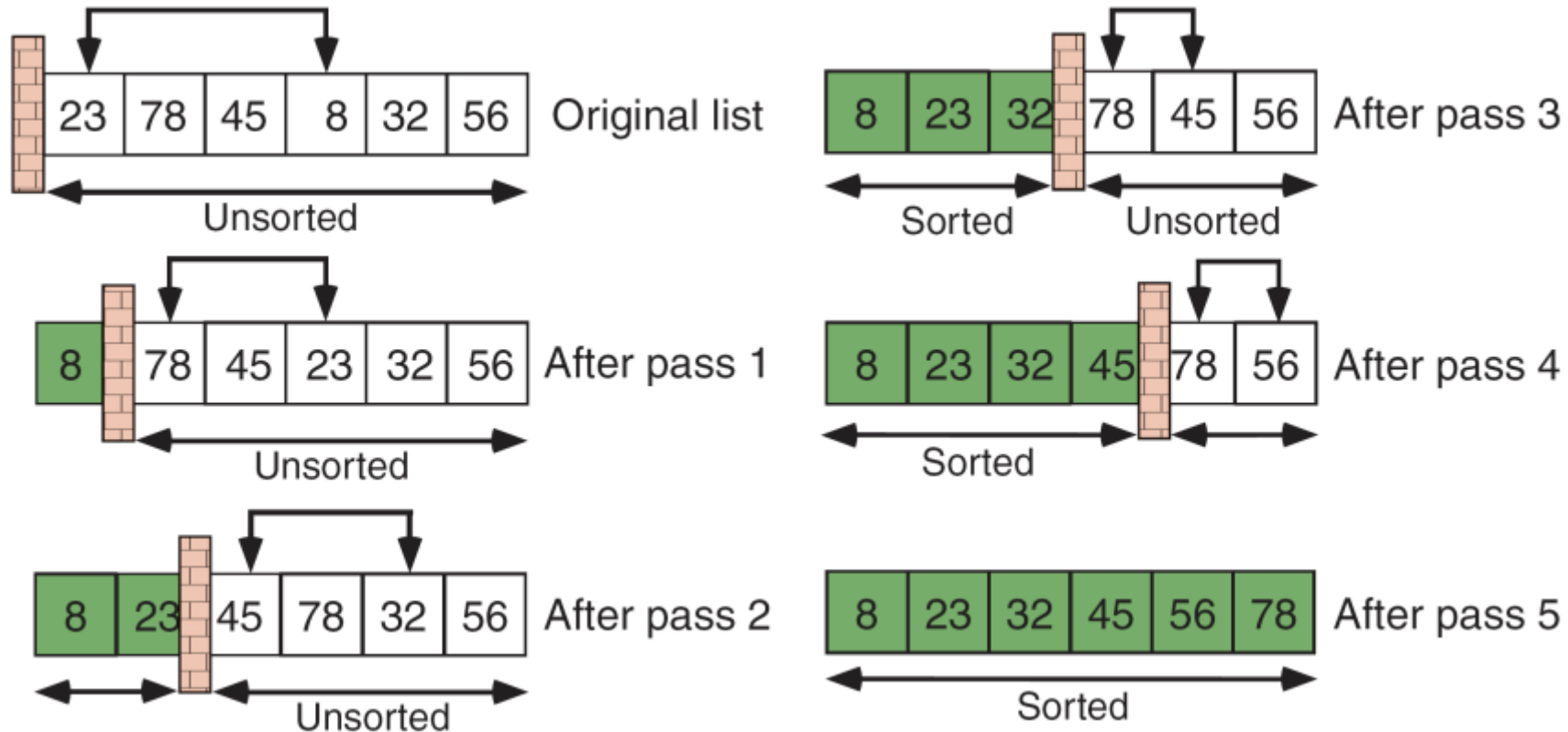
Two-dimensional Array (Matrix)



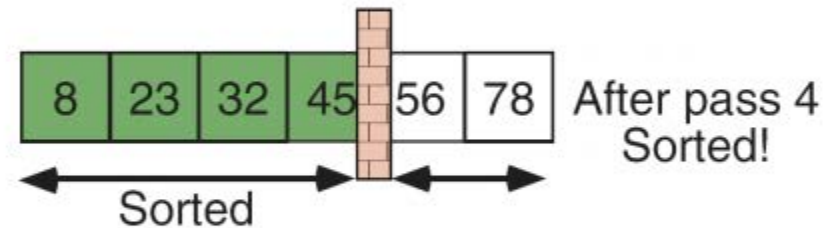
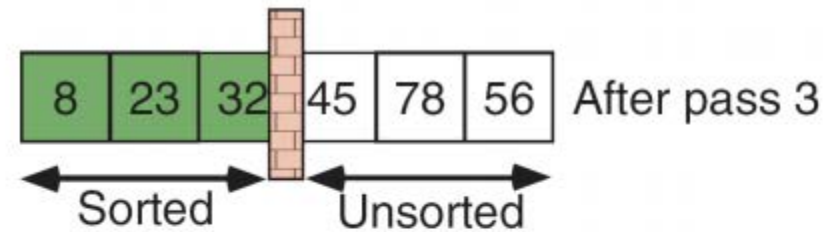
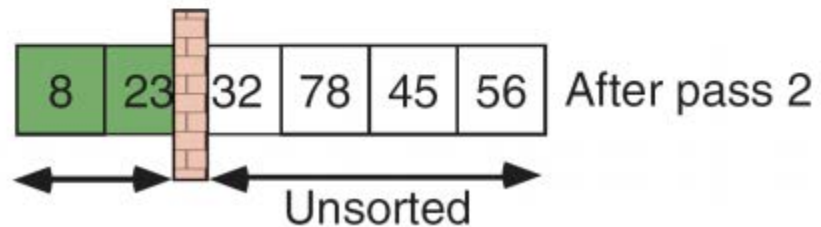
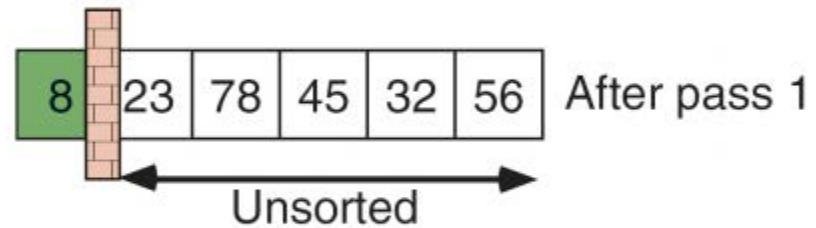
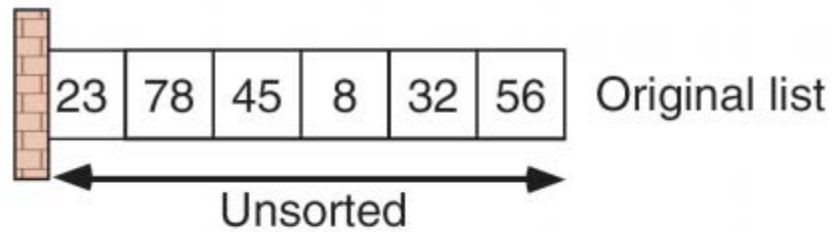
Array Sorting

- Selection Sort
 - Bubble Sort
 - Insertion Sort
-
- 책에 있는 코드 꼭 보고 한번씩 짜 보세요 - 시험에 나옴

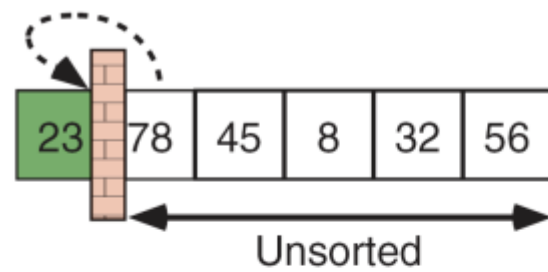
Selection Sort



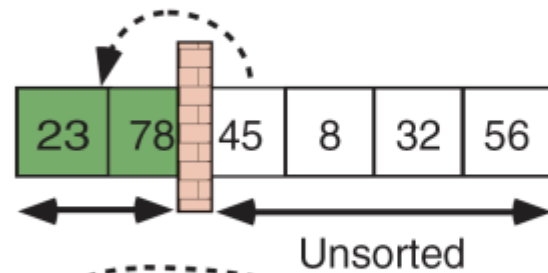
Bubble Sort Example



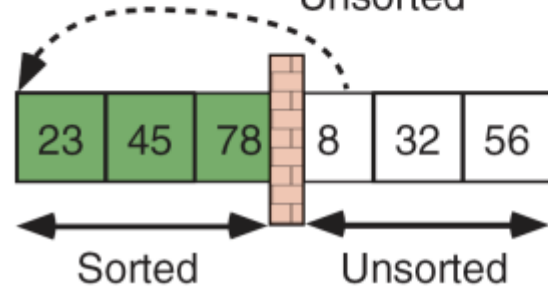
Insertion Sort Example



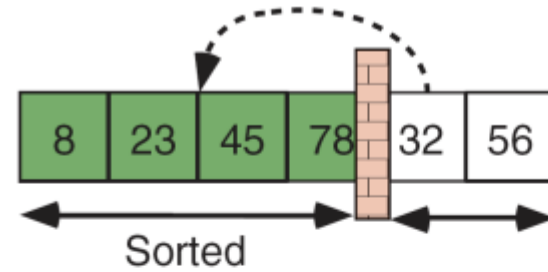
Original List



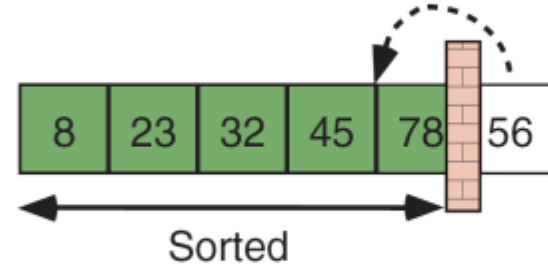
After pass 1



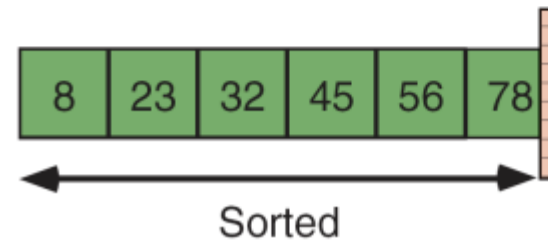
After pass 2



After pass 3



After pass 4



After pass 5

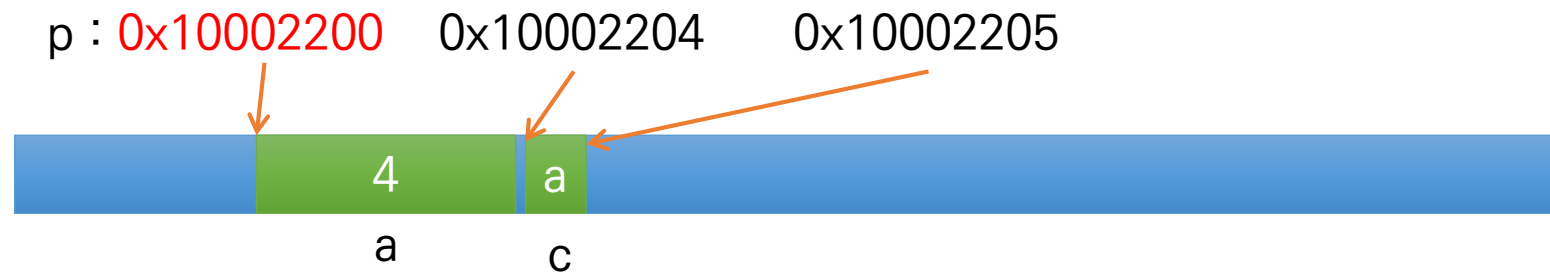
오늘 할 것

- 대망의 Pointer
- Pointer Applications (relation with Array)
- Dynamic Memory Allocation

Pointer

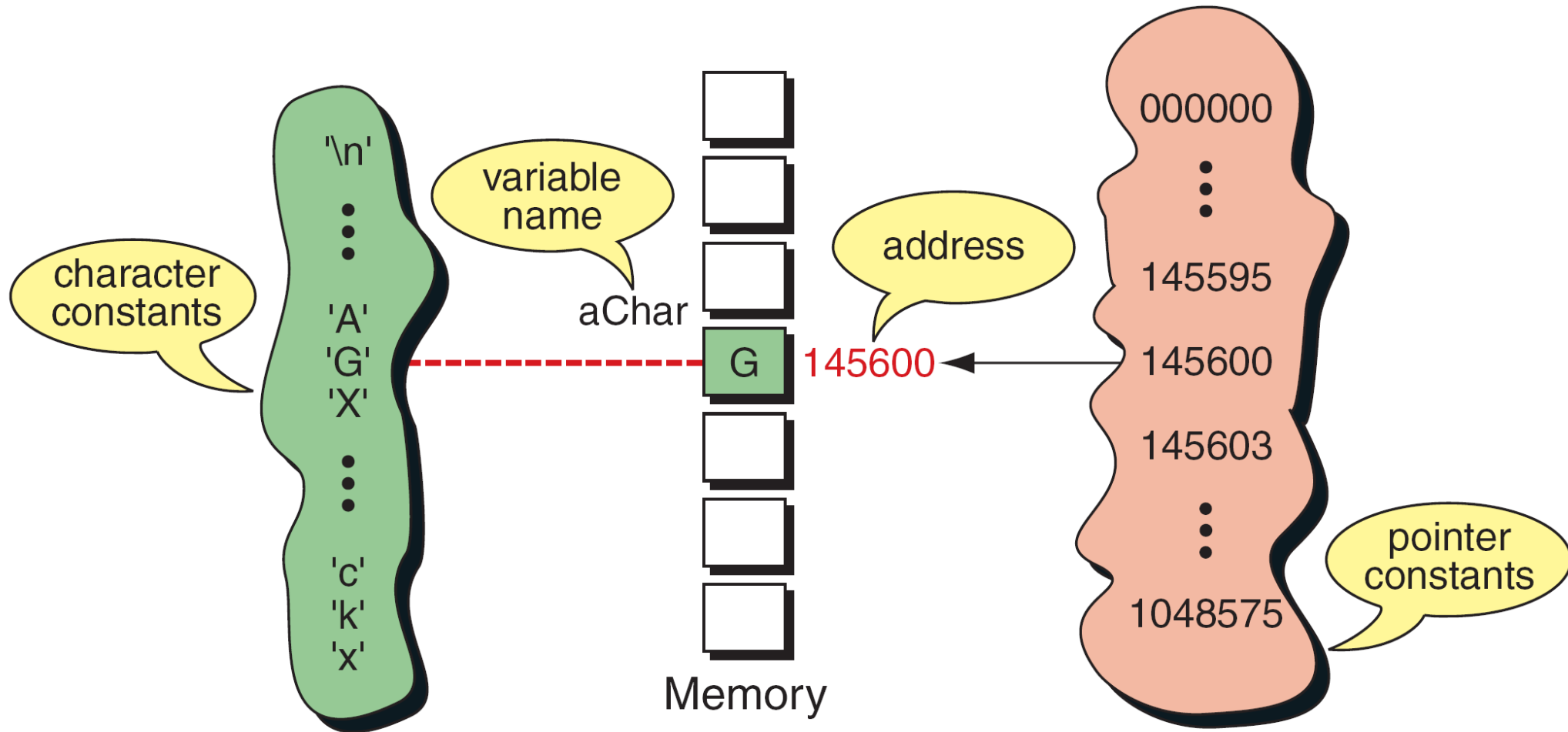
Pointer란?

- 데이터 접근에 사용되는 **주소**를 저장하는 타입 - 포인터도 타입이다
- `int a = 4; char c = 'a'; int *p = &a;`



- 메모리는 긴 일차원 공간으로 볼 수 있고, 각 byte는 자신만의 주소를 가지고 있다.

변수와 주소와의 관계



Pointer 선언, 사용

- Declaration

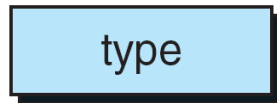
`int *ptr;`

→ 정수형 데이터가 저장된 메모리의 **위치(주소)**를 저장할 수 있는 포인터 변수

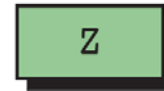
- 포인터의 타입 크기

- 4 byte! (32 bit) ex) int : 4byte, char : 1 byte

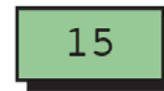
data declaration



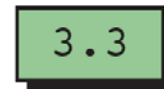
`char a;`



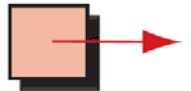
`int n;`



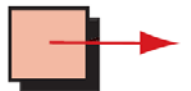
`float x;`



`char* p;`



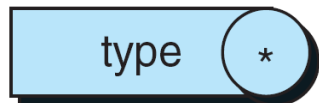
`int* q;`



`float* r;`



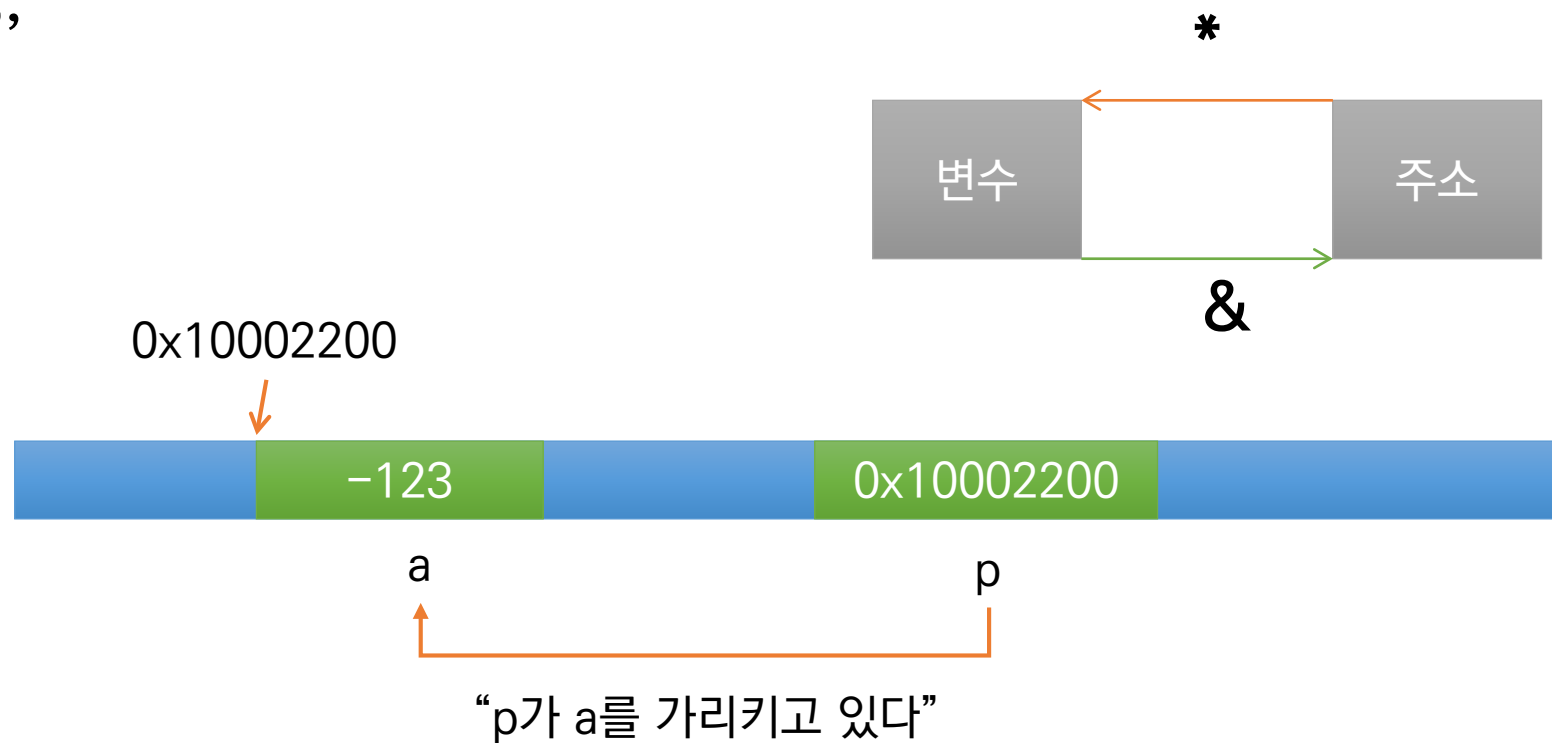
pointer declaration



Pointer 선언, 사용

```
int a = -123;
```

```
int *p = &a;
```



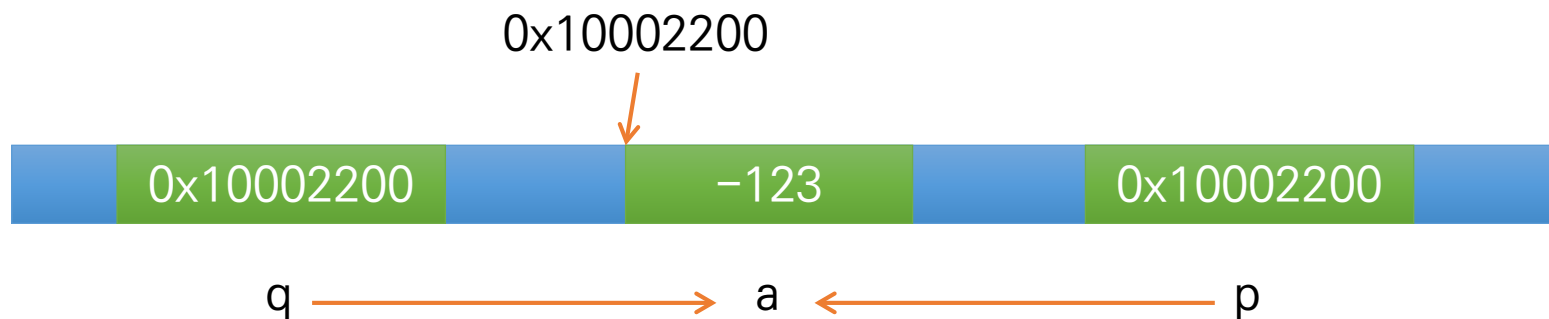
포인터 변수도 결국 '변수'이기 때문에 자신이 가리키는 주소를 메모리 어딘가에 저장하고 있다.

Pointer 선언, 사용

```
int a = -123;
```

```
int *p = &a;
```

```
int *q = p; (or = &a;)
```



“p,q 둘 다 a를 가리키고 있다”

“a 변수의 데이터는 **하나!** a를 가리키는 포인터는 **두 개!**”

예제

```
int a = 14;  
int* p = &a;
```

```
printf("%p %d %d", p, *p, a);  
a = 20;  
printf("%p %d %d", p, *p, a);
```

Results:

00135760 14 14

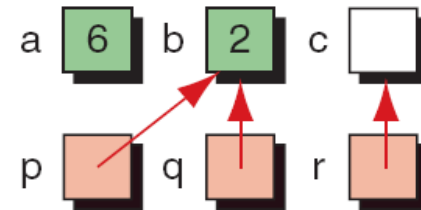
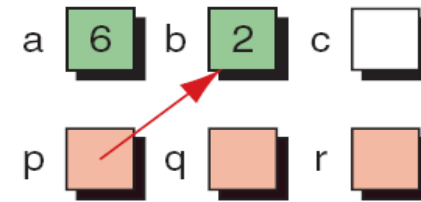
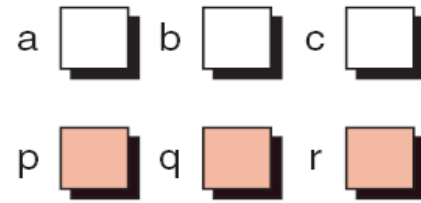
00135760 20 20

Fun with Pointers

```
int a, b, c;  
int* p, q, r;
```

```
a = 6;  
b = 2;  
p = &b;
```

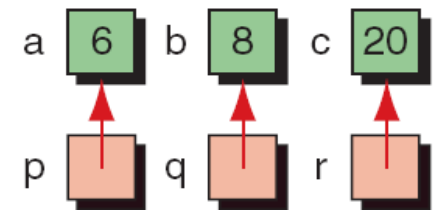
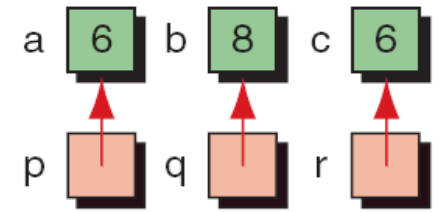
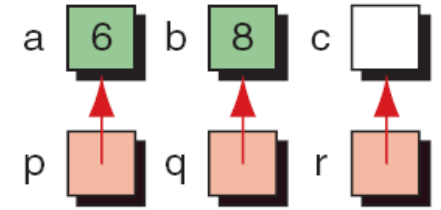
```
q = p;  
r = &c;
```



```
p = &a;  
*q = 8;
```

```
*r = *p;
```

```
*r = a + *q + *&c;
```



NULL 을 가리키는 포인터

- 포인터 변수를 초기화 하려면?

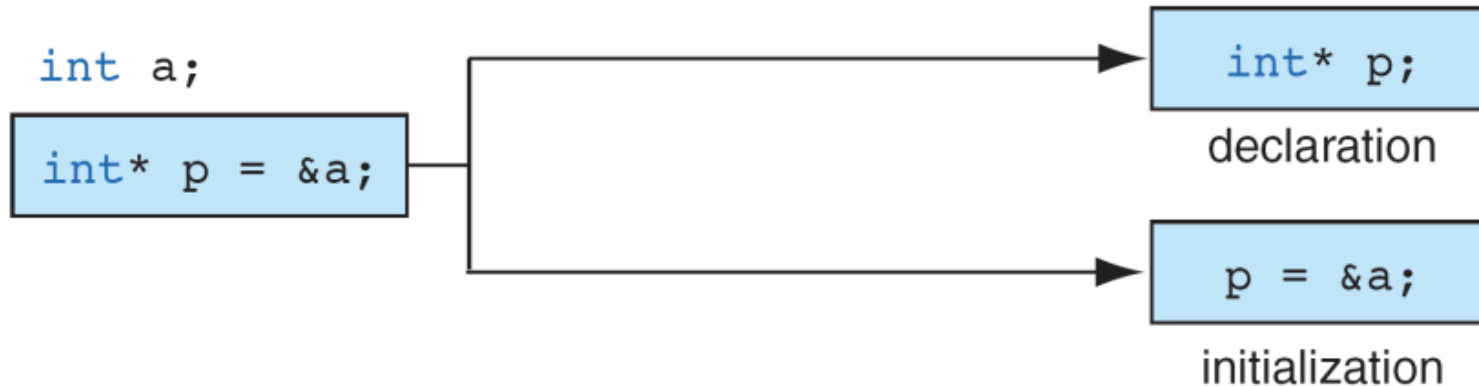
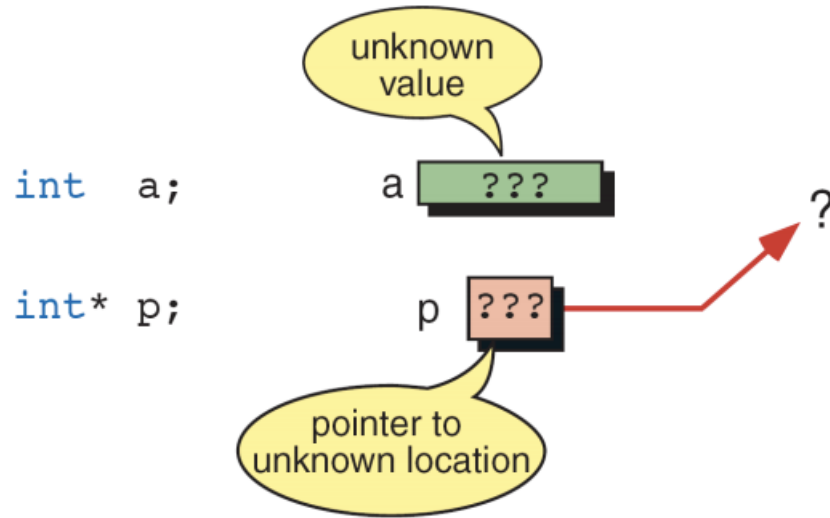


- 아무 수나 넣어서?
 - `int *p = 100;`
 - Segmentation Fault! 혹은 Fatal Error
 - 주소 100의 위치에 뭐가 있는지 모르고, 아무것도 없을 수도...
 - 함부로 건드릴 수 없는 영역에 접근하는 것은 치명적인 위험!
- NULL로 초기화

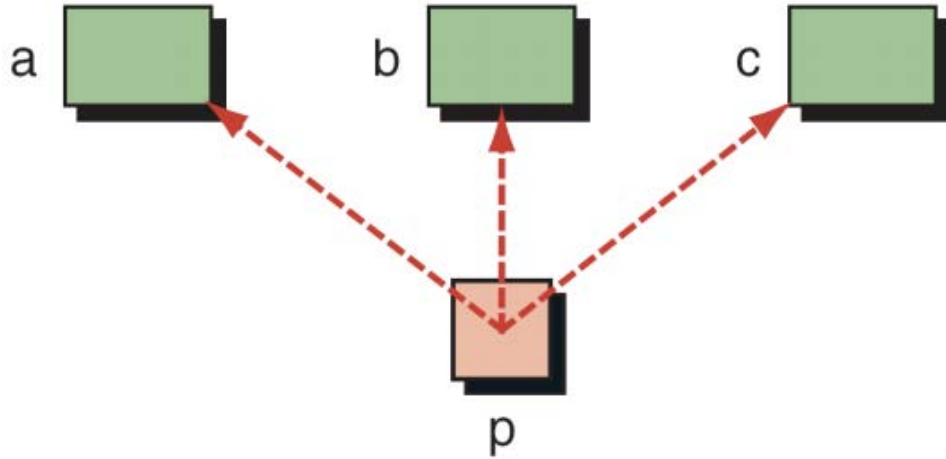
`int *p = NULL;`

활용 : `if (p != NULL)`

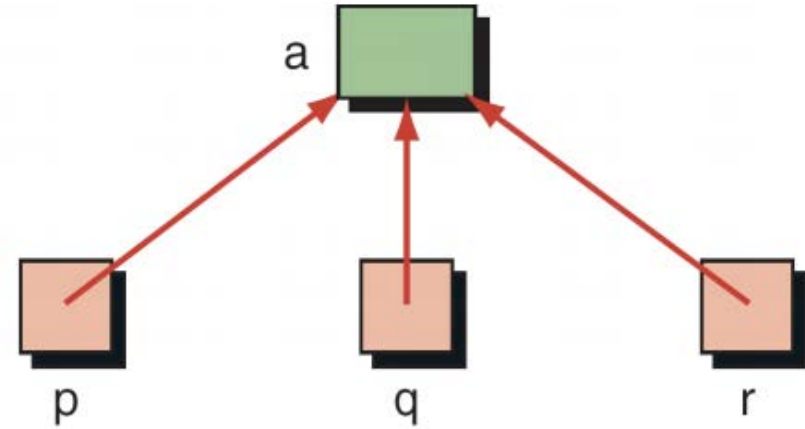
초기화 되지 않은 변수를 사용하면 위험하다



Pointer의 유연성



```
int a, b, c;  
int *p = &a;  
p = &b;  
p = &c;
```

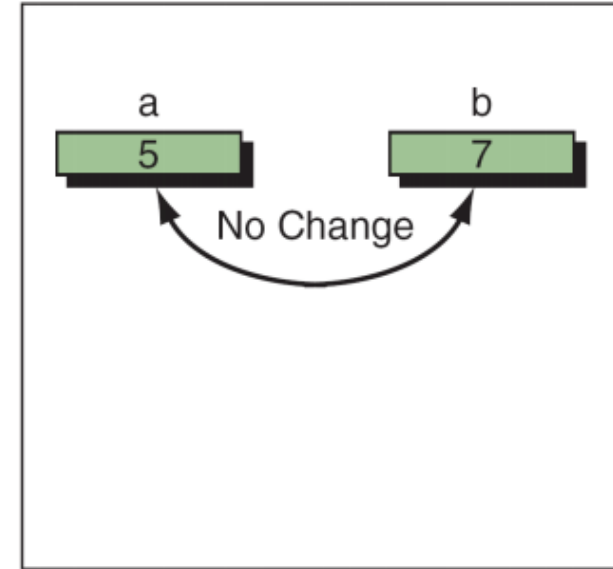


```
int a;  
int *p, *q, *r;  
p = &a;  
q = &a;  
r = &a;
```

Call-by-value

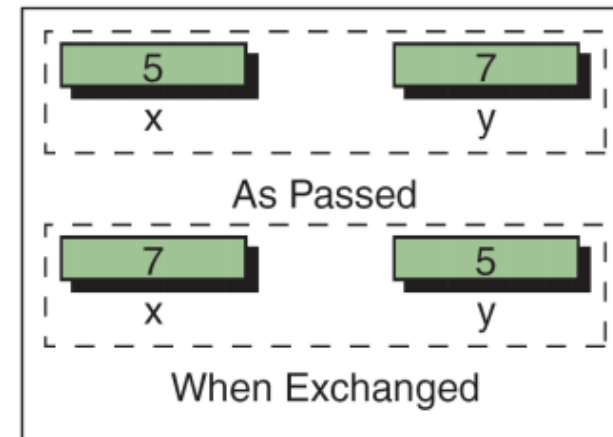
```
// Function Declarations
void exchange (int x, int y);

int main (void)
{
    int a = 5;
    int b = 7;
    exchange (a, b);
    printf("%d %d\n", a, b);
    return 0;
} // main
```



```
void exchange (int x, int y)
{
    int temp;

    temp = x;
    x     = y;
    y     = temp;
    return;
} // exchange
```



Call-by-reference

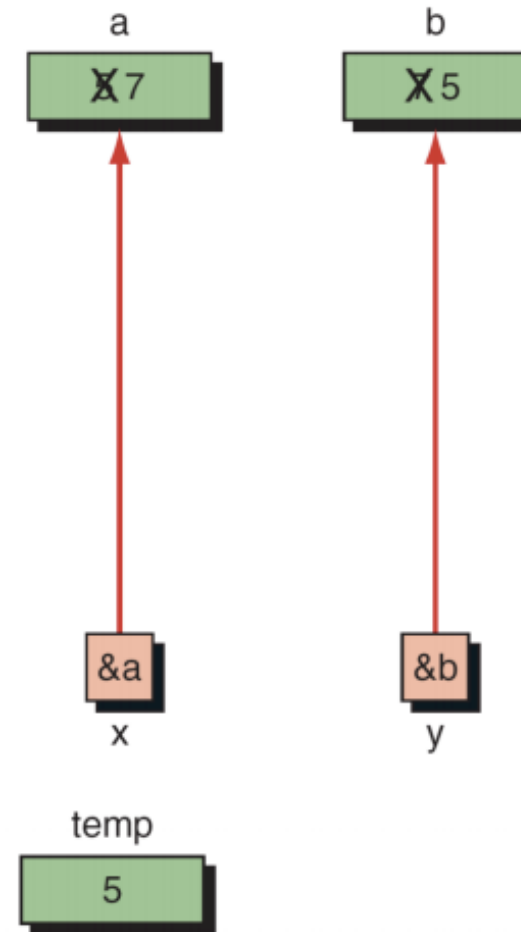
```
// Function Declaration
void exchange (int*, int*);

int main (void)
{
    int a = 5;
    int b = 7;

    exchange (&a, &b);
    printf("%d %d\n", a, b);
    return 0;
} // main
```

```
void exchange (int* px, int* py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
    return;
} // exchange
```

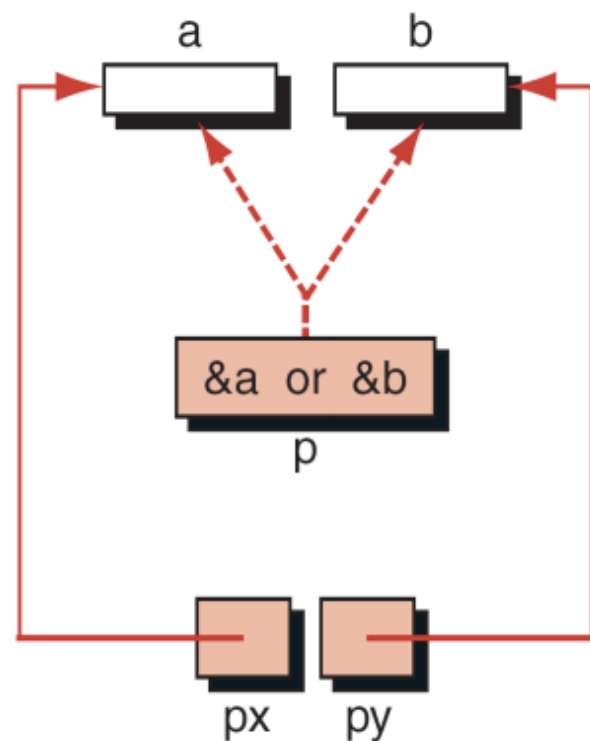


리턴 값으로서의 Pointer

```
// Prototype Declarations
int* smaller (int* p1, int* p2);

int main (void)
...
int  a;
int  b;
int* p;
...
scanf ( "%d %d", &a, &b );
p = smaller (&a, &b);
...
```

```
int* smaller (int* px, int* py)
{
    return (*px < *py ? px : py);
} // smaller
```



함수-Pointer 사용 특징

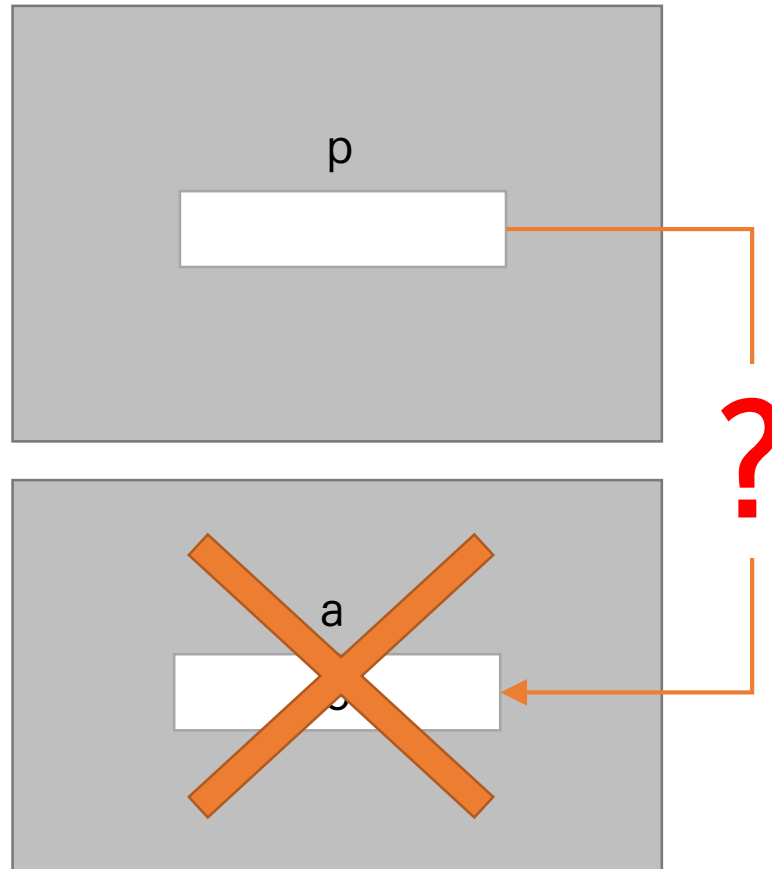
- 포인터 변수도 일반 변수처럼 함수 인자, 반환형으로 사용 가능!
- 함수가 여러 개의 리턴값을 가져야 할 때 - 여러 인자들을 포인터로 받아서 수정할 수 있다!
- 실제 그 메모리 위치를 찾아가 값을 바꿀 수 있다.
- 주의!!
함수 안에서 만들어진 local variable을 가리키는 포인터는 반환할 수 없다.

Local Variable의 주소를 리턴하는 경우

- Local variable는 그 함수(블록)이 끝나면 사라진다.
- 알 수 없는 곳으로의 포인터를 반환하는 것과 같다.

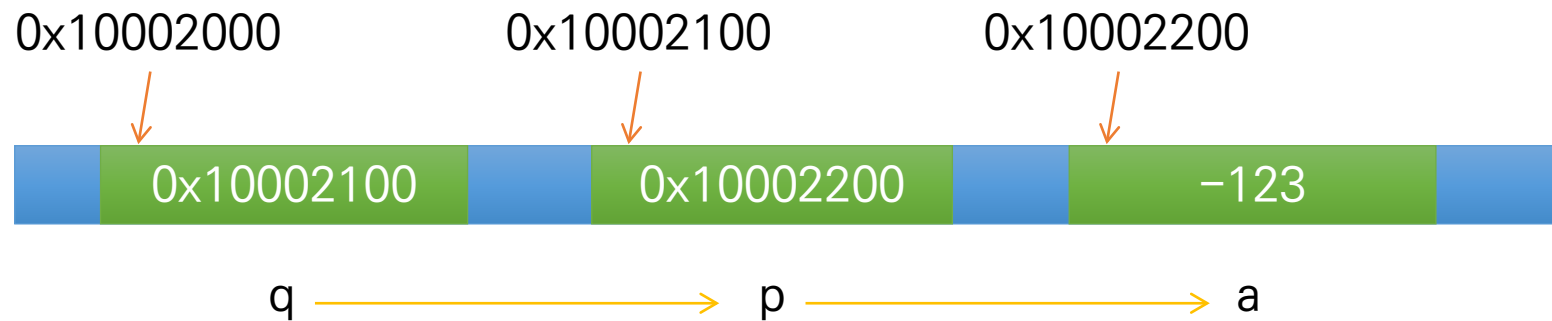
```
int main (void) {  
  
    int* p;  
  
    p = fun();  
    printf("%d", *p);  
}
```

```
int* fun (void) {  
  
    int a = 5;  
  
    return &a;  
}
```



Pointer to Pointer (다중포인터)

- 포인터를 가리키는 포인터!
- 포인터도 주소를 저장하는 하나의 변수! 이기 때문에 메모리 특정 위치에 저장된다.

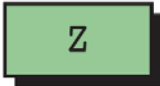


```
int a = -123;  
int *p = &a;  
int **q = &p;
```

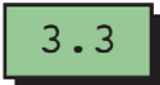
Pointer를 선언할 때 타입이 필요한 이유


`int *p; char *q;`


- 어차피 크기는 4byte(32 bit)로 다 똑같은 거 아닌가?
 - 컴퓨터가 포인터로 메모리에 접근해 데이터를 읽어올 때!
 - 그 포인터 타입에 따라 가져오는 byte 수가 달라진다.
- “int * 포인터면 여기서 부터 4byte 까지 읽어서 정수로 쳐”
→ “char * 포인터면 여기서 부터 1byte 만 읽어서 문자로 쳐”


`char a;` 

`int n;` 

`float x;` 

`char* p;` 

`int* q;` 

`float* r;` 

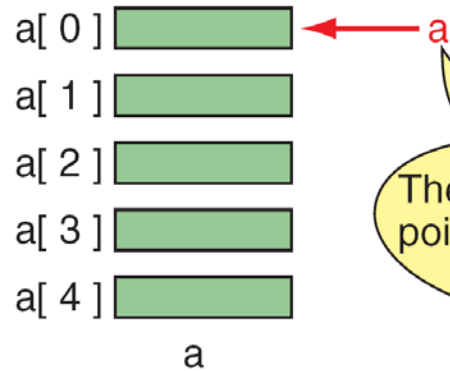
void *

- 어떤 데이터 타입과도 연관되지 않는 일반적 형태의 포인터
 - 어떤 포인터 값도 void pointer에 넣어줄 수 있다.
 - 어떤 포인터 값에도 void pointer 값을 넣어줄 수 있다.
- 단, void pointer는 그 상태 그대로는 dereferencing 할 수 없다!
Why? 타입이 없음 → 내가 보려는 데이터의 타입이 뭐지? → compile error
- **casting**을 통해서 dereferencing 할 수 있다.

```
int a = -123;  
void *p;  
p = &a;  
printf("%d", *(int *)p);
```

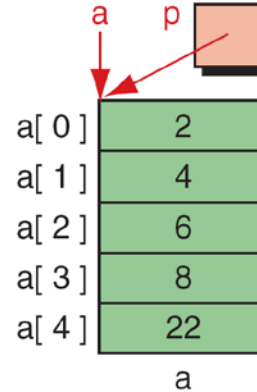
Pointer Application

배열 포인터 (Pointer to Arrays)

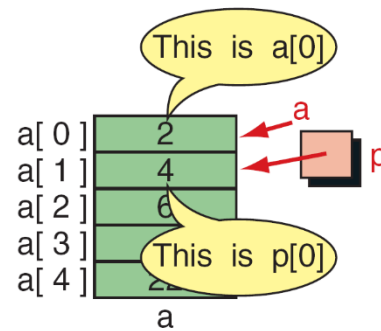
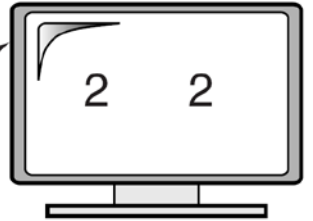


The name of an array is a pointer constant to its first element

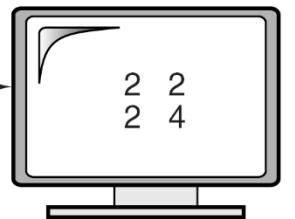
a ↔ &a[0]



```
#include <stdio.h>
int main (void)
{
    int a[5] = {2, 4, 6, 8, 22};
    int* p = a;
    ...
    printf("%d %d\n", a[0], *p);
    ...
    return 0;
} // main
```

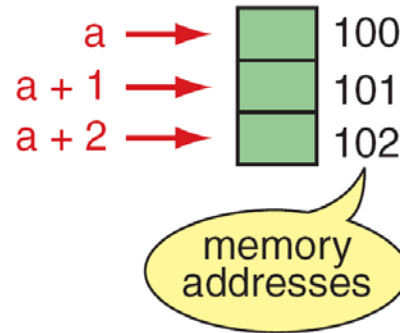
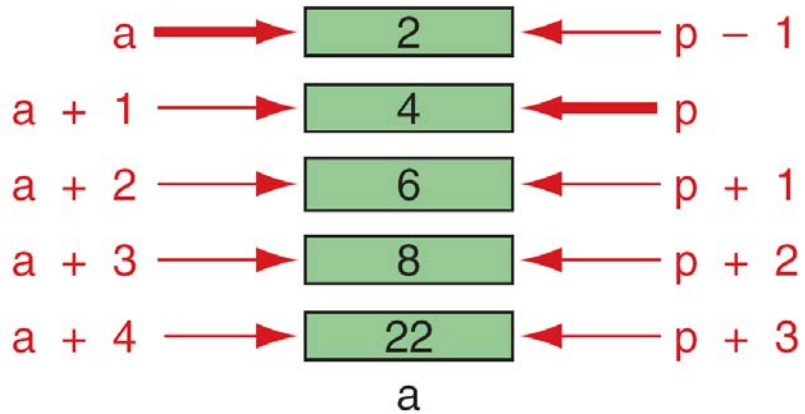


```
#include <stdio.h>
int main (void)
{
    int a[5] = {2, 4, 6, 8, 22};
    int* p;
    ...
    p = &a[1];
    printf("%d %d", a[0], p[-1]);
    printf("\n");
    printf("%d %d", a[1], p[0]);
    ...
} // main
```

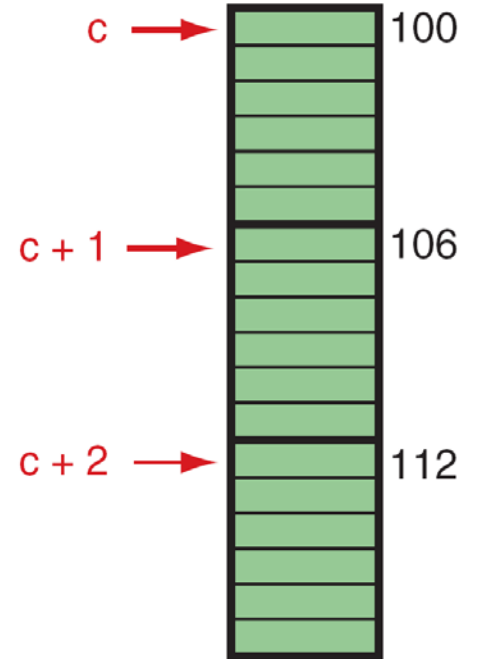
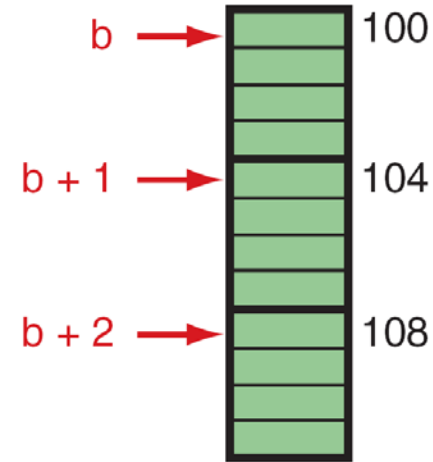


포인터 연산

- 포인터 $p \rightarrow p \pm n \rightarrow p + n * (\text{sizeof}(\text{one element}))$



```
char a[3];  
int b[3];  
float c[3];
```



Arithmetic Operations on Pointers

- $p + 5$, $5 + p$, $p - 5$
- $p1 - p2$
- $p++$, $--p$
- $p1 \geq p2$
- $p1 \neq p2$

Long Form	Short Form
<code>if (ptr == NULL)</code>	<code>if (!ptr)</code>
<code>if (ptr != NULL)</code>	<code>if (ptr)</code>

Pointer Arithmetic






- 두 포인터 p, q에 대하여 가능한 연산
 - p, q 모두 같은 데이터 형을 다루는 포인터여야 한다.
 - $p - q, q - p$: p의 위치와 q의 위치의 차이를 단위수로 계산
 - $(int)p - (int)q, (int)q - (int)p$: p와 q의 주소 값 차이
- * 주의: $p + q$ 는 사용할 수 없다. (무엇보다 의미가 없다)
왜?

$(0x0000 \sim 0xffff) + (0x0000 \sim 0xffff) = ??$

주소값 overflow가 일어날 수 있음

and 꼭 overflow가 아니더라도
사용할 수 있는 메모리 주소는 한정되어 있기 때문

Dereferencing

$a[0]$	or	$*(a + 0)$	2		a
$a[1]$	or	$*(a + 1)$	4		$a + 1$
$a[2]$	or	$*(a + 2)$	6		$a + 2$
$a[3]$	or	$*(a + 3)$	8		$a + 3$
$a[4]$	or	$*(a + 4)$	22		$a + 4$

a

$*(a + i) \longleftrightarrow a[i]$

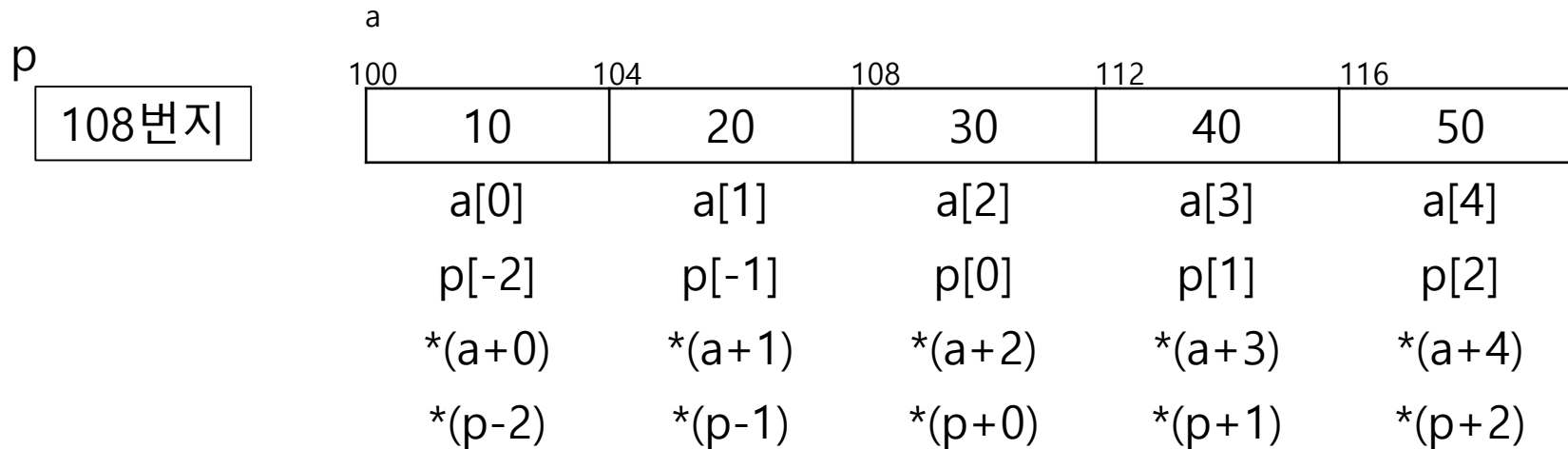
포인터와 배열

```
int a[5] = {10, 20, 30, 40, 50};
```

```
int *p = &a[2];
```

```
printf("%d %d %d %d\n", a[2], *(a+2), p[0], *(p+0)); // 30출력
```

```
printf("%d %d %d %d\n", a[1], *(a+1), p[-1], *(p-1)); // 20출력
```



2차원 Array

`table[2] == *(table+2)`

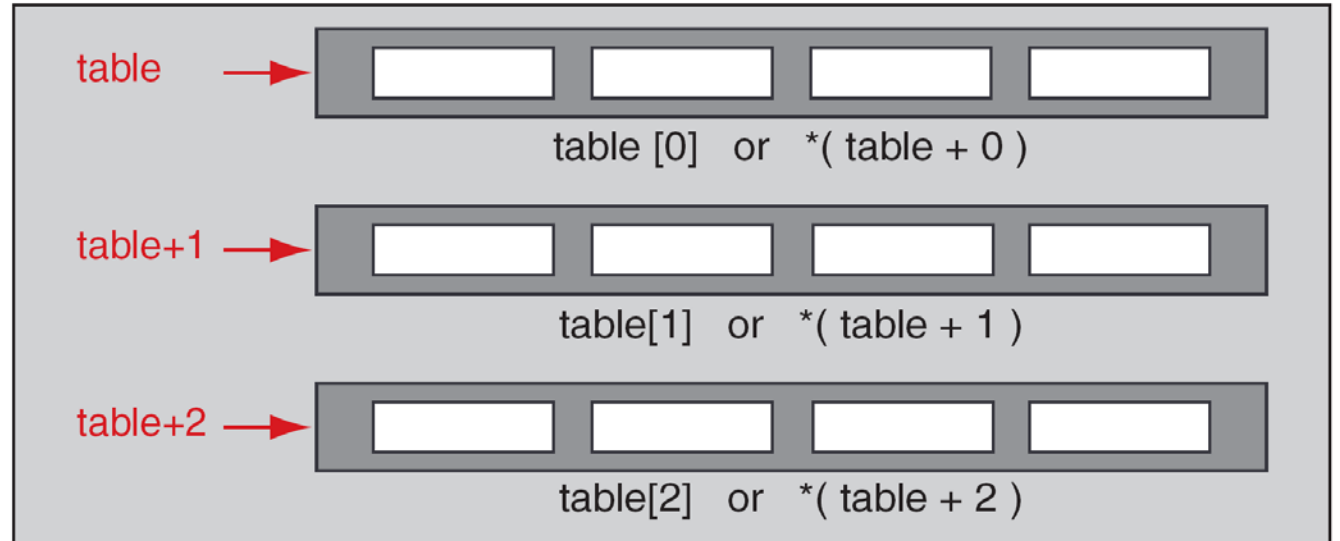
`table[2][1]`

`== (*(table+2))[1]`

`== *(table[2]+1)`

`== (*(table+2)+1)`

- 헷갈리니 다차원 배열에서는 Index를 사용하자



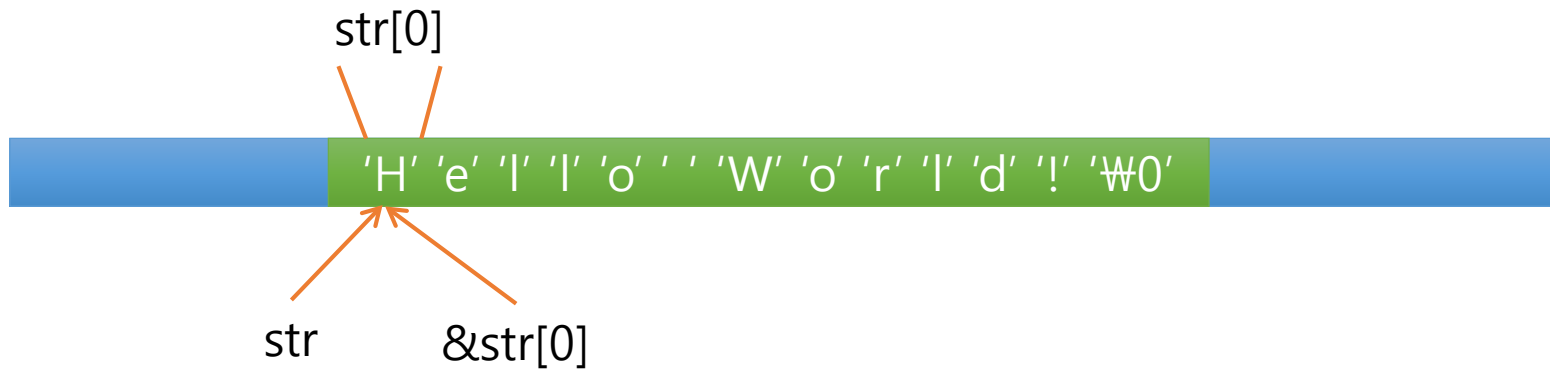
`int table[3][4];`

```
for (i = 0; i < 3; i++)  
{  
    for (j = 0; j < 4; j++)  
        printf("%6d", (*(table + i) + j));  
    printf( "\n" );  
} // for i
```

Print Table

문자열과 포인터

- 다시한번! **여러개의 문자+ '₩0'** 이 들어있는 배열
- 배열의 이름은 첫번째 element의 주소와 같다.



- 그래서 scanf로 %s 받을 때 & 안 붙인다!

문자열 한 줄 다 입력받기

```
char buf[10];
```

- gets(char *buf)
 - gets(buf); //stdin으로 부터 입력 받는다.
//길이 제한 없음 (보안상 취약)
//newline 이나 EOF 만날 때 까지
//끝에 '\0'을 자동으로 붙여준다.
- fgets(char *buf, unsigned size, FILE *file)
 - Fgets(buf, 10, stdin); //지정 stream으로 부터 입력 받는다.
//최대 크기를 지정해 줄 수있다. (널 포함)
//newline 이나 EOF 만날 때 까지
//끝에 '\0'을 자동으로 붙여준다.
//문자는 최대 size-1 개 까지 받는다.

배열 함수에 넘겨주기

- 1차원 배열

int ary1 [10];

void function(int *ary);

void function(int ary []);

- 2차원 배열

int ary2 [5] [6];

void function(int (*ary) [6]);

void function(int ary [] [6]);

- 3차원 배열

(int ary [] [3] [4])

배열 받기 연습

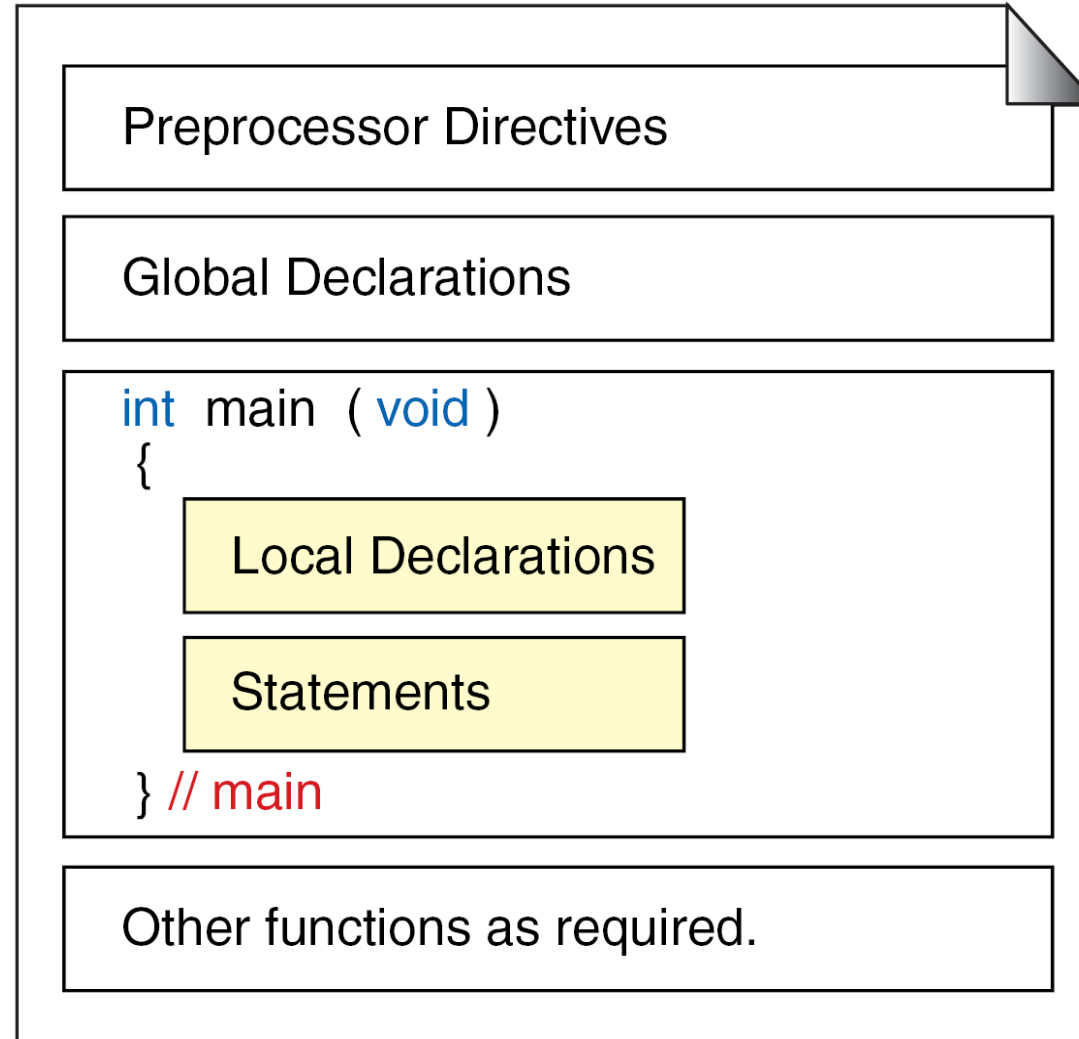
- `int* arr1 [3] ;` \rightarrow `(int** ptr1)`
- `int * arr2 [3] [5];` \rightarrow `(int* (*ptr2) [5])`
- `int** arr3 [5];` \rightarrow ?
- `int *** arr4 [3] [5];` \rightarrow ?

Storage Class

<http://mercury.kau.ac.kr/sjkwon/Lecture/cprogram/C%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D-chap13.pdf>

[http://prof.dongju.ac.kr/syhong/public_html/teaching/c/c08.htm#레지스터 변수 \(register variable\)](http://prof.dongju.ac.kr/syhong/public_html/teaching/c/c08.htm#레지스터 변수 (register variable))

Scopes



Auto Variable

- 일반적으로 지역 변수라 하면 자동 변수를 말함
- 초기화하지 않으면 쓰레기 값이 저장
- 해당 지역변수가 선언된 함수나 블록을 실행하는 시점에서 스택 (Stack)에 할당
- 블록을 종료하는 순간 메모리에서 자동으로 제거

`auto int l = 10; OR int l = 10;`

Auto Variable

```
#include <stdio.h>

int main(void) {
    int cnt;
    for(cnt=0; cnt<3; cnt++) {
        int num=0;
        num++;
        printf("%d번째 반복, 지역변수 num은 %d. \n", cnt+1, num);
    }
    if(cnt==3) {
        int num=7;
        num++;
        printf("if문 내에 존재하는 지역변수 num은 %d. \n", num);
    }
}
```

Static Variable

- 정적 지역변수 vs 정적 전역변수

```
static int global = 10;

int main(void) {
    static int local = 1;
    ...
}
```

- 프로그램이 실행되면 메모리에 할당되고, 프로그램이 종료되면 메모리에서 제거
- 초기값을 지정하지 않으면 자동으로 타입에 따라 '0'이나 '\0(null)'으로 초기화
- 초기값을 저장하면 처음 한번 지정된 초기 값으로 저장

Static Local Variable

- 지역변수 + static
- 참조는 선언된 블록 내부에서만 가능하고, 저장된 값은 블록을 종료해도 메모리에서 제거되지 않고 계속 남아 있다.
- 참조의 범위(scope)는 지역변수와 같으나, 메모리의 생명력은 프로그램이 종료되어야 종료되는 전역변수와 같은 특징
- 초기값을 지정하는 경우 초기화는 첫 번째 호출에서만
→ 두 번째 호출부터는 초기화가 이루어지지 않는다!

Static Local Variable

```
#include <stdio.h>

void sub();

int main(void) {
    int i;
    for (i = 0; i < 3; i++) {
        sub();
    }
}

void sub() {
    static int i = 1;
    int k = 3;
    printf("i=%d\t k= %d\n", i++, k++);
}
```

```
#include <stdio.h>

void sub();

int main(void) {
    int i;
    for (i = 0; i < 3; i++) {
        sub();
    }
}

void sub() {
    static int i;
    i = 1;
    int k = 3;
    printf("i=%d\t k= %d\n", i++, k++);
}
```

Static Global Variable

- 전역변수 + static
- 정적 전역변수 vs 전역변수
- 전역변수는 파일 소스가 달라도 vs 정적 전역변수는 동일한 파일에서만

main.c

```
static int global = 10;
int sub(void);

int main(void) {
    sub();
    ...
}
```

sub.c

```
static int global;

int sub(void) {
    int a = 10;
    a = a - global;
    ...
}
```

Extern Variable

- 블록이나 함수 내부에서 이미 전역 변수로 선언된 변수를 사용하고자 할 때
- 언제 쓰나요?
- 전역 변수가 선언된 위치가 이 변수를 사용하려는 블록이나 함수의 위치보다 나중에 위치할 때
- 소스 파일이 분리된 경우 다른 파일에서 선언된 전역 변수를 이용하기 위해

Extern Variable

```
int main(void)
{
    extern int gCount; //외부 전역 변수 알림
    //외부 전역 변수 gCount 알리는 위 문장을 생략하면 에러
    ...
    printf("%d", gCount);
    ...
    return 0;
}

int gCount = 100 ;
```

변수 gCount는 main() 함수 외부에서 정의된 전역 변수임을 알리지 않으면 에러가 발생한다.

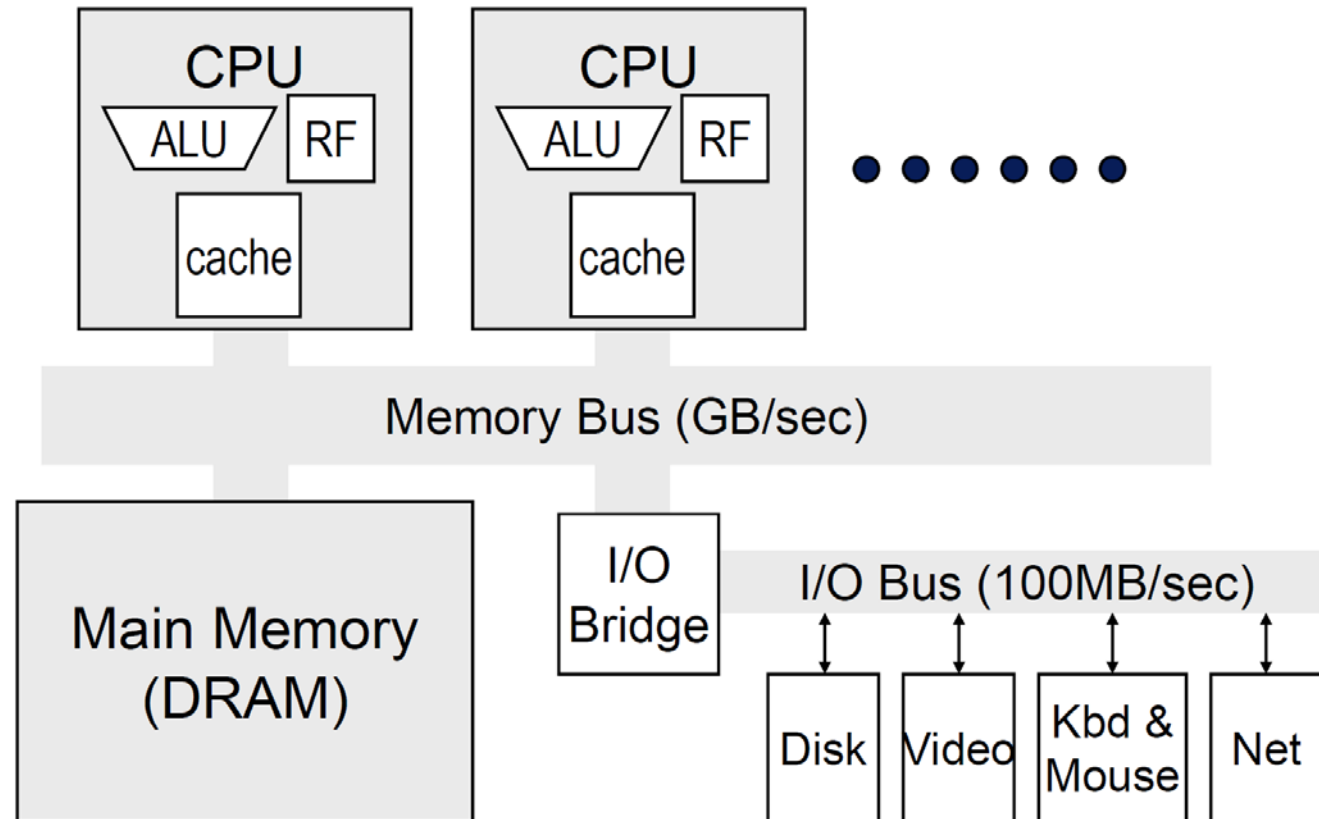
Extern Variable

```
#include <stdio.h>
int x;
main()
{
    x = 10;
    sub1();
    sub2();
    sub3();
}
sub1()
{
    x++;
    printf("sub1의 x값 = %d\n", x);
}
```

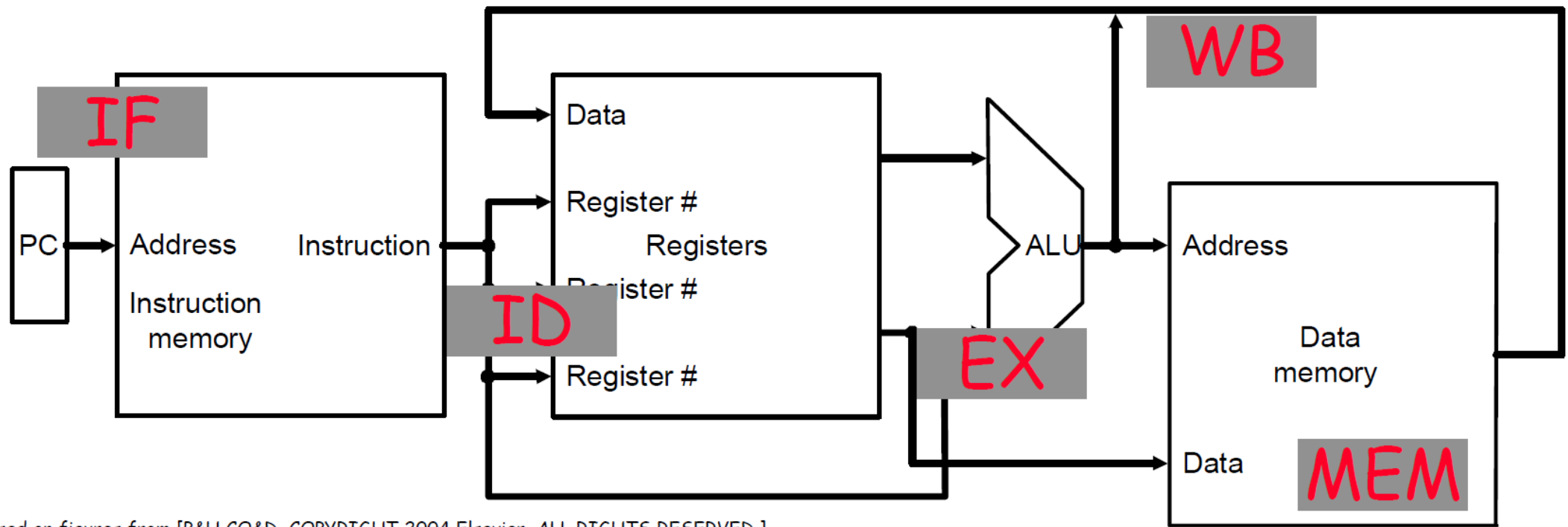
```
#include <stdio.h>
sub2()
{
    extern int x;
    ++x;
    printf("sub2의 x값 = %d\n", x);
}
sub3()
{
    int x = 50 ;
    printf ("sub3의 x값 = %d\n", x);
}
```

Register Variable

- 변수의 저장공간을 메모리가 아닌 레지스터에 할당



CPU의 동작 원리



**Based on figures from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

어셈블리 코드

- ◆ E.g. High-level Code

$$A[8] = h + A[0]$$

where **A** is an array of integers (4-byte each)

◆ Assembly Code

- suppose $\&A, h$ are in r_A, r_h
- suppose r_{temp} is a free register

```
LW r_temp 0(r_A)      # r_temp = A[0]
add r_temp r_h r_temp  # r_temp = h + A[0]
SW r_temp 32(r_A)      # A[8] = r_temp
                        # note A[8] is 32 bytes
                        # from A[0]
```


Register Variable

- 변수의 저장공간을 메모리가 아닌 레지스터에 할당
- 지역변수에만 사용 가능
- 함수나 블록이 시작되면서 CPU의 내부 레지스터에 값이 저장
- 왜 쓰나요?
CPU 내부의 임시 기억장소인 레지스터에 할당되어 변수 값을 저장하므로 입출력 속도가 빠르기 때문 (ex. 반복문의 횟수를 검사: i)
- 레지스터의 수가 한정되어 있으므로 레지스터가 모자라면 레지스터 변수로 선언하더라도 일반 지역변수로 할당

Register Variable

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

```
int main(void)
{
    register int i, sum=0 ;
    for(i=0; i<=100 ; i++)
    {
        sum += i;
        printf(" 1부터 100까지의 합 = %d\n ", sum) ;
    }
}
```

Storage Class (변수의 존재기간과 접근범위)

	지역변수	정적변수	전역변수	register변수
지정자	auto	static	extern	register
저장장소	스택	정적 데이터 영역	정적 데이터 영역	레지스터
선언위치	함수내부	함수내부/외부	함수내부/외부	함수내부
유효범위	함수내부	함수내부/외부	프로그램 전체	함수내부
생존기간	함수종료시	프로그램 종료시	프로그램 종료시	함수종료시
초기값	초기화 안됨	0으로 초기화	0으로 초기화	초기화 안됨

언제 어떤걸 쓰나요

- 일반적으로 전역변수의 사용을 자제하고, 지역변수인 자동변수를 이용
- 실행속도를 빠르게 하고 싶을 때: 레지스터 변수
- 함수나 블록 내부에서 계속 값을 저장하고 싶을 때: 정적 지역변수
- 해당 파일 내부에서만 변수를 공유하고 싶을 때: 정적 전역변수
- 프로그램의 모든 영역에서 값을 공유하고 싶을 때: 전역 변수
- 전역변수는 모든 함수에서 공유할 수 있는 저장공간을 이용할 수 있는 장점이 있지만 어디선가 잘못 바꾸면 프로그램 전체에 영향을 미친다→ 위험해
- 함수의 인자(argument)로 선언된 변수는 지역변수와 같이 함수 내부에서만 유효

다음시간

- Dynamic Memory Allocation