

C Programming Language

(11th class)

Dohyung Kim

Assistant Professor @ Department of Computer Science

Today ...

- **Revisit Pointer**

```
#include <stdio.h>

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_ptr(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void) {
    int n1, n2;
    printf("Enter two numbers : ");
    scanf("%d %d", &n1, &n2);
    printf("You have entered n1 = [%d] and n2 = [%d]\n", n1, n2);
    swap(n1, n2);
    printf("After swap,\n\tn1 = [%d] and n2 = [%d]\n", n1, n2);
    swap_ptr(&n1, &n2);
    printf("After swap_ptr,\n\tn1 = [%d] and n2 = [%d]\n", n1, n2);
    return 0;
}
```

```
#include <stdio.h>

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap_ptr(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void) {
    int n1, n2;
    printf("Enter two numbers : ");
    scanf("%d %d", &n1, &n2);
    printf("You have entered n1 = [%d] and n2 = [%d]\n", n1, n2);
    swap(n1, n2);
    printf("After swap,\n\tn1 = [%d] and n2 = [%d]\n", n1, n2);
    swap_ptr(&n1, &n2);
    printf("After swap_ptr,\n\tn1 = [%d] and n2 = [%d]\n", n1, n2);
    return 0;
}
```

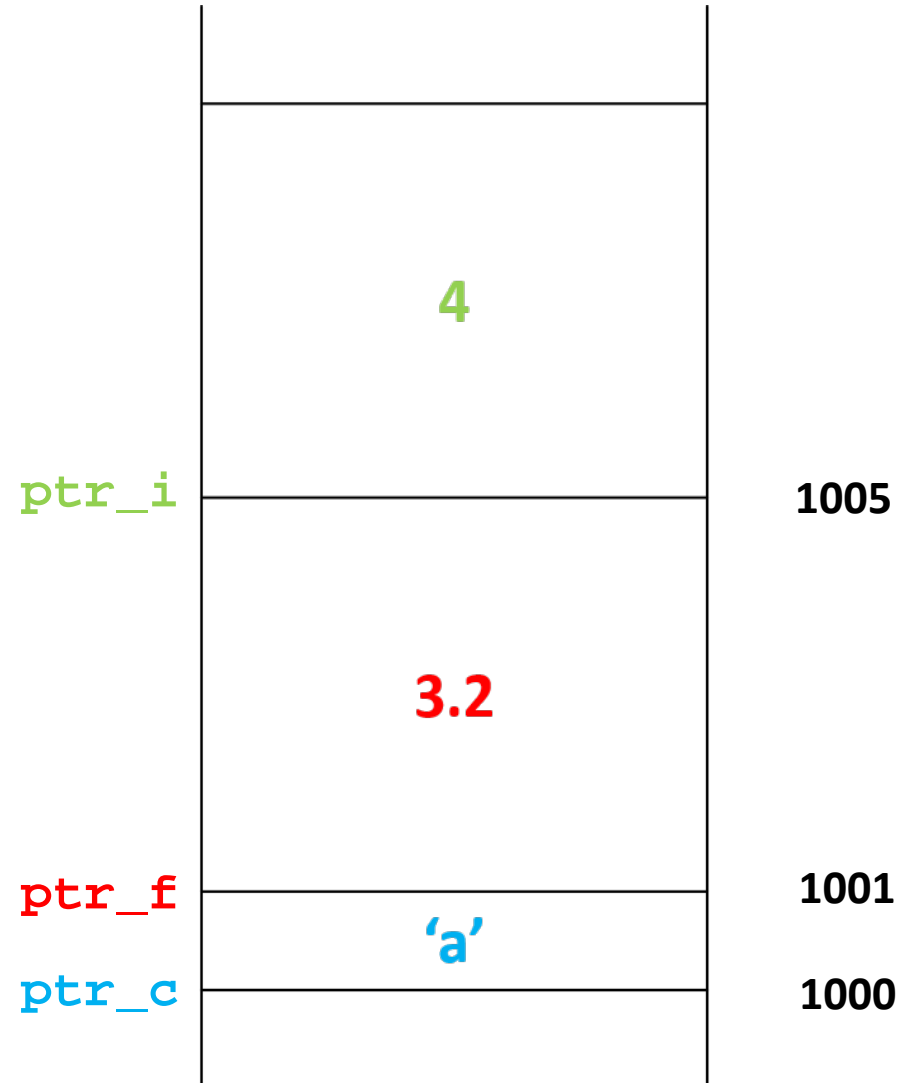
Variables and Addresses

- 변수가 저장되기 위해서는 메모리에서의 공간(몇몇 byte) 이 필요함
- 변수의 주소값은 메모리 상에서 할당된 공간의 시작 주소값임 (first byte 주소값)

```
char c = 'a';  
float f = 3.2;  
int i = 4;  
  
char *ptr_c = &c;  
float *ptr_c = &f;  
int * ptr_i = &i;
```

Storing Variables in Memory

```
char c = 'a';  
float f = 3.2;  
int i = 4;  
  
char *ptr_c = &c;  
float *ptr_f = &f;  
int *ptr_i = &i;
```



Pointer Variables

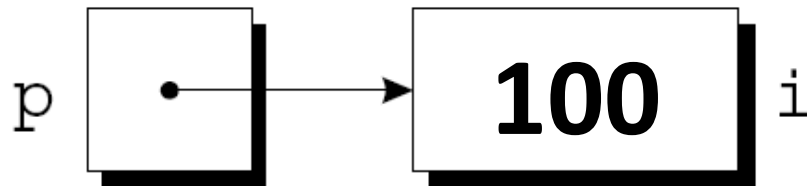
■ Pointer

- 메모리 공간에서의 주소값을 저장하기 위한 데이터 타입
- 포인터 변수는 데이터 (변수의) 주소값을 저장할 수 있음

- p라는 포인터 변수가 변수 i의 주소값 저장하고 있을 경우,
p “points to” i 라고 이야기할 수 있음

- A graphical representation:

```
int i = 100;  
int * p = &i;
```



Pointer Variables

- 포인터 변수 선언시, 반드시 *가 반드시 변수 이름앞에 위치해야 함.

```
int *p;
```

- **p** 는 int 타입의 변수를 가리키도록 선언되었음. 다만 선언만 되었고 변수에 내용(data)이 할당이 되지 않았기 때문에 현재는 아무것도 가리키고 있지 않음

- 포인터를 사용하기 전에 초기화하는 것은 매우 중요함

- 가능하면 선언하면서 NULL(0)로 초기화하는 것을 권장함.

```
int *p = 0;
```

- 포인터 타입의 **conversion specification**은 **%p**.

The Address Operator

- 포인터 변수를 초기화하는 방법으로 특정 변수의 주소값을 할당할 수 있음:

```
int i, *p;
```

```
...
```

```
p = &i;
```

- & (ampersand) 는 “address of” (주소를 나타내는) 연산자임
 - Or a reference operator
- "p = &i" → p변수에 i변수의 주소값을 할당한다는 의미임
 - p 는 i를 가리킴

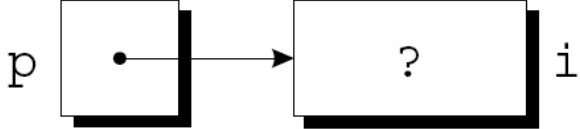
The Dereference Operator

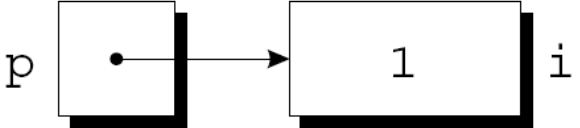
- 포인터 변수가 다른 변수를 가리키게 되면, * (dereference) 연산자를 통해서 해당 포인터 변수가 가리키고 있는 변수의 데이터에 접근할 수 있음
- 만약 포인터 변수 `p` 가 `i`를 가리킬 경우, `i` 변수에 담겨있는 내용을 다음과 같이 화면에 출력할 수 있음

```
int i = 100;  
int *p = &i;
```

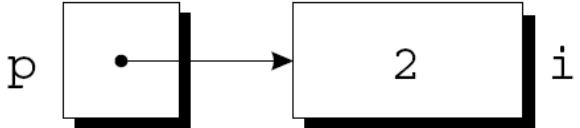
```
printf("%d\n", *p);
```

The Dereference Operator Examples

`p = &i;` 

`i = 1;` 

```
printf("%d\n", i);      /* prints 1 */
printf("%d\n", *p);     /* prints 1 */
```

`*p = 2;` 

```
printf("%d\n", i);      /* prints 2 */
printf("%d\n", *p);     /* prints 2 */
```

The Dereference Operator

- * (dereference operator)를 초기화 하지 않은 포인터 변수 앞에서 사용할 경우 예상치 못한 결과를 얻을 수 있음

```
int *p;  
printf("%d", *p);    /*** WRONG ***/
```

- * (dereference operator)를 사용하여 초기화 하지 않은 포인터 변수에 값을 저장하는 것은 위험할 수 있음

```
int *p;  
*p = 1;              /*** WRONG ***/
```

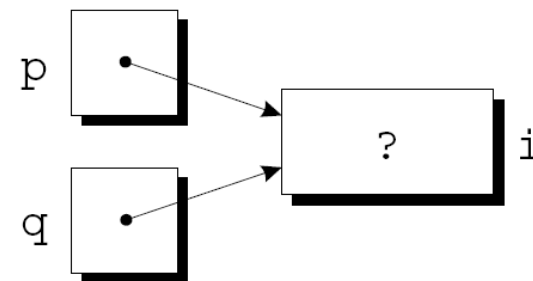
Pointer Assignment

■ 포인터 할당의 예:

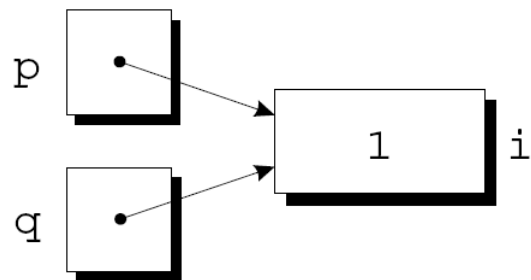
```
int i, *p, *q;
```

```
p = &i;
```

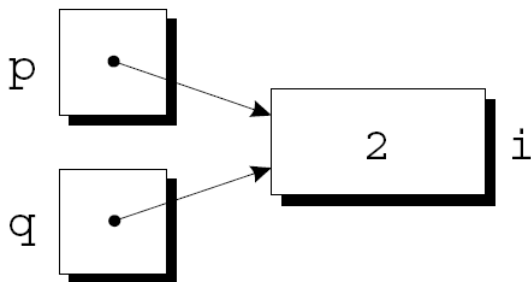
```
q = p;    /* q now points to the same place as p */
```



```
*p = 1;
```

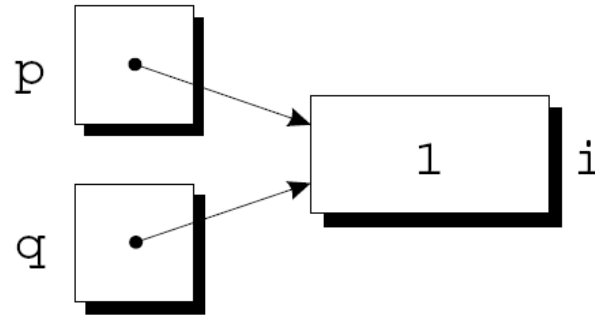


```
*q = 2;
```

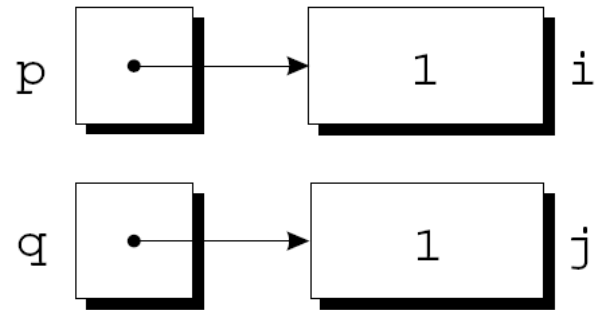


`q = p;` vs `*q = *p;`

```
i = 1; j = 2;  
p = &i;  
q = p;
```



```
p = &i;  
q = &j;  
*q = *p;
```



Errors

- 해당 코드에서 어떤 점들이 잘못되었을까?

```
int x, *p;  
x = 10;  
*p = x;
```

```
int x, *p;  
x = 10;  
p = x;
```

```
int x;  
char *p;  
p = &x;  
printf("%d\n", *p);
```

```
int x;  
int *p;  
p = &x;  
printf("%d\n", p);
```

Passing Arguments to Functions

- Call-by-value : 변수의 **값**들을 함수의 인자(argument, parameter)로 넘겨주는 것
 - 함수 호출시 넘겨주는 **값**이 함수의 지역변수로 **복사**됨
 - call-by-value는 C programming에서 기본적인 파라미터 전달 방법임
- Call-by-reference : 변수의 **주소**를 함수의 인자로 넘겨주는 것
 - 함수 호출시 포인터 값을 인자로 사용하게 됨
 - 이 경우 **주소값**이 **복사**되어 함수의 지역변수에 저장되기 때문에, 주소값이 가리키는 곳에 직접 접근 가능함


```
#include <stdio.h>

void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

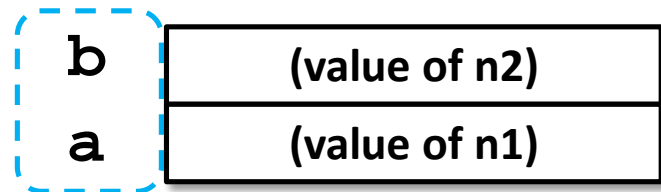
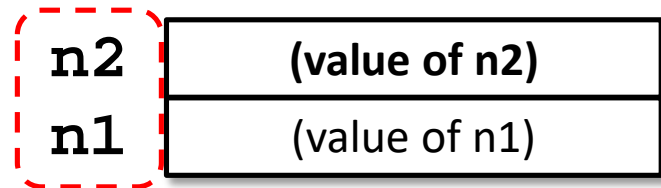
void swap_ptr(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void) {
    int n1, n2;
    printf("Enter two numbers : ");
    scanf("%d %d", &n1, &n2);
    printf("You have entered two numbers n1 = [%d] and n2 = [%d]\n", n1,
n2);

    swap(num1, num2);
    printf("After swap,\n\tn1 = [%d] and n2 = [%d]\n", n1, n2);
    swap_ptr(&num1, &num2);
    printf("After swap_ptr,\n\tn1 = [%d] and n2 = [%d]\n", n1, n2);
    return 0;
}
```

swap in detail

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main(void) {  
    int n1, n2;  
    ...  
    swap(n1, n2);  
    ...  
    return 0;  
}
```



Caller's Variable Scope

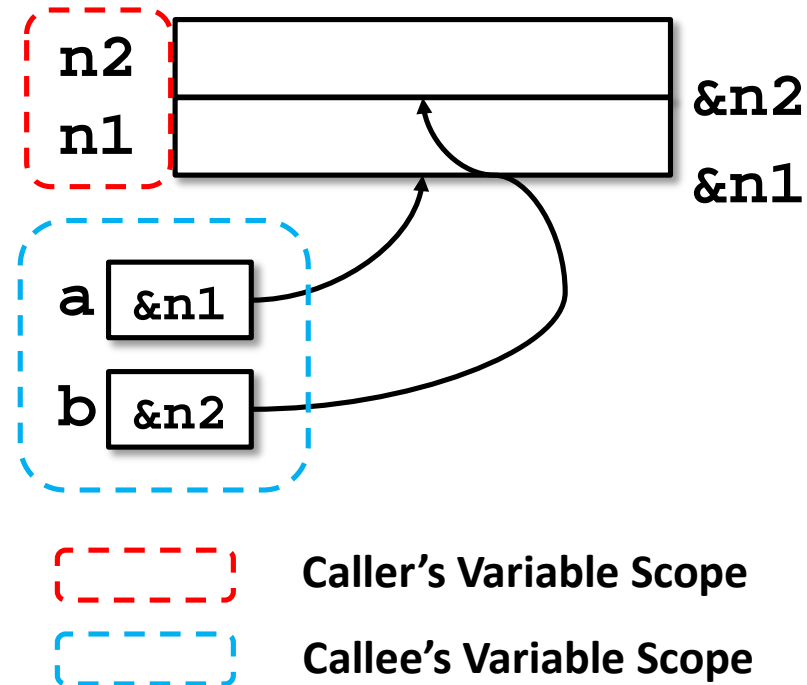


Callee's Variable Scope

- swap 함수안에서, a 와 b 변수의 값이 교환됨. 그러나 a, b변수 모두 지역변수로서 함수가 return됨과 동시에 사라짐
- n1 와 n2 변수의 값에는 변화가 없음

swap in detail

```
void swap_ptr(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
int main(void) {  
    int n1, n2;  
    ...  
    swap_ptr(&n1, &n2);  
    ...  
    return 0;  
}
```



- `n1` 과 `n2` 변수의 주소값이 인자로 전달됨
- `swap_ptr` 함수의 지역변수 `a`와 `b`에는 `n1` 과 `n2`의 주소값이 저장되어 있기 때문에 `a, b`를 활용해서 `n1` 과 `n2`의 내용이 변경 가능함

Exercise : Pointer Variable Declaration

- 변수 num의 주소값을 화면에 출력하시오.

```
void ex1()
{
    int num = 10;

    printf("%d\n", num);
    /*
    Fill in here
    */
}
```

Exercise - Address of Pointer Variables

- `ex2()` 함수 호출의 결과가 오른쪽 상자에 표시되어 있다. 해당 결과를 설명하시오. (세번째 결과는 어떻게 나온 것일까?)

```
void ex2()
{
    int num = 10;

    printf("%d\n", num);
    printf("%p\n", &num);
    printf("%d\n", &num);
}
```

```
[Result]
10
00FDF9D4
16644564
```

Pointers as Return Values

```
int *max(int *a, int *b)
{
    if (*a > *b)
        return a;
    else
        return b;
}
```

→ 함수의 return 타입이 int*

Pointers as Return Values

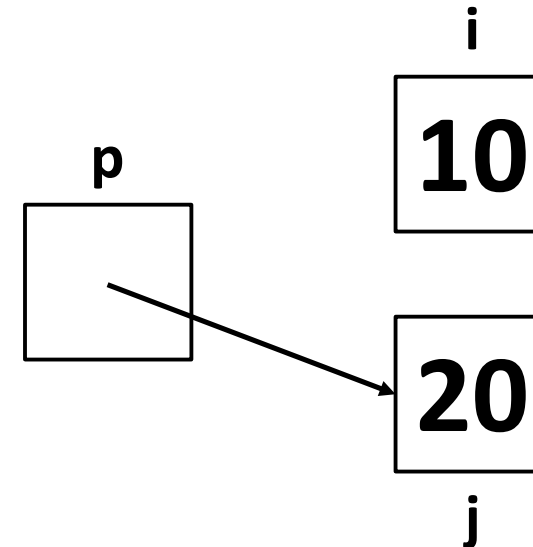
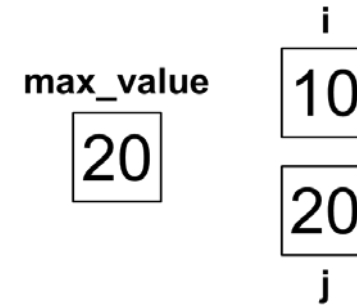
■ A call of the max function:

```
int *p_max, i, j;  
...  
p_max = max(&i, &j);
```

함수 호출 후, p 는 i or j 변수를 가리키게 됨

call by value

```
int max_value, i, j;  
...  
max_value = max(i, j);
```



Example 1

- 앞의 max 함수를 사용하여 실행해 보시오.

```
int main()
{
    srand(0);
    int num1 = rand() % 100, num2 = rand() % 100;
    printf("num1: %d\n", num1);
    printf("num2: %d\n", num2);
    printf("max: %d\n\n", max(num1, num2));

    int *max = max_pointer(&num1, &num2);
    int max_value = *max;
    *max = 0;
    printf("num1: %d\n", num1);
    printf("num2: %d\n", num2);
    printf("max: %d\n", *max);
    printf("max: %d\n", max_value);

}
```


Example 2

- 포인터를 활용하여 함수로부터 여러개의 결과값을 받아올 수 있음

- scanf

```
int main()  
{  
    int num1, num2, num3;  
    scanf("%d %d %d", &num1, &num2, &num3);  
    printf("%d %d %d", &num1, &num2, &num3);  
}
```

Exercise

- 배열의 최대, 최소값을 return하는 함수를 작성하시오

```
void min_max(int number[], int size, <parameters>)  
{
```

<Source Code>

```
}
```

Pointers to Pointers

■ 포인터 변수도 변수임

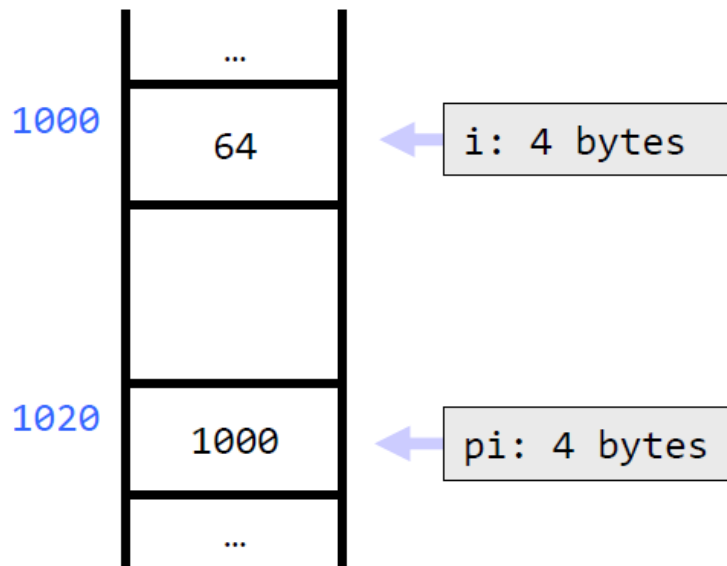
- 포인터 변수도 자신의 주소값을 가지고 있음
- 포인터 변수를 가리키는 포인터도 가능함

```
#include <stdio.h>

int main(void)
{
    int i = 64;
    int *pi = &i;

    printf("%d\n", i);
    printf("%d\n", *pi);

    printf("%p\n", &i);
    printf("%p\n", pi);
    printf("%p\n", &pi);
}
```



Pointers to Pointers

■ 포인터변수를 가리키는 포인터

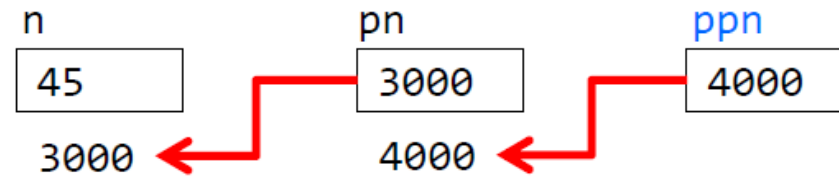
- Double (**)를 사용
- * 변수는 변수의 주소값을 저장하는데 사용됨
- ** 변수는 포인터 변수의 주소값을 저장하는데 사용

■ Declaration Example

```
int **ptr2ptr;
```

Pointers to Pointers

```
int void main()
{
    int n = 45, *pn, **ppn;
    pn = &n;
    ppn = &pn;
}
```

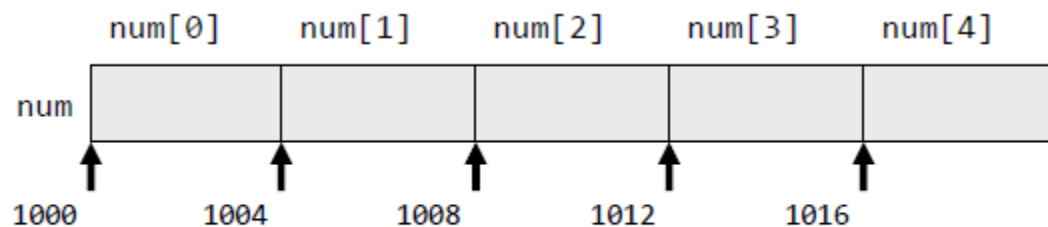


Statement	Output
*pn	45
**ppn	45
pn	3000 (= &n)
ppn	4000 (= &pn)

Array vs. Pointer

■ 각 배열의 원소의 주소값

```
int num[5];
```



```
&num[0] == 1000  
&num[1] == 1004  
&num[2] == 1008  
&num[3] == 1012  
&num[4] == 1016
```

Array vs. Pointer

■ 배열 이름은 포인터와 호환 가능

```
int main(void)
{
    int a[10] = {10}, *pa;

    pa = a;
    printf("%p %p %p\n", a, pa, &a[0]);
    printf("%d %d %d\n", *a, *pa, a[0]);

    return 0;
}
```

Pointer Arithmetic

- 포인터 연산은 C의 우수한 특징중 하나임
- 주소값을 쉽게 조작할 수 있도록 해줌
 - 배열의 원소를 가리키는 포인터에 대한 +/- 연산 이 가능함
 - p 가 배열 a 의 원소를 가리키고 있을 때, p 에 대한 포인터 연산을 통해서 a 의 다른 원소들에 대한 접근이 가능함

Run & Observe

```
void run_and_observe()
{
    int a[3] = { 1,2,3 }, *b = &a[0], *c = a;

    printf("FIRST CASE\n");
    for (int i = 0; i < 3; i++)
        printf("a[%d]:%2d, *(b+%d):%2d, *(c+%d):%2d\n", i, a[i], i, *(b + i), i, *(c + i));
    for (int i = 0; i < 3; i++)
        printf("&a[%d]:%p,\t(b+%d):%p,\t(c+%d):%p\n", i, &a[i], i, (b + i), i, (c + i));

    char ch[3] = { 'a', 'b', 'c' }, *p_ch = ch;

    printf("SECOND CASE\n");
    for (int i = 0; i < 3; i++)
        printf("ch[%d]:%2d, *(p_ch+%d):%2d\n", i, ch[i], i, *(p_ch + i));
    for (int i = 0; i < 3; i++)
        printf("&ch[%d]:%p,\t(p_ch+%d):%p\n", i, &ch[i], i, (p_ch + i));
}
```

Increment of Pointers

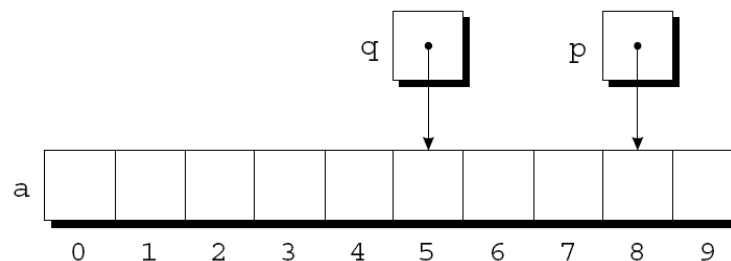
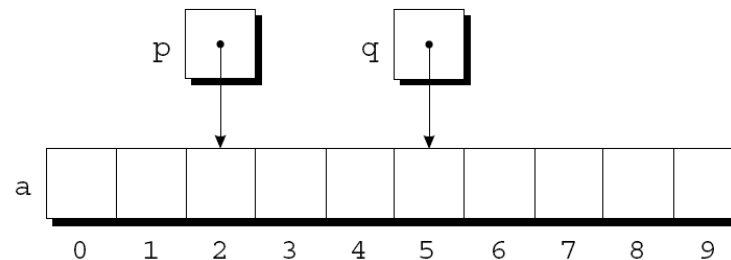
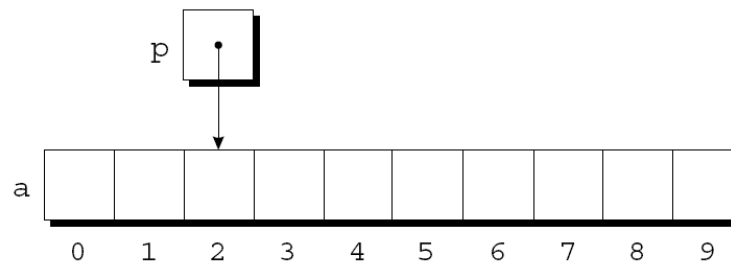
- 포인터 변수와 함께 사용되는 “+”의 뜻은 포인터를 다음 원소로 옮기라는 뜻임

```
int a[10], *p, *q;
```

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

```
q = p + 3;
```

```
p += 6;
```



Decrement of Pointers

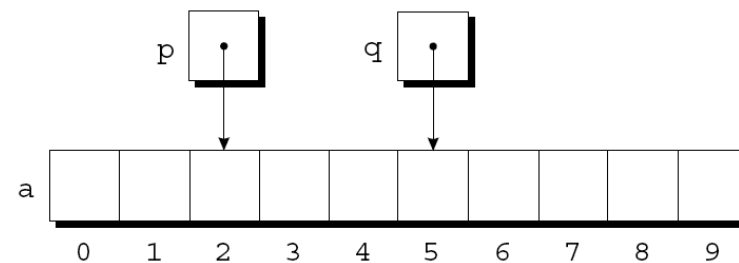
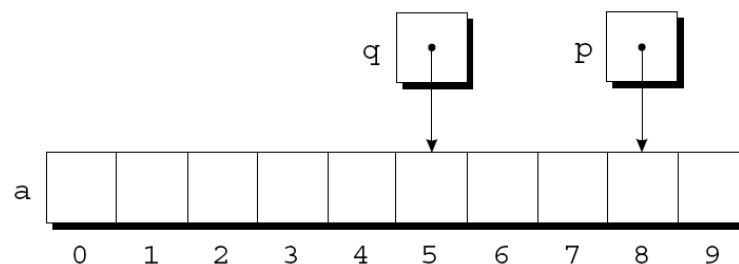
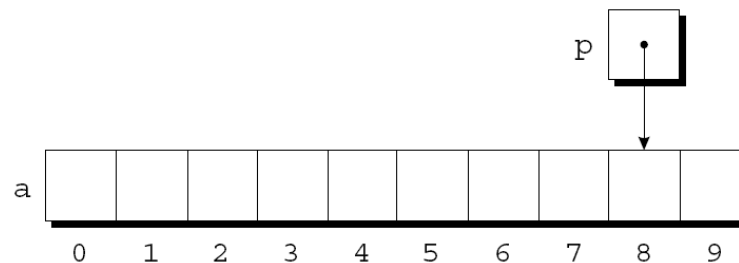
- 포인터 변수와 함께 사용되는 “-”의 뜻은 포인터를 이전 원소로 옮기라는 뜻임

```
int a[10], *p, *q;
```

```
p = &a[8];
```

```
q = p - 3;
```

```
p -= 6;
```



Using Pointers for Array Processing

- 포인터 연산 (반복적으로 포인터 변수의 값을 증가시킴으로써)을 통해서 배열의 원소들을 방문할 수 있음
- 배열 `a`의 원소들의 합을 계산하는 반복문:

```
#define N 10
...
int a[N]={11,34,82,7,64,98,47,18,29,20}, sum, *p;
...
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

Using an Array Name as a Pointer

- 배열의 이름은 배열의 첫번째 원소를 가리키는 포인터 값으로 사용가능함
- 배열의 이름을 포인터로 사용하는 예:

```
int a[10];  
  
*a = 7;          /* stores 7 in a[0] */  
*(a+1) = 12;     /* stores 12 in a[1] */
```

Q. $*(a+1)$ vs. $*a + 1$??

- $a + i$ 와 $\&a[i]$ 는 같은 표현
- $*(a+i)$ 와 $a[i]$ 도 같은 표현

Using an Array Name as a Pointer

■ 원래 구문:

```
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

■ 간소화된 구문

```
for (p = a; p < a + N; p++)  
    sum += *p;
```

혹은

```
p = a;  
while (p < a+N)  
    p++;
```

Warning: Pointer Arithmetic

- 포인터의 +/- 연산은 쉽게 유효한 메모리 영역 밖으로 포인터를 옮길 수 있기 때문에 주의
- 컴파일러가 유효한 메모리 영역을 체크해주지 않음

Exercise

- 배열의 최대, 최소값을 return해주는 함수를 작성하시오.
 - 포인터 연산을 사용하여 배열을 traverse할 것

```
void min_max(int number[], int size, <parameters>)  
{
```

<Source Code>

```
}
```


Q and A

