C Programming Language

(11th class)

Dohyung Kim

Assistant Professor @ Department of Computer Science

Today …

- **Revisit Pointer**

```c
#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;
}
```

```c
#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;
}
```

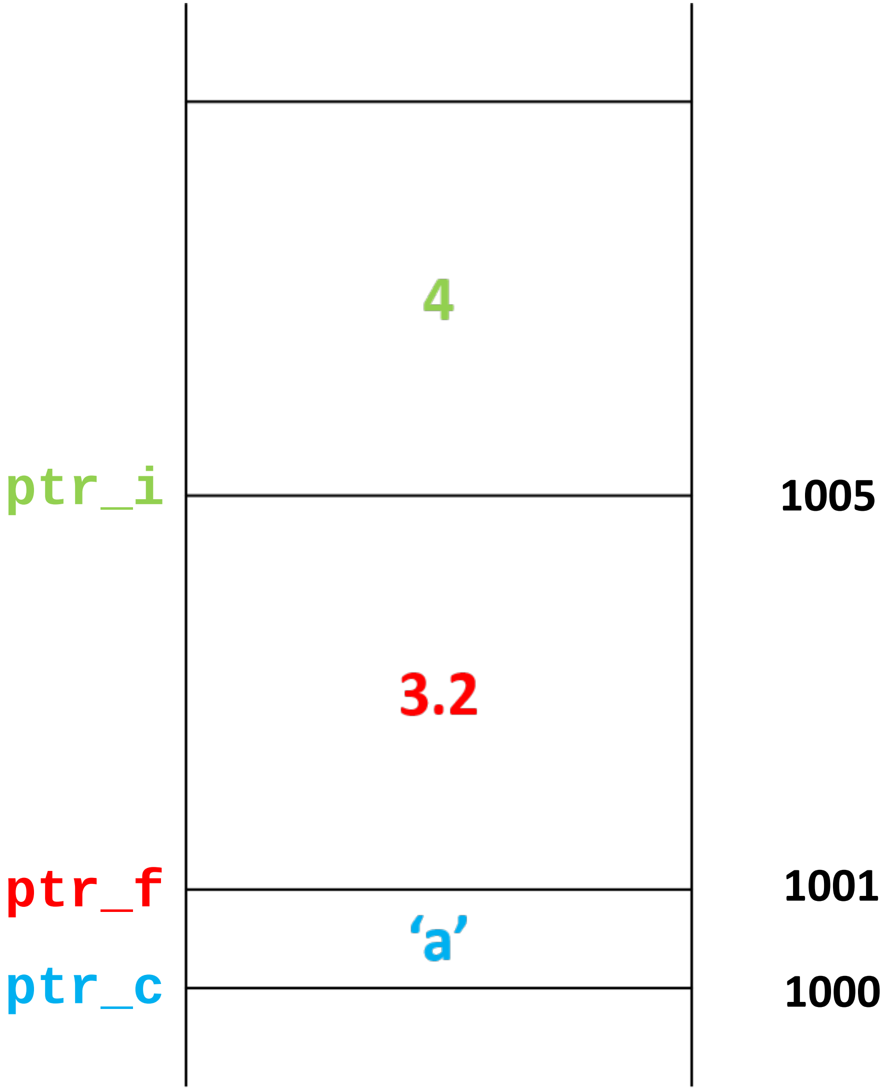
Variables and Addresses

- Each variable in a program occupies some bytes of memory.
- The address of the first byte is the address of the variable.

```
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;
```



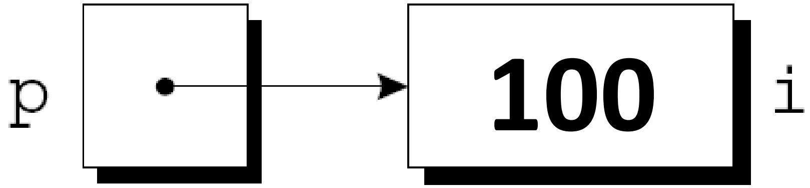
Pointer Variables

- **Pointer**
  - A data type whose value refers to the address of another value stored elsewhere in the memory.
  - A pointer variable can store the address of data (or a variable)

- **When we store the address of a variable `i` in the pointer variable p, we say that p "points to" `i`.**

- **A graphical representation:**

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



- **When a pointer variable is declared, its name must be preceded by an asterisk:**

```
int *p;
```

- **p is a pointer variable pointing to a variable of type `int`, but points nowhere in particular yet.**

- **It's crucial to initialize p before we use it.**
  - It is recommended to initialize pointer variables with NULL (0) to prevent a unaware access to a garbage address.

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

- **A conversion specification for the pointer type is %p.**

The Address Operator

- **One way to initialize a pointer variable is to assign it the address of a variable:**

  ```
  int i, *p;
  …
  p = &i;
  ```

- **& (ampersand) is an "address of" operator.**
  - or a reference operator

- **`"p = &i"` assigns the address of `i` to the variable `p`.**
  - Now, `p` points to `i`.

The Dereference Operator

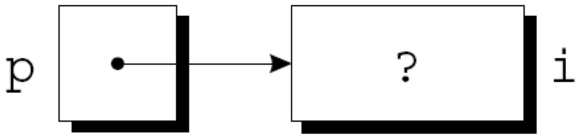
- **Once a pointer variable points to another variable, we can use the * (dereference) operator to access what's stored in the referenced variable.**

- **If p points to `i`, we can print the value of `i` as follows:**

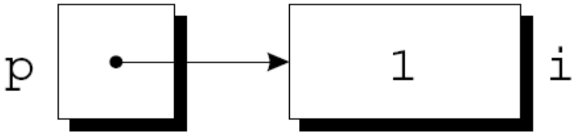
```
int i = 100;
int *p = &i;

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

The Dereference Operator Examples

```
p = &i;
```

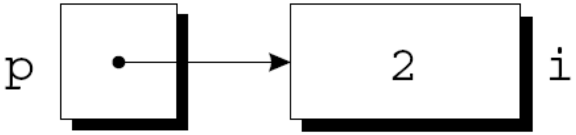


```
i = 1;
```



```
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

- Applying the deference operator to an uninitialized pointer variable causes undefined behavior:

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

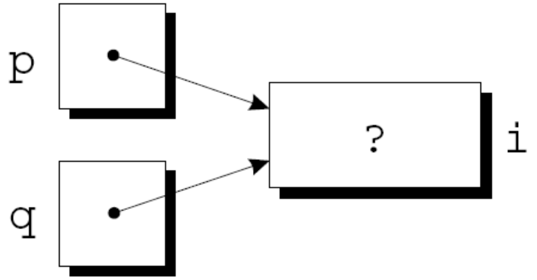
- Assigning a value to *p (uninitialized) can be dangerous:

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

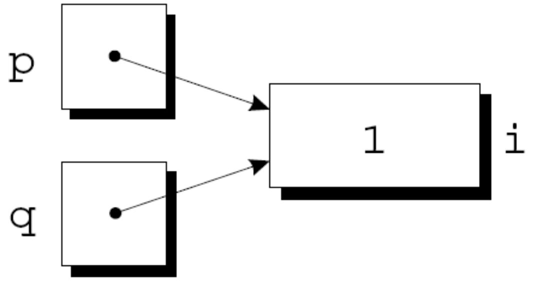
Pointer Assignment

- **Example of pointer assignment:**

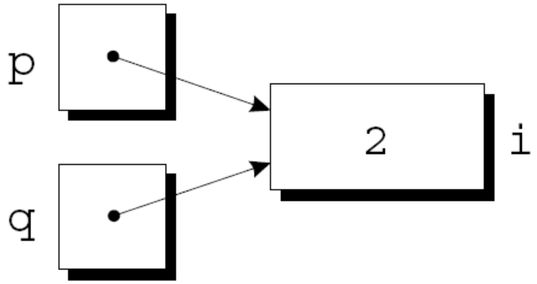
```
int i, *p, *q;
p = &i;
q = p;    /* q now points to the same place as p */
```



```
*p = 1;
```

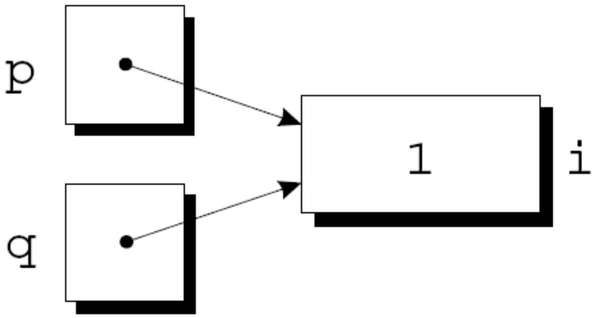


```
*q = 2;
```

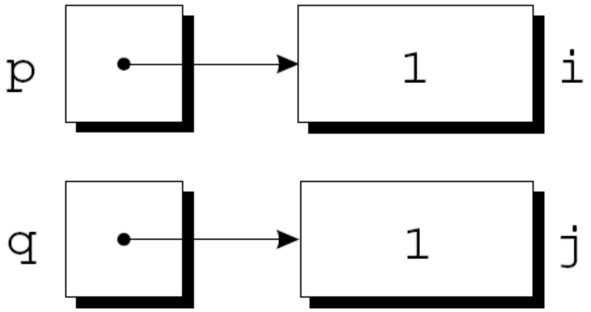


q = p; vs *q = *p;

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



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



Errors

- **What’s wrong with these codes?**

```
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

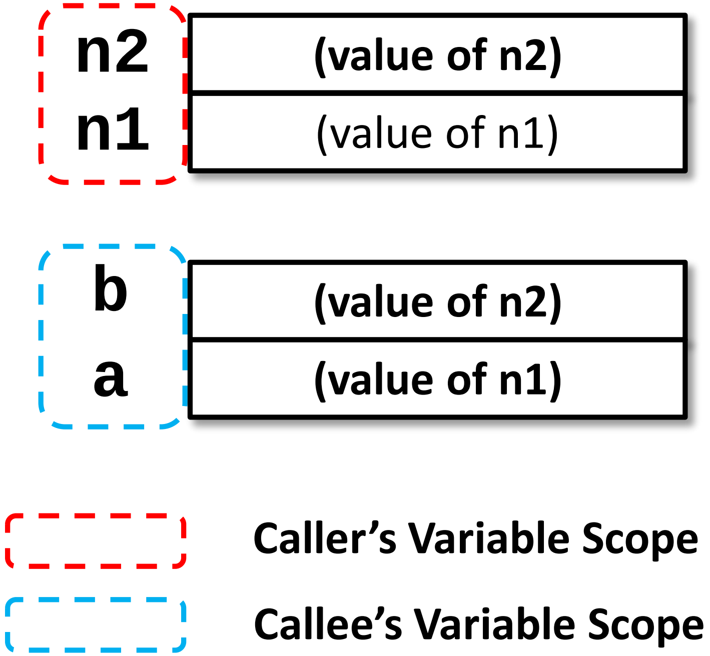
- **Call-by-value : passing the values of variables to a function as arguments**
  - Only values are copied to local function variables.
  - The call-by-value is the default rule in C programming.

- **Call-by-reference**
  - Functions may use pointer variables as arguments.
  - We can directly access the address of variables in the functions.

```
#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

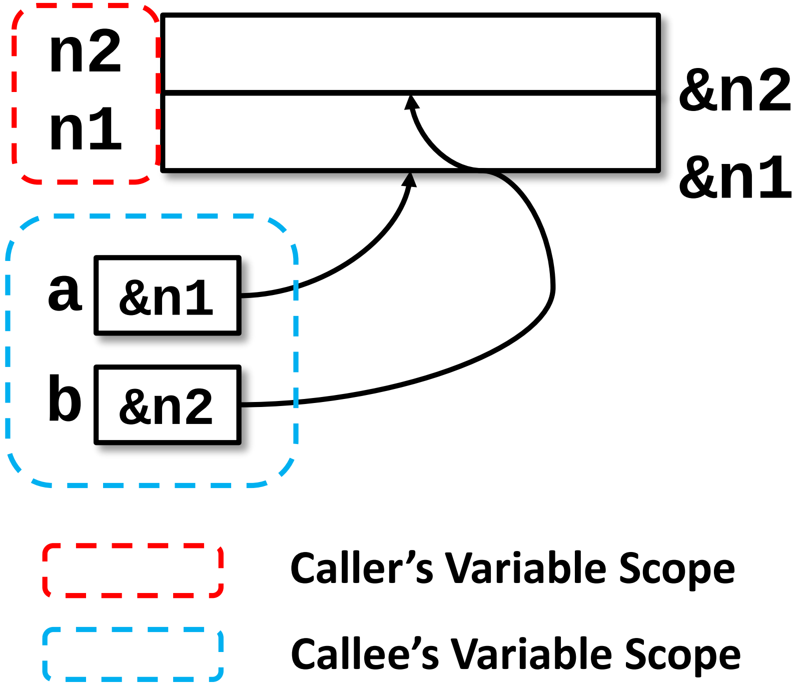
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}

int main(void) {
        int n1, n2;
        …
        swap(n1, n2);
        …
        return 0;
}
```



- **In swap function, values of a and b are swapped. But, the variables themselves are removed after returning. So, values of n1 and n2 cannot be swapped.**

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;
}
```



- The location of `n1` and `n2` are directly accessed in the `swap_ptr` function, and the values of `n1` and `n2` will be swapped.

Exercise : Pointer Variable Declaration

- **Print out the address of the variable num.**

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

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

Exercise - Address of Pointer Variables

- **The result of a function call `ex2()` is as the right-side box. Explain the result. (How can we calculate the third line?)**

```
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;
}
```

→ **Return type of the function is `int*`**

Pointers as Return Values

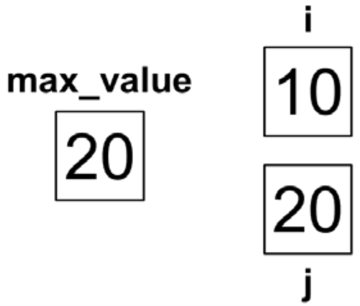
- A call of the `max` function:

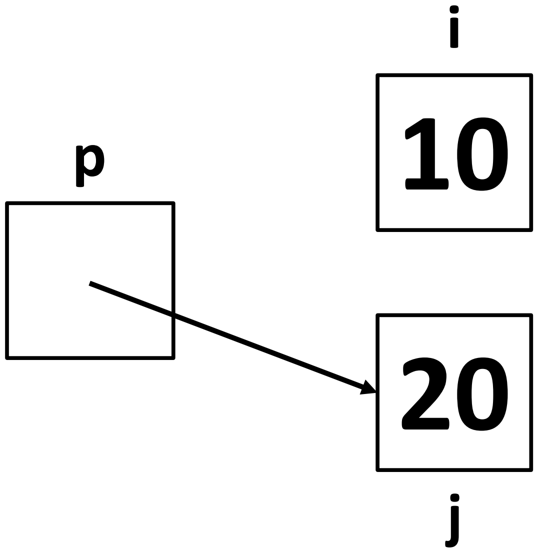
```
int *p_max, i, j;
…
p_max = max(&i, &j);
```

After the call, `p` points to either `i` or `j`.

call by value

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





Example 1

- **Using the `max` function above**

```
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

- **Pointers can be used for multiple return values.**

- **scanf**

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

Exercise

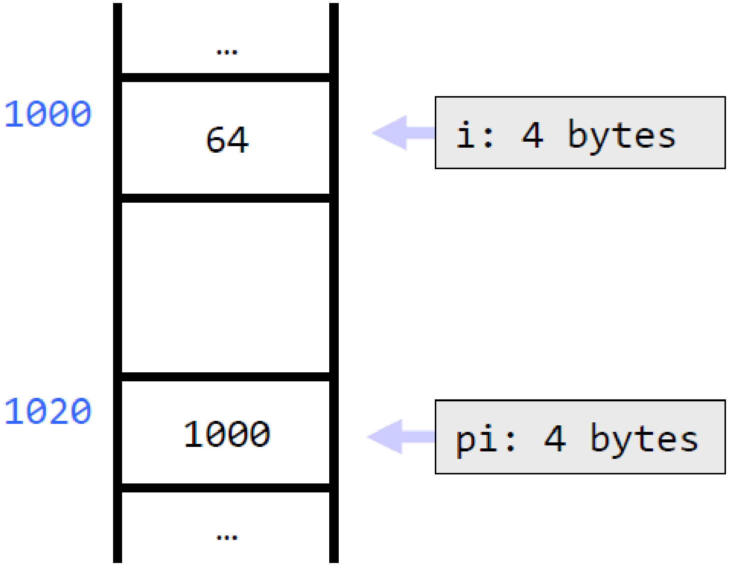
- **Make a function that returns the min and the max values of an array.**

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

      <Source Code>

}
```

Pointers to Pointers

- **A pointer variable is also a variable.**
  - The pointer has its own address.
  - We can have pointers to reference other pointers

```
#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

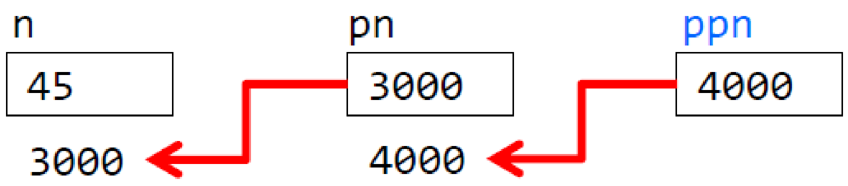
- **What are pointers to pointers ?**
  - Double (**) is used to denote the double pointer.
  - A usual pointer stores the address of the variable
  - A double pointer stores the address of pointer variables.

- **Declaration Example**

```
int **ptr2ptr;
```

Pointers to Pointers

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

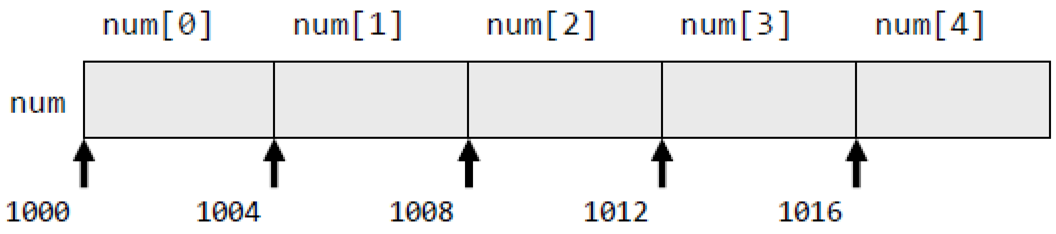


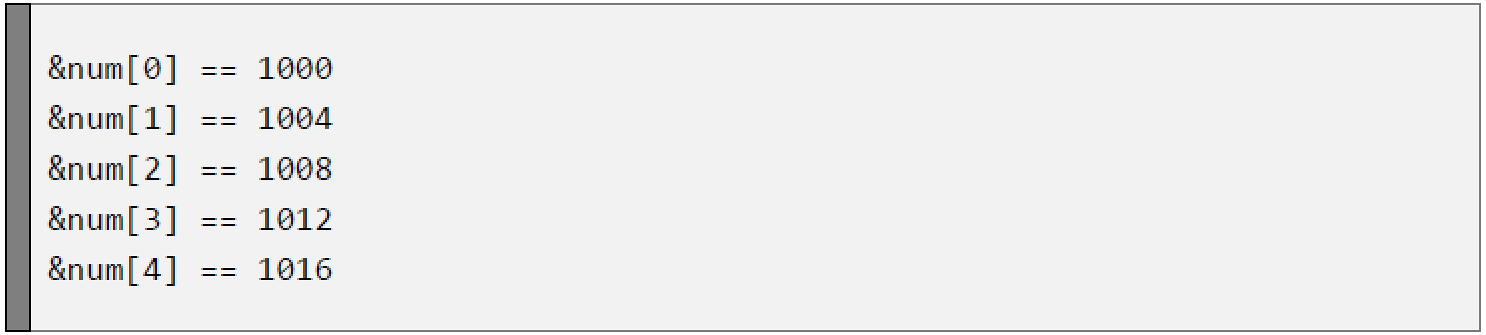
Statement	Output
*pn	45
**ppn	45
pn	3000 (=&n)
ppn	4000 (=&pn)

Array vs. Pointer

- **An address of each element in an array**

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





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

Array vs. Pointer

- **An array name is compatible with pointers**

```
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

- **Pointer arithmetic is one of the powerful features of C .**

- **This allows us to easily manipulate addresses directly.**

  - C allows us to perform arithmetic (addition and subtraction) on pointers to array elements.

  - If p points to an element of an array a, the other elements of a can be accessed by performing pointer arithmetic on p.

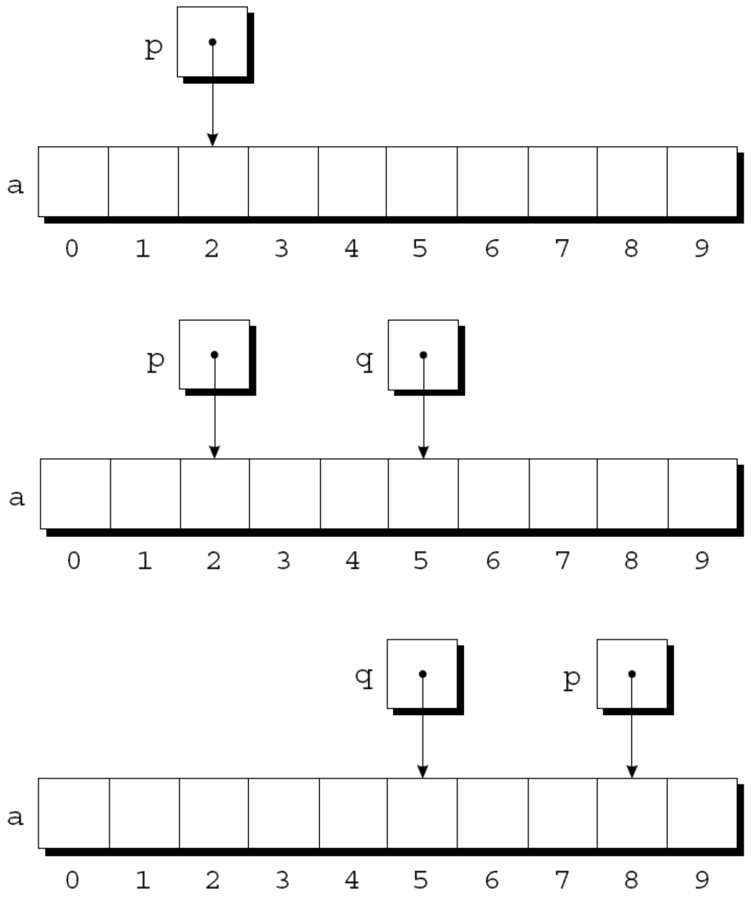
Run & Observe

```c
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

- **The meaning of “+” with a pointer is moving the pointer to a “next element”.**

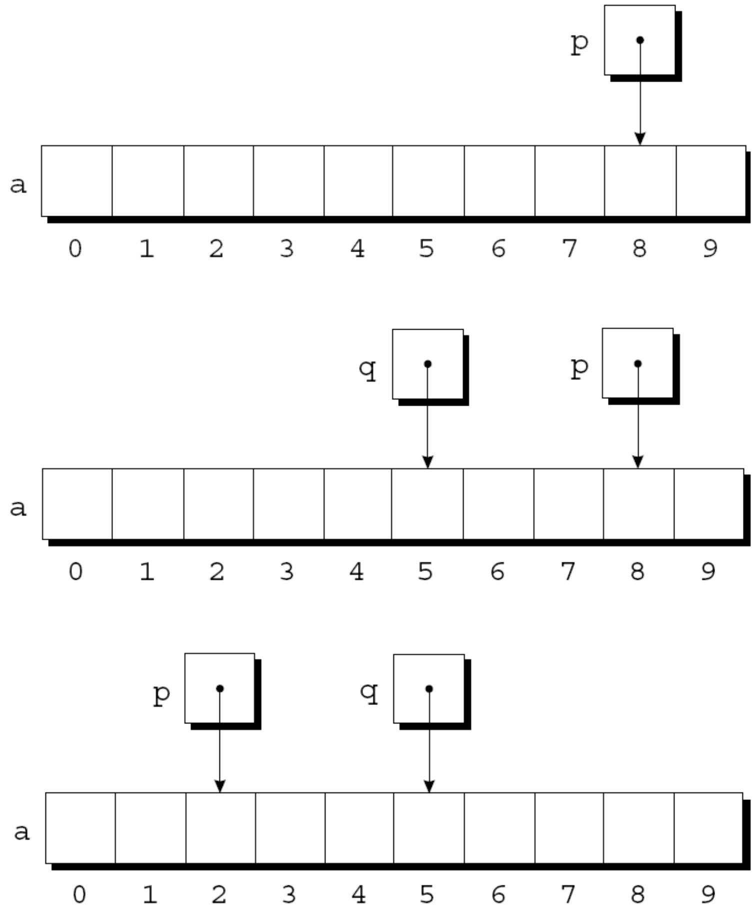
```
int a[10], *p, *q;

p = &a[2];

q = p + 3;

p += 6;
```



Decrement of Pointers

- **The meaning of “-” with a pointer is moving the pointer to a “previous element”**

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

p = &a[8];

q = p - 3;

p -= 6;
```



Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.

- A loop that sums the elements of an array a:

```
#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

- The name of an array can be used as a pointer to the first element in the array.

- Examples of using an array of a as a pointer:

```
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 is the same as &a[i].
- Also, *(a+i) is equivalent to a[i].

Using an Array Name as a Pointer

- **Original version:**

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

- **Simplified version :**

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

or

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

Warning: Pointer Arithmetic

- **The addition and subtraction of pointers would easily go outside the valid memory range.**

- **Compilers do not check the valid boundary.**

Exercise

- **Make a function that returns the min and max value of an array.**
  - You should use **pointer arithmetic** while traversing the array.

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

      <Source Code>

}
```

Q and A