Advanced C Programming & Lab

9. Pointers

Sejong University

Outline

1) Pointers?

- **Memory**
  - A place where information is stored to execute a program
  - Each 1 byte (8 bits) has a physical address

  - Conceptually, a series of spaces of size 1byte

  - Generally, size of an address is 4 bytes, represented as a hexadecimal number

Memory Address

0x003BDC97	0x003BDC98	0x003BDC99	0x003BDCA0	0x003BDCA1	0x003BDCA2	0x003BDCA3
0000 1101	0100 1010	0000 0001	0000 0000	0001 0010	1111 1110	1110 1101

Value stored in a memory

Example (byte)

1) Pointers?

- **Variables and Memory**
  - When declaring a variable, a memory location is assigned to the variable
  - &: the starting address of a variable

```
int a = 0;
printf("%d...%p", a, &a);  // %#X: hexadecimal

Result:
0...003BDC98
```

Address

0x003BDC97	0x003BDC98	0x003BDC99	0x003BDCA0	0x003BDCA1	0x003BDCA2	0x003BDCA3
0000 1101	0000 0000	0000 0000	0000 0000	0000 0000	1111 1110	1110 1101

Variable a – memory allocation (4bytes) : once allocated, it is fixed

1) Pointers?

- **Print the address using printf()**
  - Address is an integer
    - ✓ Print as a decimal (%d) or hexadecimal (%x) number
    - ✓ Compilation Warning
  - Conversion specification
    - ✓ %p : as a hexadecimal number

```
int a;

printf("%d\n", &a);   // Compilation Warning
printf("%#x\n", &a);  // Compilation Warning
printf("%X\n", &a);   // Compilation Warning
printf("%p\n", &a);
```

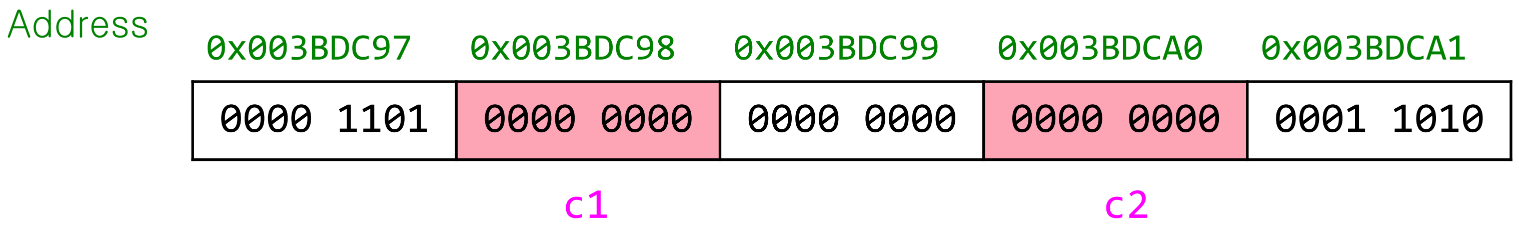
Result

```
3923096
0x3bcd98
3BDC98
003BDC98
```

1) Pointers?

- **Meaning of variables in C**
  1. Allocated space (Not address)
     - ✓ Variables on the left side (l-value) in an assignment or declaration statement
  2. Stored value
     - ✓ Variables on the right side (r-value) in an assignment statement or selection statement

```
char c1, c2;    // Allocate a space for c1, c2
c1 = c2;        // Store the value of c2 in the allocated space for c1
if( c1 < c2 )   // if the value of c1 is smaller than the value of c2
printf("%c",c1); // Pass the value of c1
```



1) Pointers? (Practice 1)

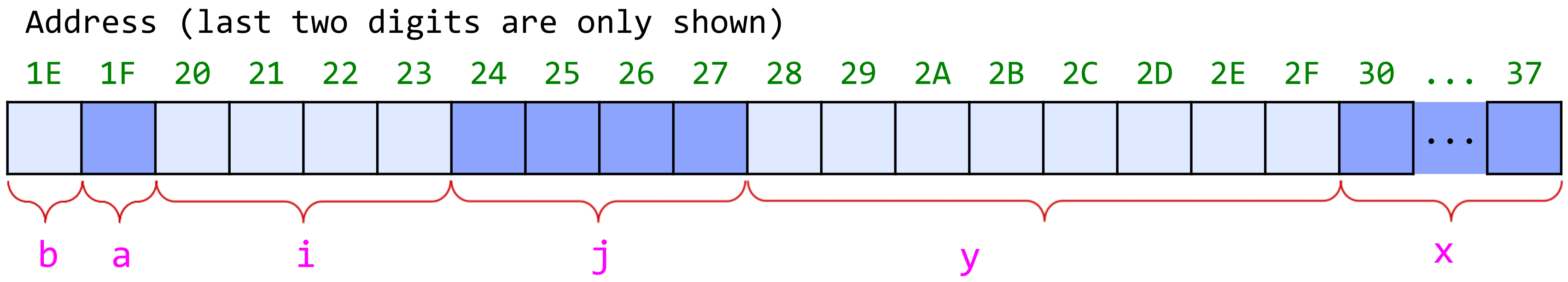
- **(Practice1) Print the address of a variable**

```
char a, b;
int i, j;
double x, y;
```

Result

```
a: 0018F91F, b: 0018F91E
i: 0018F924, j: 0018F920
x: 0018F930, y: 0018F928
```

- Memory


Address (last two digits are only shown)
1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 ... 37
...
b
a
i
j
y
x

1) Pointers? (Practice 2)

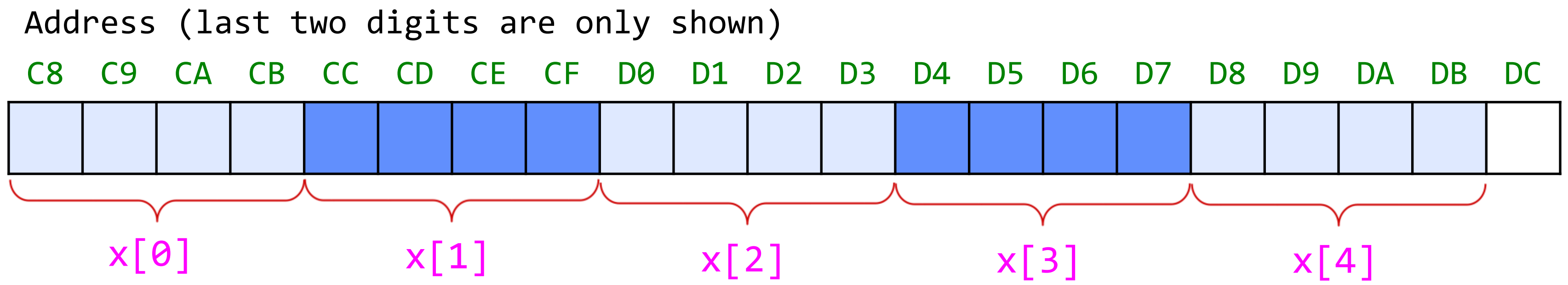
- **(Practice2) Print the address of the elements in an array.**

```
int x[5];
```

Result

```
x[0]: 001FFEC8
x[1]: 001FFECC
x[2]: 001FFED0
x[3]: 001FFED4
x[4]: 001FFED8
```

- Memory


Address (last two digits are only shown)
C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC
x[0]
x[1]
x[2]
x[3]
x[4]

1) Pointers?

- **Pointers: Data Type**
  - Refer to the **Address of a variable**
  - Point to a variable
  - Address is a number, but not always int type
  - Should declare as a pointer variable

1) Pointers?

- **Declaration**
  - data type + * (indicate a pointer) + variable name
  - int type pointer: point int type variable
  - float type pointer: point float type variable

    - ✓ Place *
    - ✓ ex)

```
char *pch;
int *pnum;
```

    - ✓ pch: **character type pointer** variable
    - ✓ pnum: **int type pointer** variable

→ pch and pnum store the address of different data types

1) Pointers?

- **Declaration**
  - Place (*) next to data type or variable name

```
char    *pch;
int     *pnum;
```

=

```
char*    pch;
int*     pnum;
```

✓ Generally, next to a variable name

  - Can declare pointer variables and (normal) variables

```
int *pnum1, num1=10, *pnum2, num2, arr[10];
```

✓ int type pointer variable: pnum1, pnum2
✓ int type: num1, num2, num1 is 10
✓ int type array: arr

1) Pointers?

- **Initialization**

```
int num, *pnum = &num;
```

- pnum: the address of a variable num

- (Warning!!) Compilation error

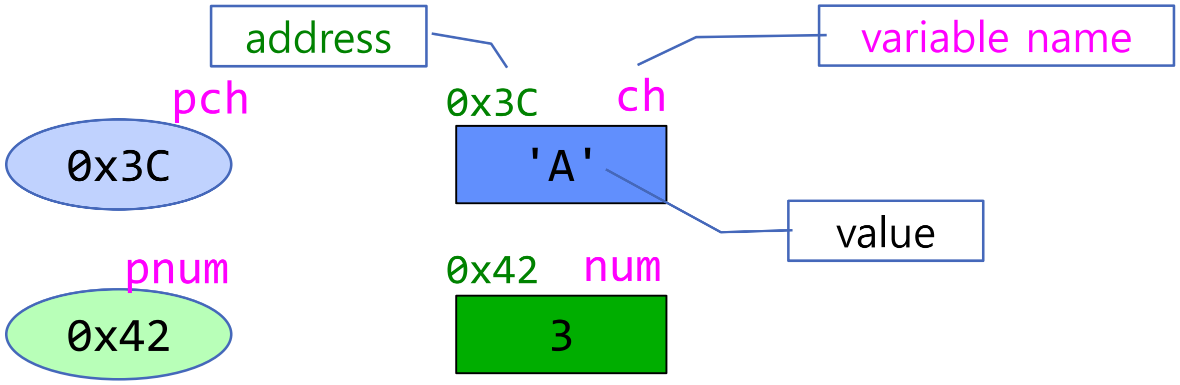
```
int *pnum = &num, num;        // Compilation error
```

  - ✓ Do not know the address of a variable num

1) Pointers?

- **Assignment**
  - **Address only**

```
char ch='A';
char *pch;
int num=3, *pnum;

pch = &ch;          // address of ch -> pch
pnum = &num;         // address of num -> pnum

printf("%c %p\n",ch, pch);    // print address: %p
printf("%d %p\n",num, pnum);
```

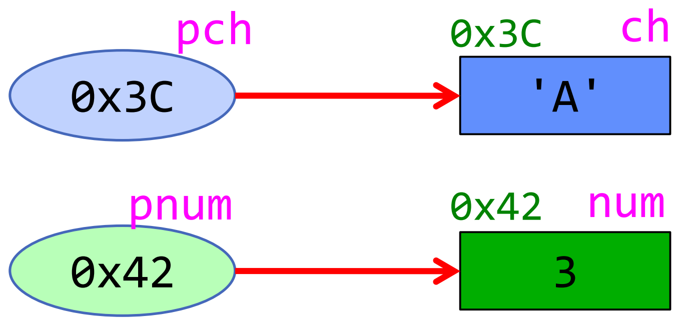


1) Pointers?

- **Assignment**
  - Point a variable: Assign the address of a variable to a pointer variable **(arrow →)**

```
pch = &ch;     // assign the address of ch to pch
pnum = &num;  // assign the address of num to pnum
```

  - **“a pointer variable pch points to a variable ch"**

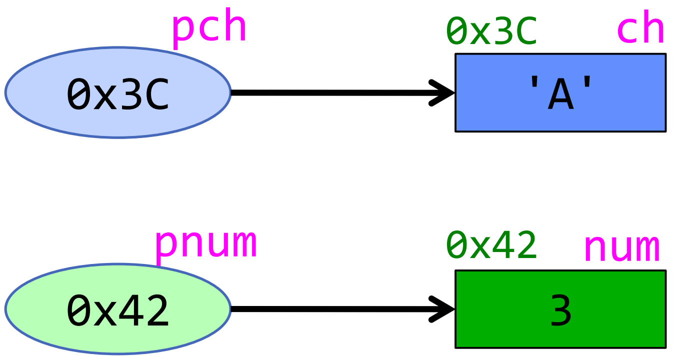


1) Pointers?

- **Pointer operator ***
  - Access to the variable pointed by a pointer variable

  ex) *pch : a variable that is pointed by a pointer variable pch
  a value that is stored in a memory location 0x3c

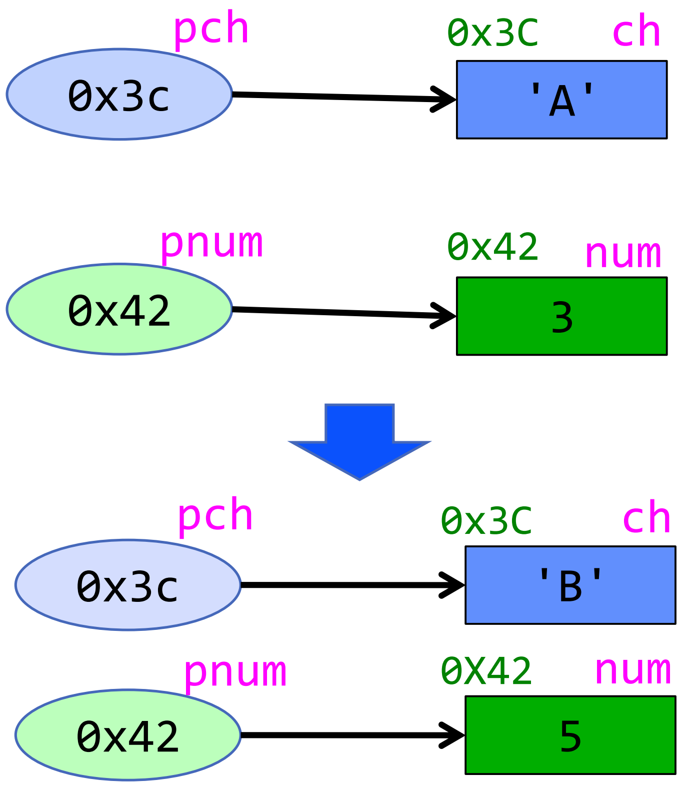
```
char ch='A', *pch;
int num=3, *pnum;

pch = &ch;
pnum = &num;

printf("%c %p\n", *pch, pch);
printf("%d %p\n", *pnum, pnum);

Output Display:
A 001EA03C
3 001EA042
```



1) Pointers?

- **Assignment**
  - Ex) ch='B ' is equivalent to *pch='B'
    former: **direct access,** latter: **indirect access**

```
char ch='A', *pch;
int num=3, *pnum;

pch = &ch;
pnum = &num;

*pch = 'B';
*pnum = 5;

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

Output Display:
B
5
```



1) Pointers?

- **Example**
  - *pnum points to an integer
  - Precedence of operators

```
int num=3, *pnum = &num;

*pnum = *pnum / 2 + 4;   // Integer operation:
                         //  assign 3/2+4=5 to num

if( *pnum == 5 )         // Integer comparison
   ++*pnum;              // Integer operation  ++(*pnum)
printf("%d", *pnum);     // As a function argument

Result:
6
```

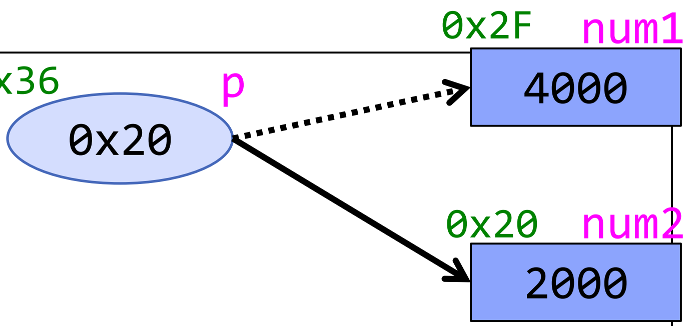
1) Pointers? (Practice 3)

- **(Practice 3) Draw a memory diagram:**

  - ✓ Declare int type num1, num2 and int type pointer p, initializing with the address of num1 (one statement)
  - ✓ Assign 3000 to the variable pointed by p
  - ✓ Assign the value that p points out to num2
  - ✓ p points to num2
  - ✓ Decrease the value that p points out by 1000
  - ✓ Assign the value that p points out, multiplied by 2, to num1
  - ✓ Pint out num1, num2, p
  - ✓ Print out the address of num1, num2, p

1) Pointers?

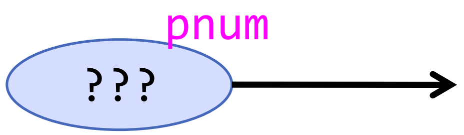
```
int num1, num2, *p = &num1;

*p = 3000;
num2 = *p;
p = &num2;
*p = *p - 1000;
num1 = *p * 2; // second *: multiplication

printf("Value: num1=%d num2=%d p=%p\n", num1, num2, p);
printf("Address: num1=%p num2=%p p=%p\n", &num1, &num2, &p);
```



Do not know what a pointer is?
Draw a diagram !!

1) Pointers?

- **Caution 1 (Initialization)**
  - No initialization?



```
int *pnum ;    // pnum: garbage value
*pnum = 9 ;   // Runtime error
```

➔

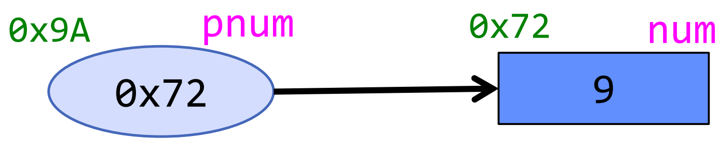
```
int *pnum, num;
pnum = &num; // assign the address of a varible
*pnum = 9;
```

- **NULL Pointers**
  - ✓ Do not point any variables
  - ✓ In reality, it is 0. Selection statement treats it as false
  - ✓ pnum = **NULL**;
  - ✓ Likewise, a (normal) variable is initialized to 0

1) Pointers?

- **Caution 2**
  - **&** : Any variables (including pointer variables)
    - ✓ A pointer variable is a variable (allocated in a memory)
  - ***** : Only pointer variables

```
int num=9, *pnum = &num;

printf("%p %p %p\n", &pnum, pnum, *pnum);

printf("%p %p %p\n", &num, num, *num); // Compilation error
```



1) Pointers?

- **Caution 3 (Assignment)**
  - Variable data type and pointer data type should be the same
    - ✓ Assign char type address to int type pointer?

```
char ch='A', *pch;
int num=3, *pnum;

pch = &num;    // compilation warning or error
pnum = &ch;    // compilation warning or error

*pch = 66;     // No error, but not a good way
*pnum = 'a';   // May cause an error

printf("%c\n", *pch);
printf("%d\n", *pnum);
```

    - ✓ Draw a memory diagram

1) Pointers?

- **Caution 3 (Assignment)**
  - Ex

```
char ch='A', *pch = &ch;
int *pnum;

pnum = pch;    // Compilation warning or error

*pnum = 1024;  // May cause runtime error

printf("%c\n", *pch);
printf("%d\n", *pnum);
```

  - Differing data types: no syntax error but may cause an error during execution

1) Pointers?

- **Size of a pointer**
  - Same as the size of the address (depending on the system)
  - sizeof()

```
char *pch;
int *pnum;
double *pdnum;

printf("%d,", sizeof(pch));
printf("%d,", sizeof(pnum));
printf("%d\n", sizeof(pdnum));

Result:
4,4,4
```

  - **Regardless of pointer data type,** the same amount of memory space is needed

Outline

2) Arrays and Pointers

- **Name of an array**
  - Name of a (normal) variable
    - ✓ Value stored in a variable
    - ✓ &: address

```
int a = 9;

printf("%d %p", a, &a); // value, address
```

  - Name of an array: **starting address of an array**
    - ✓ b and &b refer to the same address

```
int b[10] = {0};

printf("%p %p", b, &b); // address of an array b

Output Display:
001ce2f0 001ce2f0
```

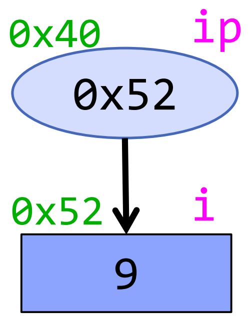
2) Arrays and Pointers

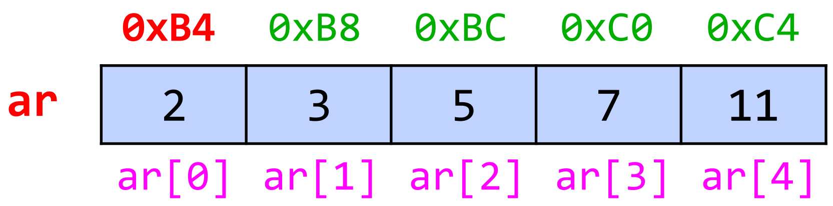
- **(Normal) variable and array**

(Normal) Variable	Array

`int i=9, *ip = &i;`
 	
`i  : value`
`ip : address`
	`int ar[5]={2, 3, 5, 7, 11};`
	
	`ar[2] : value in ar[2]`
	`ar : starting address of ar`

`&i  : address of i`
`&ip : address of ip`
	`&ar[2] : address of ar[2]`
	`&ar : starting address of ar`



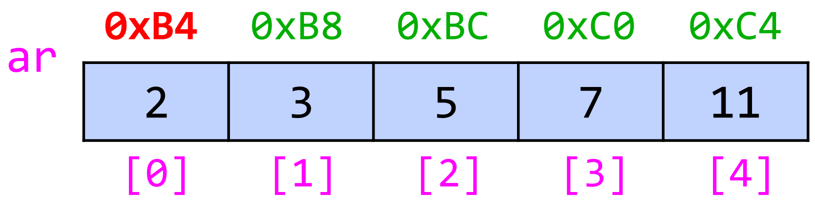


※ element of an array:
equivalent to a (normal) variable

2) Arrays and Pointers

- **Access by address**
  - Name of an array is an address, can use *
    - ✓ ar : starting address of ar
    - ✓ *ar : value stored in the starting address -> first element

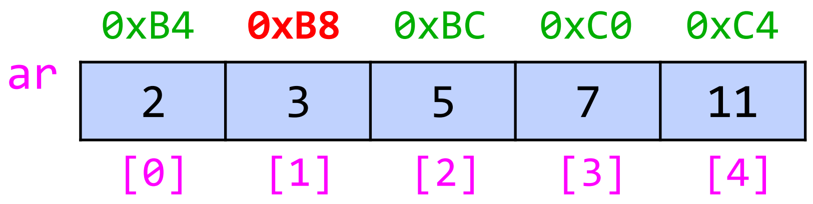
```
int ar[5]={2, 3, 5, 7, 11};

printf("%p %d %d\n", ar, ar[0], *ar);

Output Display:
001E40B4 2 2
```



2) Arrays and Pointers

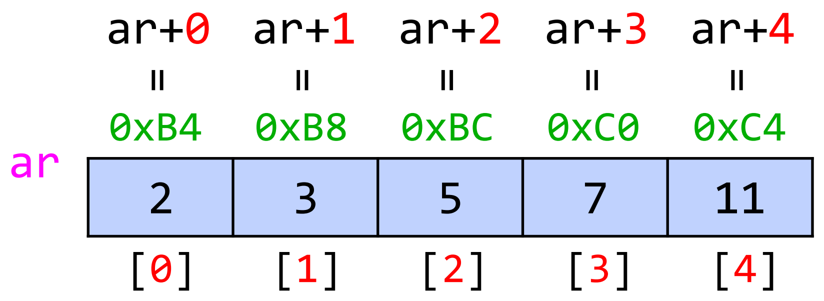
- **Increment/Decrement (ar: 0xB4)**
  - ar+1 ? 0xB5 ? 0xB8 ?
  - *(ar+1)?

```
int ar[5]={2, 3, 5, 7, 11};

printf("%p %d %d\n", ar+1, ar[1], *(ar+1));

Output Display:
001E40B8 3 3
```



2) Arrays and Pointers

- **Increment/Decrement: Depending on the size of a variable**
  - int type array: 4
  - **ar+i** : the address of **i** th element of an array ar
  - ***(ar+i)** : the value of **i** th element of an array ar, i.e., ar[i]



2) Arrays and Pointers

- **(Practice 4) Declare and print arrays and their elements**

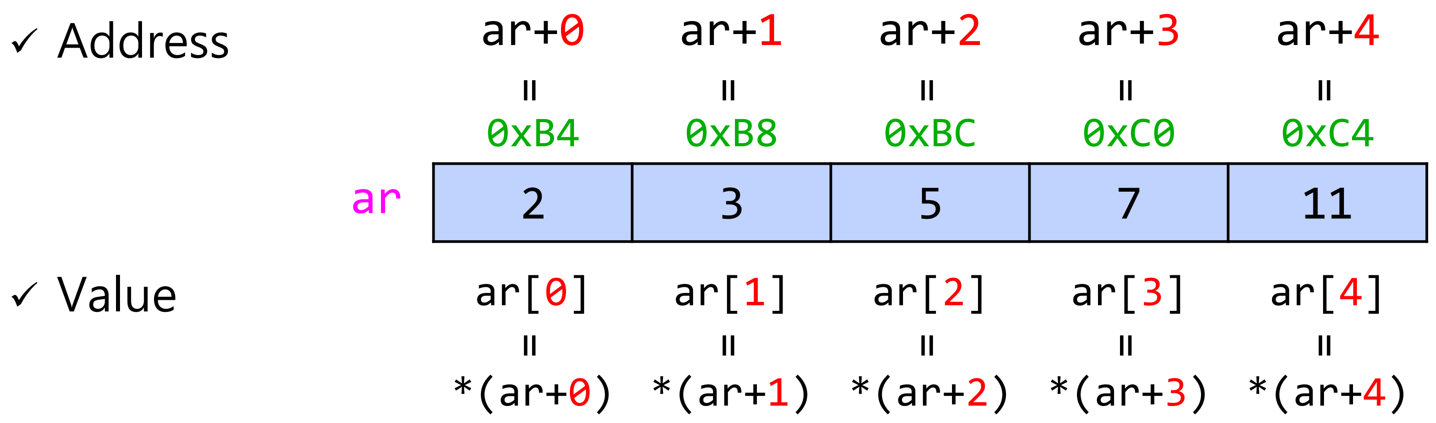
```
char car[5]={'H','e','l','l','o'};
double dar[5]={1.1, 2.2, 3.3, 4.4, 5.5};
```

- ✓ `car, car[0], *car`
- ✓ `car+1, car[1], *(car+1)`
- ✓ `car+2, car[2], *(car+2)`

- ✓ `dar, dar[0], *dar`
- ✓ `dar+1, dar[1], *(dar+1)`
- ✓ `dar+2, dar[2], *(dar+2)`

2) Arrays and Pointers

- **Pointers can be used to refer an array**



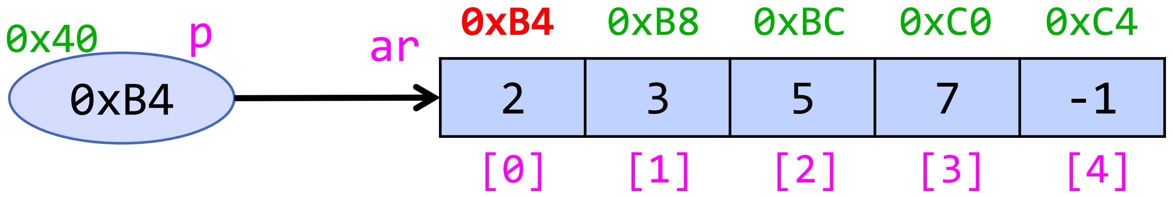
- **Increment/decrement of an address: depending on the size of a variable**

`*(ar+i) = ar[i]`

2) Arrays and Pointers

- **Assign an array to a pointer variable**
  - Name of an array is an address: Pointer

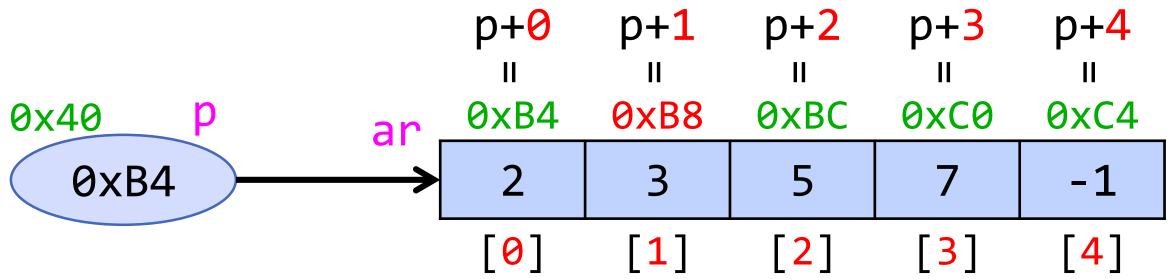
```
int ar[5]={2, 3, 5, 7, -1};
int *p = ar;

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

Output Display:
001E40B4 2
```



2) Arrays and Pointers

- **Increment/decrement: pointer variables**
  - Depending on the size of a variable

```
int ar[5]={2, 3, 5, 7, -1};
int *p = ar;

printf("%p %d\n", p+1, *(p+1));

Output Display:
001E40B8 3
```



2) Arrays and Pointers

- **Use a pointer variable as an array**
  - Index of an array

```
int ar[5]={2, 3, 5, 7, 11};
int *p = ar;

printf("%p %d %d\n", p, p[0], *p);
printf("%p %d %d\n", p+1, p[1], *(p+1));

Output Display:
001E40B4 2 2
001E40B8 3 3
```

2) Arrays and Pointers

- **(Practice 5) Declare and print pointer variables**

```
char car[5]={'H','e','l','l','o'}, *cp=car;
double dar[5]={1.1, 2.2, 3.3, 4.4, 5.5}, *dp=dar;
```

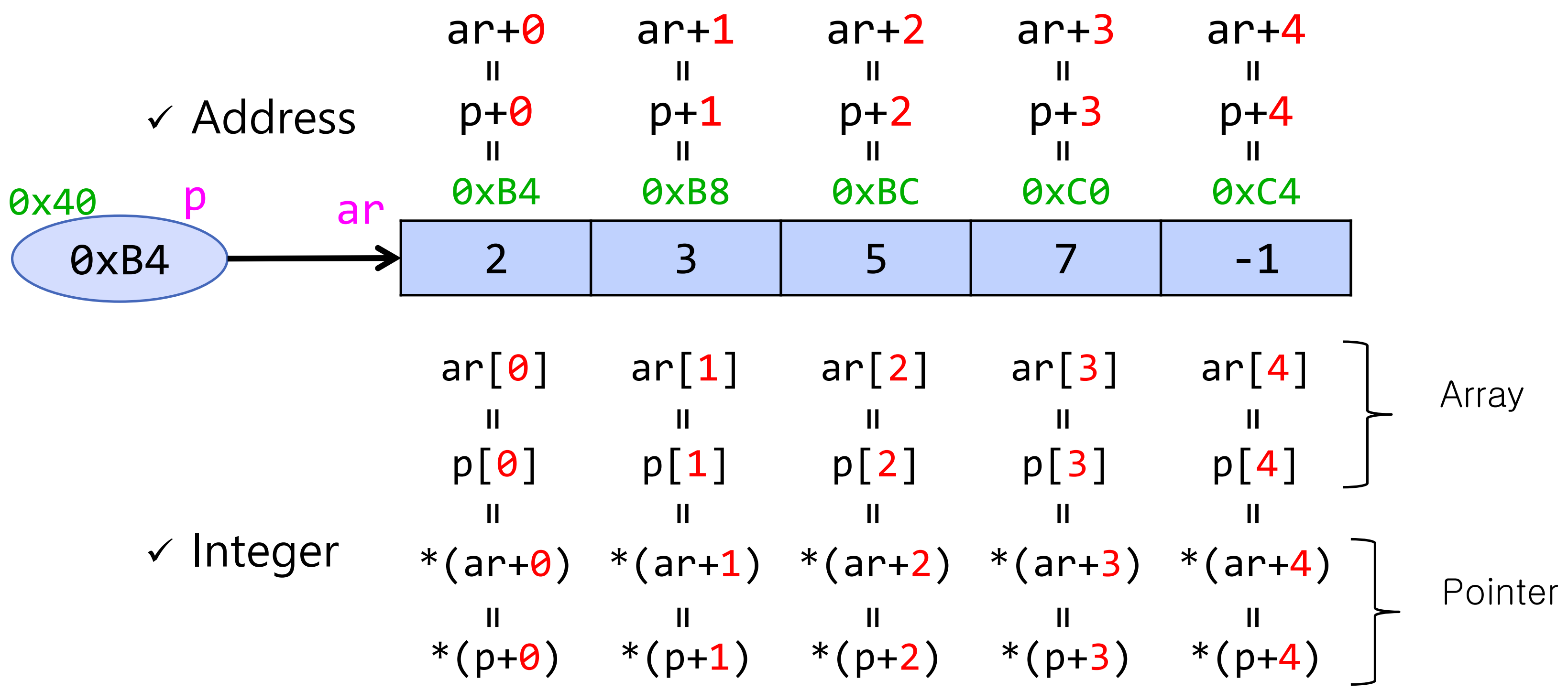
✓ `cp, cp[0], *cp`
✓ `cp+1, cp[1], *(cp+1)`
✓ `cp+2, cp[2], *(cp+2)`

✓ `dp, dp[0], *dp`
✓ `dp+1, dp[1], *(dp+1)`
✓ `dp+2, dp[2], *(dp+2)`

2) Arrays and Pointers

- **Arrays and Pointers**
  - Both denote address

```
int ar[5], *p = ar;
```



✓ Address
ar+0 = p+0 = 0xB4
ar+1 = p+1 = 0xB8
ar+2 = p+2 = 0xBC
ar+3 = p+3 = 0xC0
ar+4 = p+4 = 0xC4
0x40
p
0xB4
ar
2
3
5
7
-1
✓ Integer
ar[0] = p[0] = *(ar+0) = *(p+0)
ar[1] = p[1] = *(ar+1) = *(p+1)
ar[2] = p[2] = *(ar+2) = *(p+2)
ar[3] = p[3] = *(ar+3) = *(p+3)
ar[4] = p[4] = *(ar+4) = *(p+4)
Array
Pointer

2) Arrays and Pointers

- **Arrays and Pointers**

```
int ar[5], *p = ar;
```

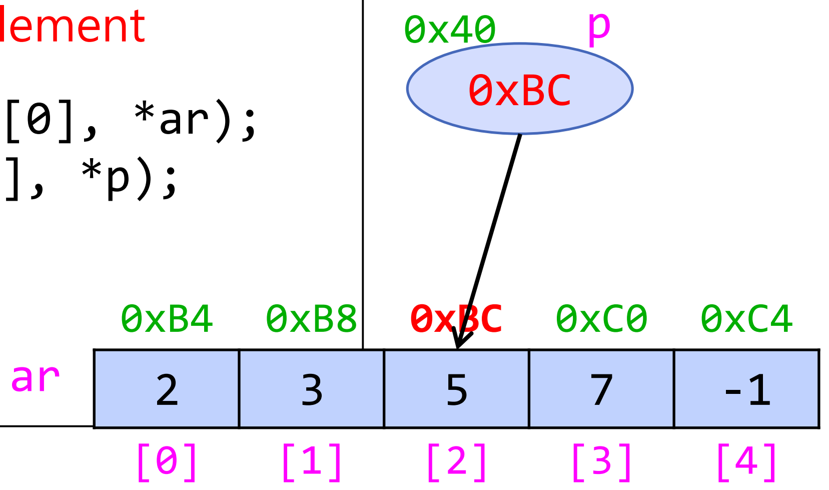
- Address + 1, increase by the size of a varaible
  - ✓ ar + 3 , p + 3 : ar, p: address
- Access the value
  - ✓ arr[3] , p[3] : index
  - ✓ *(arr+3) , *(p+3) : pointer

Both refer to an address

2) Arrays and Pointers

- **Caution 1**
  - Can point to any element of an array

```
int ar[5]={2, 3, 5, 7, -1};
int *p = &ar[2];      // 2nd element

printf("%p %d %d\n", ar, ar[0], *ar);
printf("%p %d %d\n", p, p[0], *p);

Result:
001E40B4 2 2
001E40BC 5 5
```



2) Arrays and Pointers

- **Caution 2**
  - Parenthesis
    - ✓ *(ar+2) == ar[2] == 5
    - ✓ *ar + 2 == *(ar) + 2 == ar[0]+2 == 4 (Operator precedence)

```
int ar[5]={2, 3, 5, 7, 11};
int *p = ar;

printf("%d %d\n", *(ar+2), *ar+2);
printf("%d %d\n", *(p+4), *p+4);

Output Display:
5 4
11 6
```

2) Arrays and Pointers

- **Caution 3**
  - Amount of increment/decrement is determined by the **data type** of a pointer

  - For example, Assign char * pointer to int array
    - ✓ Increase/decrease by 1 (size of char type)

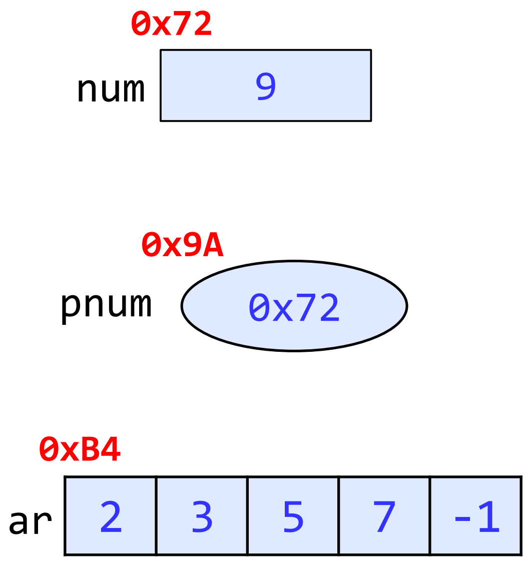
```
int ar[5]={2, 3, 5, 7, -1}, i;
char *p = (char *) ar;

for( i=0; i < 5 ; ++i )
  printf("%p, %d\n", p+i, *(p+i));
```

```
001E40B4, 2
001E40B5, 0
001E40B6, 0
001E40B7, 0
001E40B8, 3
```

    - ✓ Remove the second line (char *), could cause compilation error (depending on compiler)

2) Arrays and Pointers

- **Array vs Pointer**
  - int num;
    - ✓ **Value** stored in num: changeable
    - ✓ **Address** assigned to num: unchangeable
  - int *p;
    - ✓ **Value(address)** stored in p: changeable
    - ✓ Address assigned to p: unchangeable
  - int ar[5];
    - ✓ **Value** stroed in ar: changeable
    - ✓ **Address** assigned to ar: unchangeable
      - ✓ Name of an array: **constant pointer** - unchangeable
      - ✓ Different when it appears on the left side (l-value) and right side (r-value)



2) Arrays and Pointers

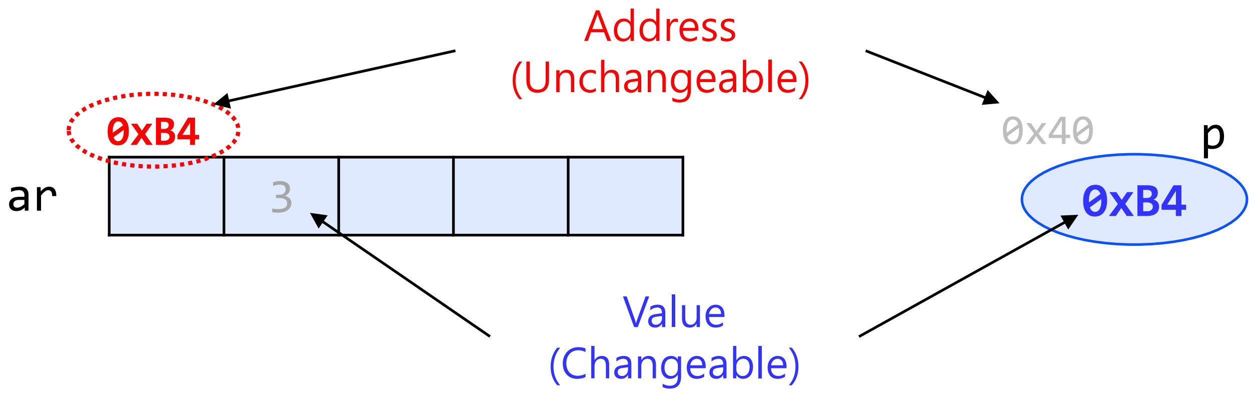
- **Difference between the name of an array and a pointer variable**

```
int num, *p, ar[5];

num = 1;        // OK
++num;          // OK
&num = ar       // NO (Compilation Error)

p = &num;        // OK
++p;            // OK
&p = ar;       // NO (Compilation Error)

ar = &num;      // NO (Compilation Error)
++ar;           // NO
&ar = &num;     // NO (Compilation Error)
               // ar and &ar are the same
```

2) Arrays and Pointers

- **Name of an array and a pointer variable**

	Name of an array (ar)	Pointer variable (p)
r-value	Starting address of an array	Value stored in a pointer variable
l-value	Cannot change the starting address of an array	Can change the value stored in a pointer variable
	`ar = ar + 1;` (X)	`p = p + 1;` (O)



Outline

3) Pointer Operation

- **Addition, Subtraction**
  - ++, --, +=, -=

```
int ar[5]={2, 3, 5, 7, -1}, i=4;
int *p = &ar[1];

printf("%p %d\n", p, *p);          // &ar[1], ar[1]
printf("%p %d\n", --p, *p);        // ar[0]
printf("%p %d\n", p+i, *(p+i));   // ar[4](value of p)+16

Result:
0018F9A8 3
0018F9A4 2
0018F9B4 -1
```

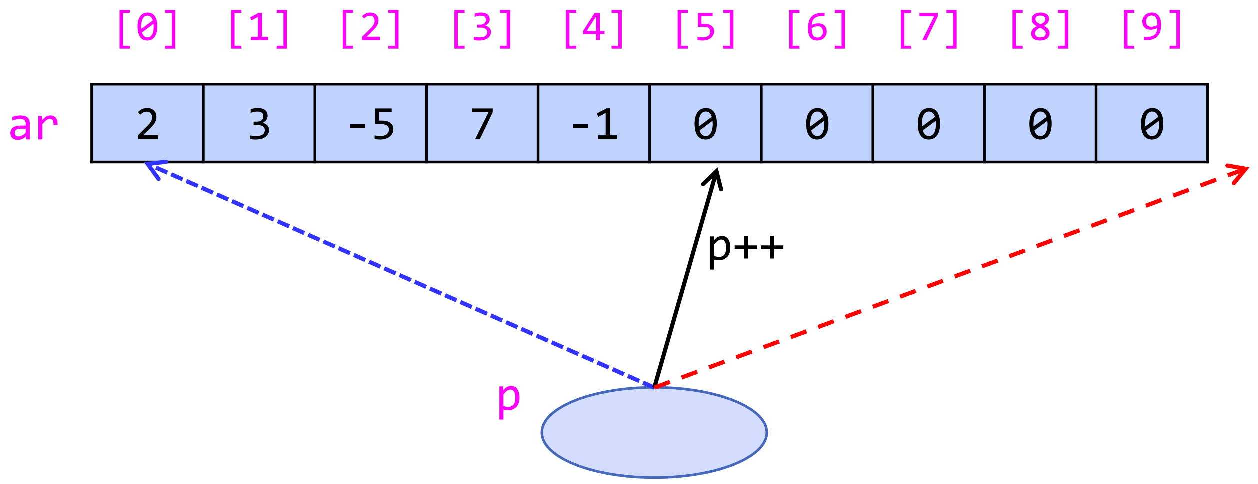
3) Pointer Operation

- **(Practice 6) Addition**
  - Declare int pointer p1 and character pointer p2, Initialize them to NULL
  - Print out p1 and p2
  - Increase p1 and p2 by 1
  - Print out p1 and p2
  - Increase p1 and p2 by 2
  - Print out p1 and p2

  - [Check the results]
    - ✓ Check p1 and p2
    - ✓ Note: NULL denotes an address 0

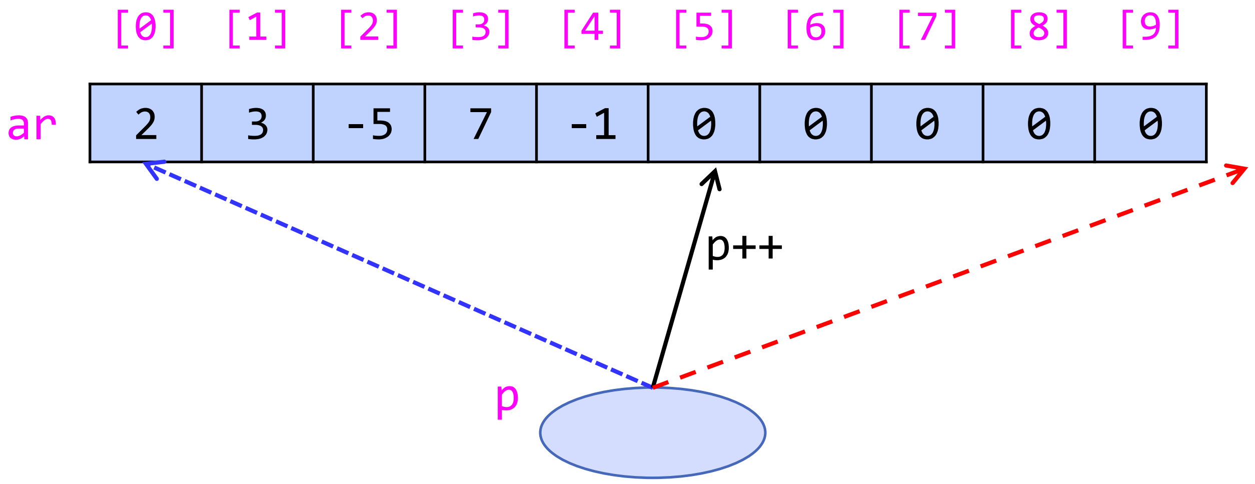
3) Pointer Operation

- **(Practice 7) Pointer operation to traverse an array**
  - Declare int array ar[10], initialize it to {2, 3, 5, 7, -1}
  - Declare int variable I and int pointer p
  - p points to ar
  - for loop: increase I, repeat the following 10 times
    - ✓ Print out the value pointed by p, increase p by 1



3) Pointer Operation

```
int ar[10]={2, 3, 5, 7, -1};
int i, *p;

p=ar;
for(i=0;i<10;i++)
   printf("%d ", *(p++));
```



3) Pointer Operation

- **Address comparison**
  - ==, != , < , > , >= , <=

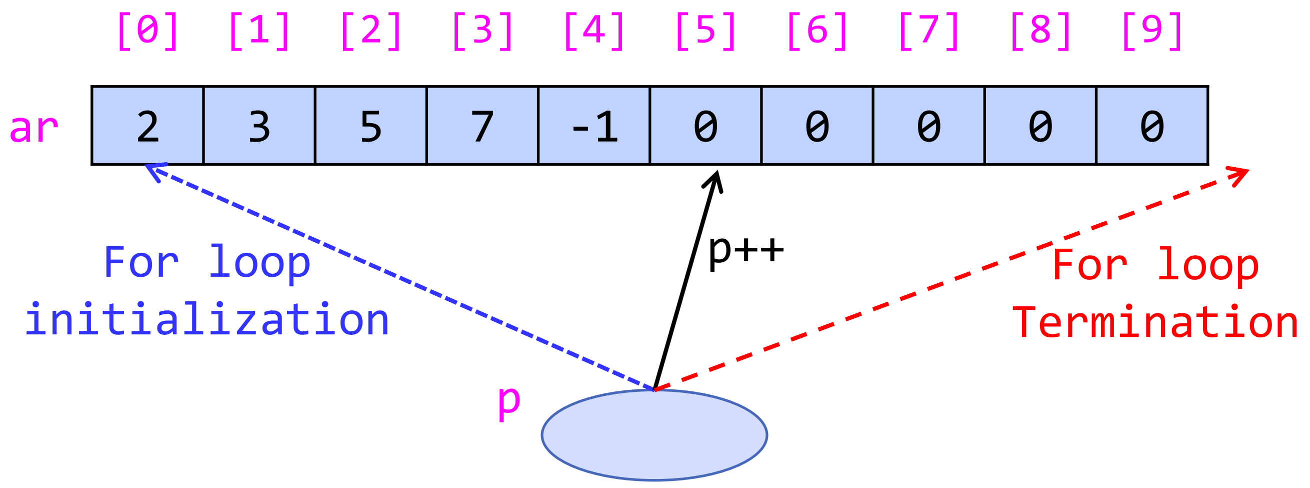
```
int ar[5]={2, 3, 5, 7, -1}, *p1, *p2;

p1 = &ar[1]; p2 = &ar[4];

printf("%p %p\n", p1, p2);
printf("%d %d\n", p1 < p2, *p1 < *p2);

Result:
0018F9DC 0018F9E8
1 0
```

✓ Pointer operation can be used for ordinary variables, more useful for arrays

3) Pointer Operation

- **(Example) Pointer operation to traverse an array (ver. 2)**
  - Address comparison (Practice 7)

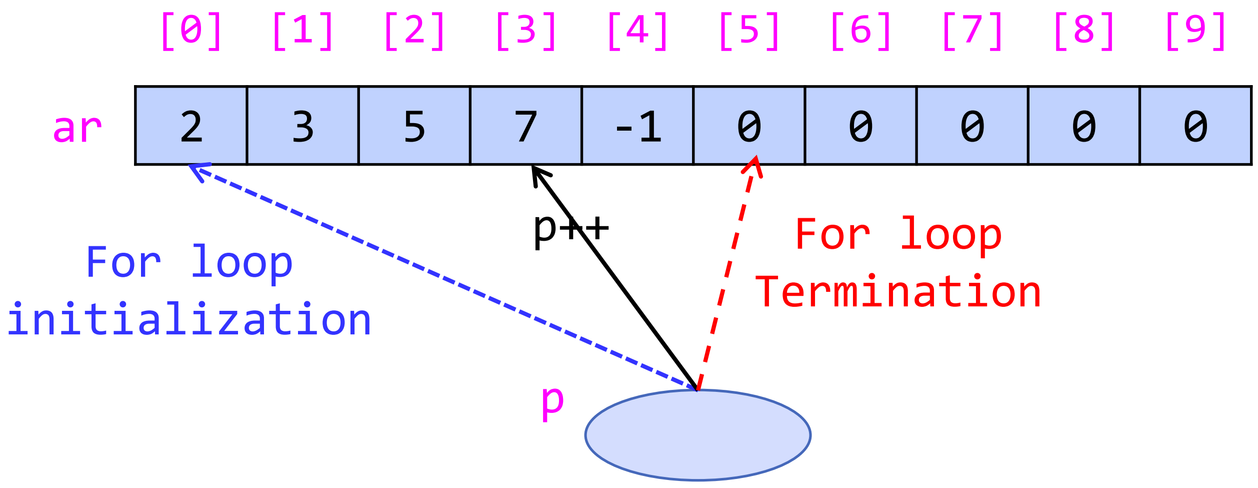
```
int ar[10]={2, 3, 5, 7, -1};
int *p;

for( p = ar ; p < &ar[10] ; p++ )
    printf("%d ", *p);
```



3) Pointer Operation

- **(Example) Pointer operation to traverse an array**
  - Repeat until the element of an array is 0

```
int ar[10]={2, 3, 5, 7, -1};
int *p;

for( p = ar ; *p ; p++ )
    printf("%d ", *p);
```



3) Pointer Operation

- **Caution**
  - **Addition and Subtration** are allowed
    - ✓ Multiplication and division are not allowed
  - Only integers can be used
    - ✓ Double type and address are not allowed

```
int num1, num2;

printf("%p", &num1 * 2);      // Compilation Error
printf("%p", &num1 + &num2); // Compilation Error
```

Outline

1) Pointers?
2) Arrays and Pointers
3) Pointer Operation
4) **Pointer Arguments**
5) Arrays of Pointers

4) Pointer Arguments

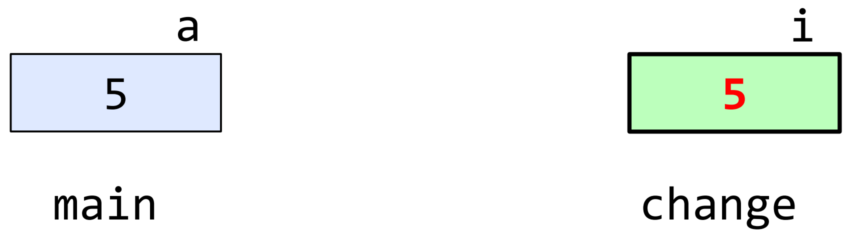
- **Function – Integer argument**
  - Function Call: Allocate a space to a formal parameter(variable), Assign an integer value (actual parameter) that is passed to the function

Pass the value stored in a variable a

5

```
void main()
{   int a = 5;

    change(a);

    printf("a=%d\n", a);
}
```

```
void change(int i)
{
    i = 10;
}
```



Memory diagram: staring change()

4) Pointer Arguments

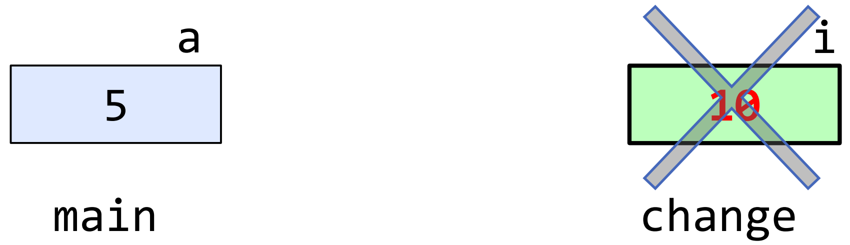
- Function body: Assign 10 to a local variable i
- Function termination: Eliminate the local variable (including arguments) (return allocated memory space)

```
void main()
{  int a = 5;

   change(a);

   printf("a=%d\n", a);
}
```

```
void change(int i)
{
   i = 10;
}
```

Result:

```
a=5
```



Eliminate variable

Memory diagram: terminating change

4) Pointer Arguments

- **Address as a function argument?**
  - Declare a **pointer** as an argument
  - Indirect reference by a pointer variable

```
void main()
{  int a = 5;

   change(&a);

   printf("a=%d\n", a);
}
```

```
void change(int *p)
{
   *p = 10;
}
```

4) Pointer Arguments

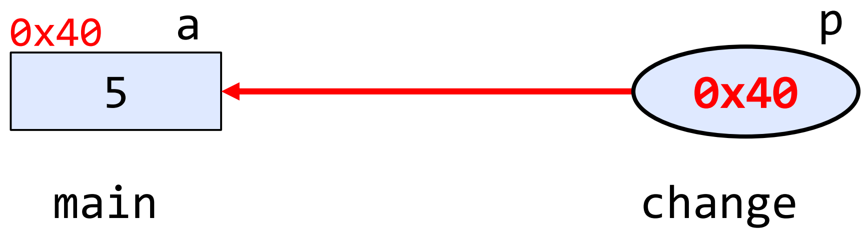
- Function Call: Allocate a space to a formal parameter(variable), Assign the address to the parameter

```
void main()
{  int a = 5;

   change(&a);

   printf("a=%d\n", a);
}
```

Pass the address of a

0x40

```
void change(int *p)
{
   *p = 10;
}
```



Memory diagram: staring change()

4) Pointer Arguments

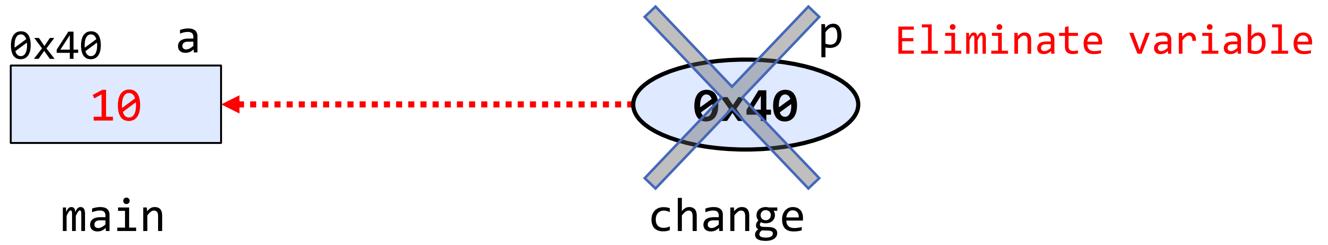
- Function body: Assign 10 to the variable pointed by p
- Function termination: Eliminate local variables (including arguments) (return allocated memory space)

```
void main()
{  int a = 5;

   change(&a);

   printf("a=%d\n", a);
}
```

```
void change(int *p)
{
   *p = 10;
}
```

Result:

```
a=10
```



Memory diagram: terminating change()

4) Pointer Arguments

- **Function arguments**
  - Call-by-value: Call a function using the value of a variable as an argument
    - ✓ Cannot change the value of the variable
  - Call-by-reference: Call a function using the address as an argument
    - ✓ Can change the value of the variable
  - However, the procedure (passing argument and control) **is identical**

4) Pointer Arguments

- **Comparison**

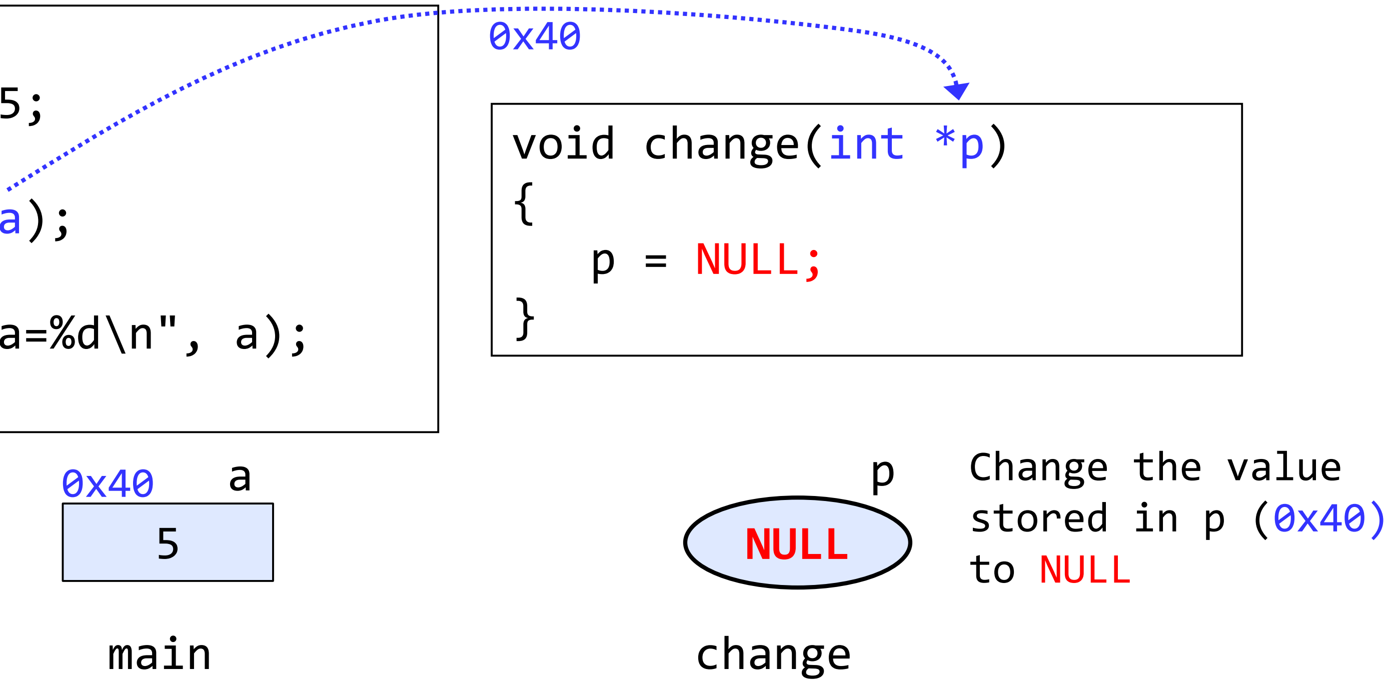
Call-by-value	Call-by-reference
`void change(int i)`	`void change(int *p)`
`{`	`{`
`   i = 10;`	`   *p = 10;`
`}`	`}`
`void main()`	`void main()`
`{  int a = 5;`	`{  int a = 5;`
`   change(a);`	`   change(&a);`
`   printf("a=%d\n", a);`	`   printf("a=%d\n", a);`
`}`	`}`
`Result:`	`Result:`
`a=5`	`a=10`

4) Pointer Arguments

- **(Example)**
  - Change the value of a local variable in change()
    The identical procedure with the previous example

```
void main()
{  int a = 5;

   change(&a);

   printf("a=%d\n", a);
}
```

```
void change(int *p)
{
   p = NULL;
}
```



Change the value stored in p (0x40) to NULL

Memory diagram: terminating change()

4) Pointer Arguments

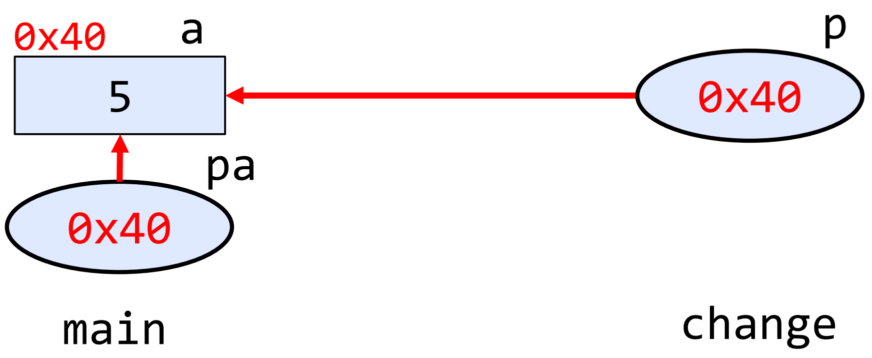
- **Example**
  - Function argument is a pointer variable (Same procedure)

```
void main()
{   int a = 5;
    int *pa = &a;

    change(pa);

    printf("a=%d\n", a);
}
```

The value of pa

0x40

```
void change(int *p)
{
    *p = 10;
}
```



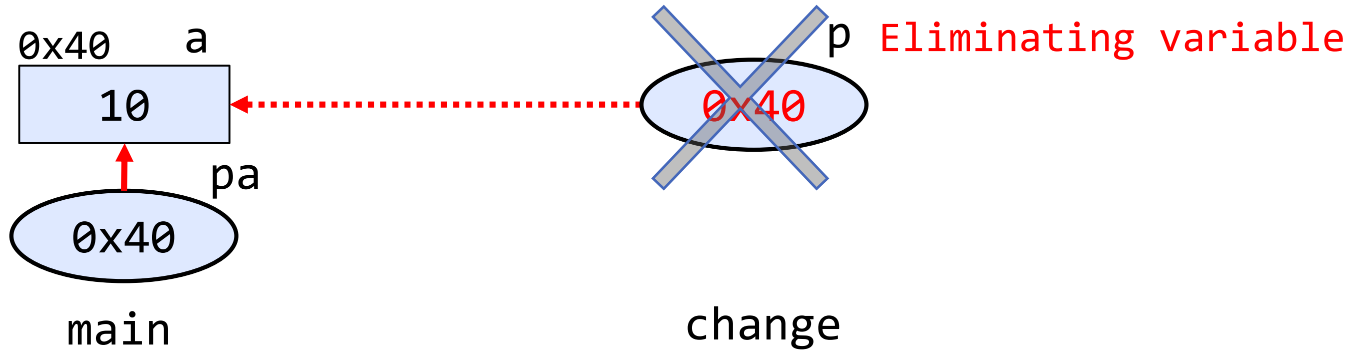
Memory diagram: starting change()

4) Pointer Arguments

- The value of a variable a in main becomes 10
- The variable p in change is eliminated

```
void main()
{  int a = 5;
   int *pa = &a;

   change(pa);

   printf("a=%d\n", a);
}
```

```
void change(int *p)
{
   *p = 10;
}
```



Memory diagram: terminating change()

4) Pointer Arguments

- **(Example) Swap two variable**
  - Assign the value of a variable y to x, then lose the value of x
    Store the value of x to a temporary variable tmp

```
void main(){
   int x = 10, y = 20, tmp;

   tmp = x;
   x = y;
   y = tmp;

   printf("%d %d", x, y);
}
```

⇨

Result

```
20 10
```

4) Pointer Arguments

- **Swap function**
  - Swap two variables
  - Using swap(), does it change the values in main()?

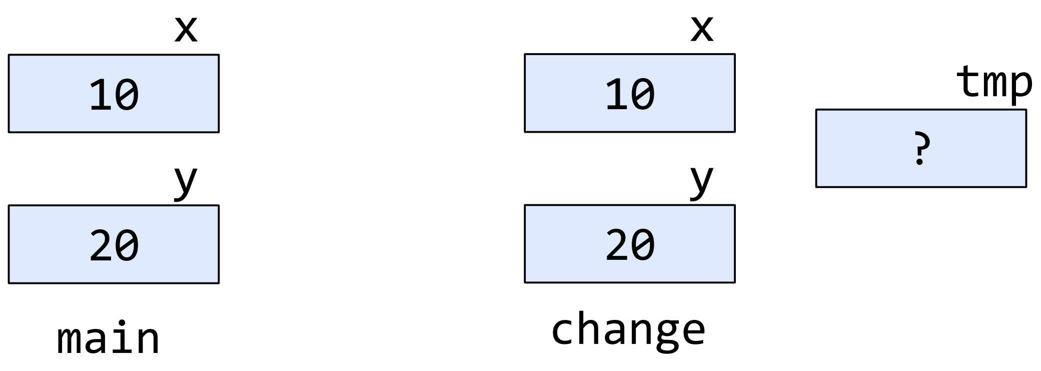
```
void swap(int x, int y){
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
void main(){
   int x = 10, y = 20;

   swap(x, y);

   printf("%d %d", x, y);
}
```

Result

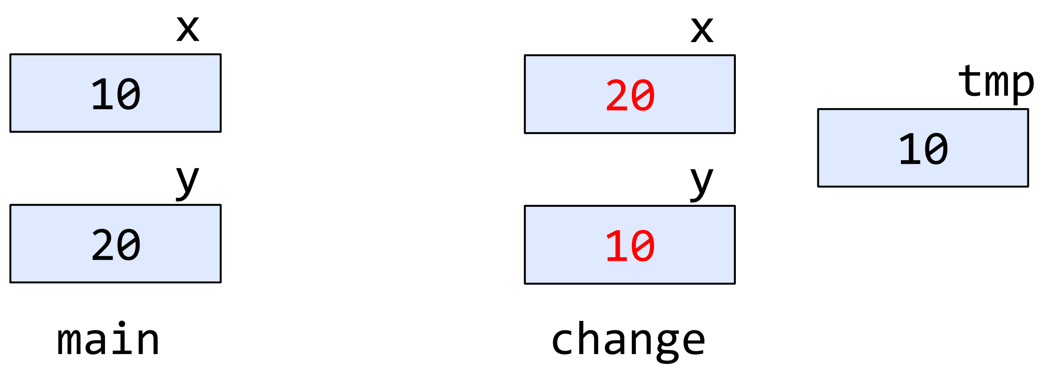
```
10 20
```

4) Pointer Arguments

- **Why not? Draw memory diagram**



Memory diagram: starting swap()



Memory diagram: terminating swap()

4) Pointer Arguments

- **Use pointers**
  - Function arguments: int pointers
  - Call-by-reference

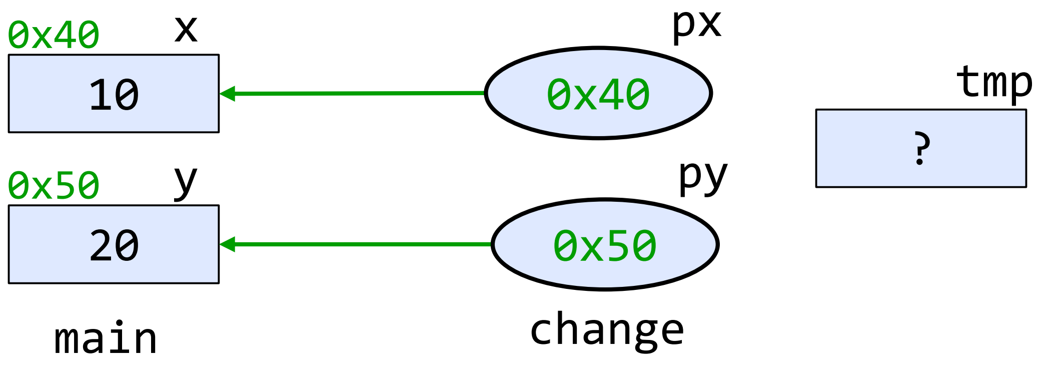
```
void swap(int *px, int *py){
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
}
void main(){
   int x = 10, y = 20;

   swap(&x, &y);

   printf("%d %d", x, y);
}
```

Result

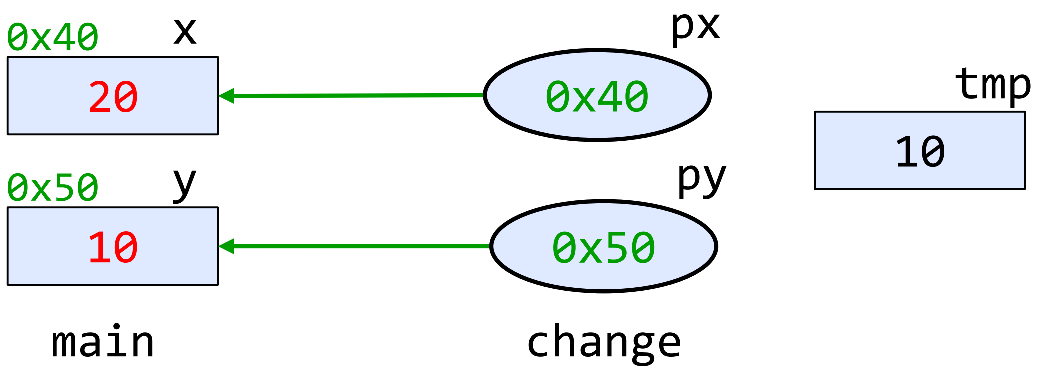
```
20 10
```

4) Pointer Arguments

- **Why? Draw memory diagram**



Memory diagram: starting swap()



Memory diagram: terminating swap()

4) Pointer Arguments

- **Use the name of an array as a function argument**
  - Change the values of an array in init(), also affect the array in main()
  - Why? Pass the starting address of an array to the function

```
void init(int ar[]){
   ar[0] = ar[1] = 0;
}
void main(){
   int ar[2]={-2,4};

   init(ar);

   printf("%d %d",ar[0],ar[1]);
}
```

Result

```
0 0
```

4) Pointer Arguments

- **Use the name of an array as a function argument**
  - Is int ar[ ] in init() an array? pointer?
  - **A pointer variable** storing the starting address of ar in main()

```
void init(int ar[]){
    ...
}
```

=

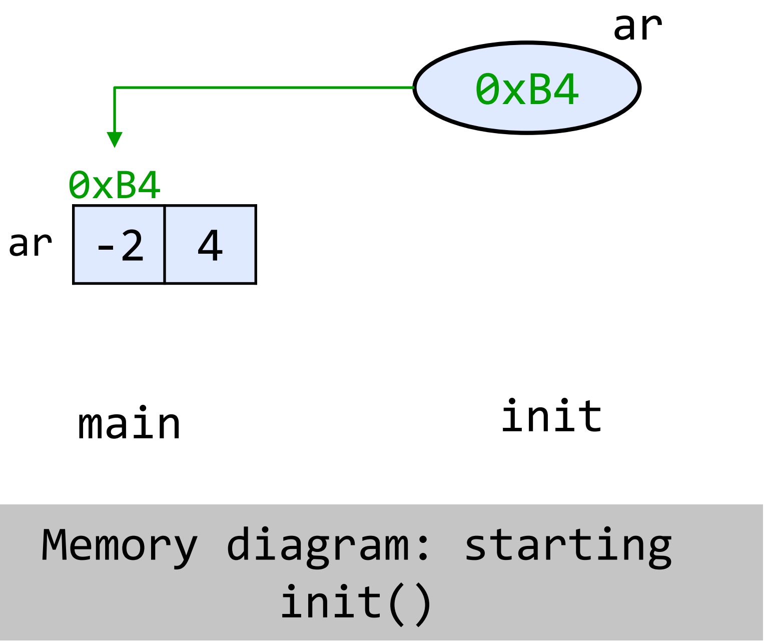
```
void init(int *ar){
    ...
}
```

  - Two are identical

4) Pointer Arguments

- **Draw a memory diagram?**

```
void init(int ar[]){
   ar[0] = ar[1] = 0;
}
void main(){
   int ar[2]={-2,4};

   init(arr);

   printf(...); // omitted
}
```



4) Pointer Arguments

- **scanf( ): why do we use &?**

```
int x;
scanf("%d", &x);
```

  - Store the value received from a user to the variable x
  - To change the value of x in scanf(), pass its address

- **printf( )?**

```
int x = 0;
printf("%d", x);
```

  - Only need the value of x

4) Pointer Arguments

- **scanf( ): should we always use &?**
  - If it is an address, it will work

```
int x[5], *p=&x[2];

scanf("%d", &x[0]);
scanf("%d", p);
scanf("%d", p-1);

printf("%d %d %d",x[0],x[1],x[2]);
```

Input 1

```
1 2 3
```

Output 1

```
1 3 2
```

Input 2

```
-1 4 9
```

Output 2

```
-1 9 4
```

4) Pointer Arguments

- **Function that returns an address**
  - Use * , indicating that it returns the address

```
void main(){
   int ar[5]={2,1,3,0,4};
   int *p1;

   p1 = next_addr(&ar[1]);

   printf("%d",*p1);
}
```

```
int *next_addr(int *p)
{
   return p+1;
}
```

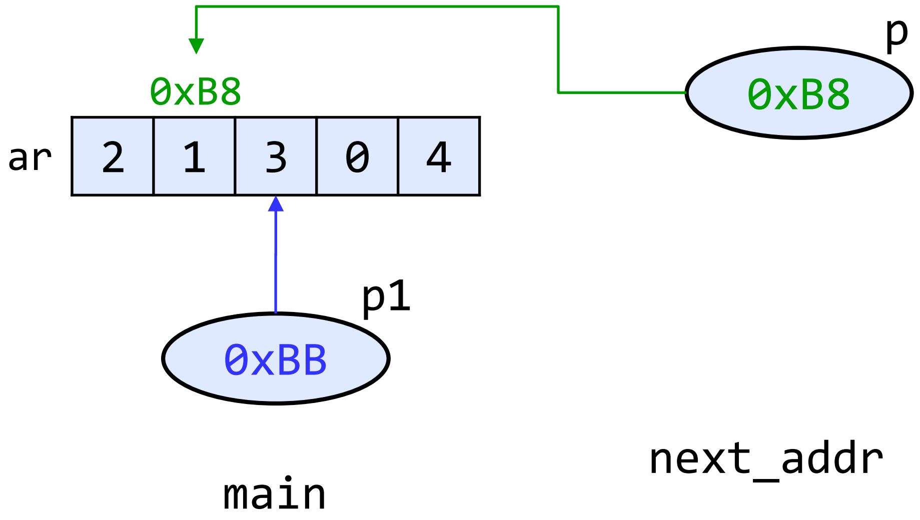
Result

```
3
```

4) Pointer Arguments

- **Memory diagram?**

```
int *next_addr(int *p)
{
   return p+1;
}

void main(){
   int ar[5]={2,1,3,0,4};
   int *p1;

   p1 = next_addr(&ar[1]);

   printf("%d",*p1);
}
```



Memory diagram: terminating next_addr()

4) Pointer Arguments

- **[Practice]**
  - Receive the address of two int variables, Return the address of the variable that is smaller. (Assume: two values are different)
    - ✓ Use main() below

```
void main(){
   int ar[5]={2,1,3,0,4};
   int *p1;

   p1 = smaller(&ar[1], &ar[3]);

   printf("%d",*p1);
}
```

```
?    smaller (    ?    )
{
           ?
}
```

Result

```
0
```

Outline

1) Pointers?

2) Arrays and Pointers

3) Pointer Operation

4) Pointer Arguments

5) **Arrays of Pointers**

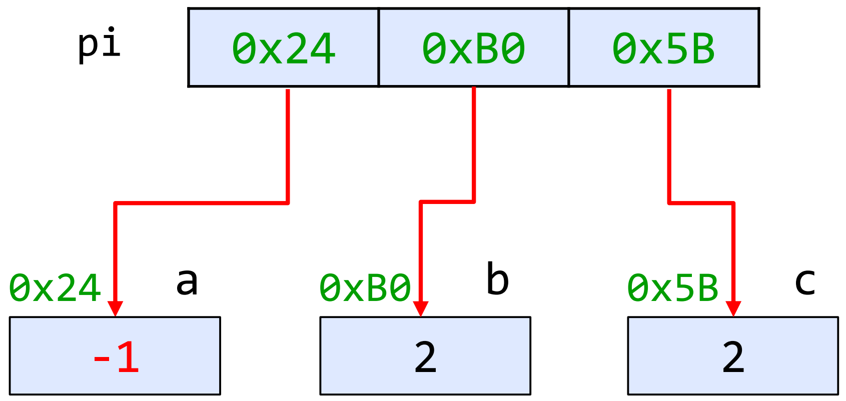
5) Arrays of Pointers

- **Arrays of Pointers**
  - Pointer variables can be declared as an array
- **Declare arrays of pointers**
  - Pointer declaration + Array declaration

```
void main(){
   int *pi[3];   // array of pointers declaration
   int a=1, b=2, c=3, i;

   pi[0] = &a;   // Element of an array is int pointer
   pi[1] = &b, pi[2] = &c;

   *pi[0] = -1; // *pi[0]은 *(pi[0]) ? (*pi)[0] ?

   for( i=0; i < 3 ; ++i )
      printf("%p %d\n", pi[i], *pi[i]);
}
```

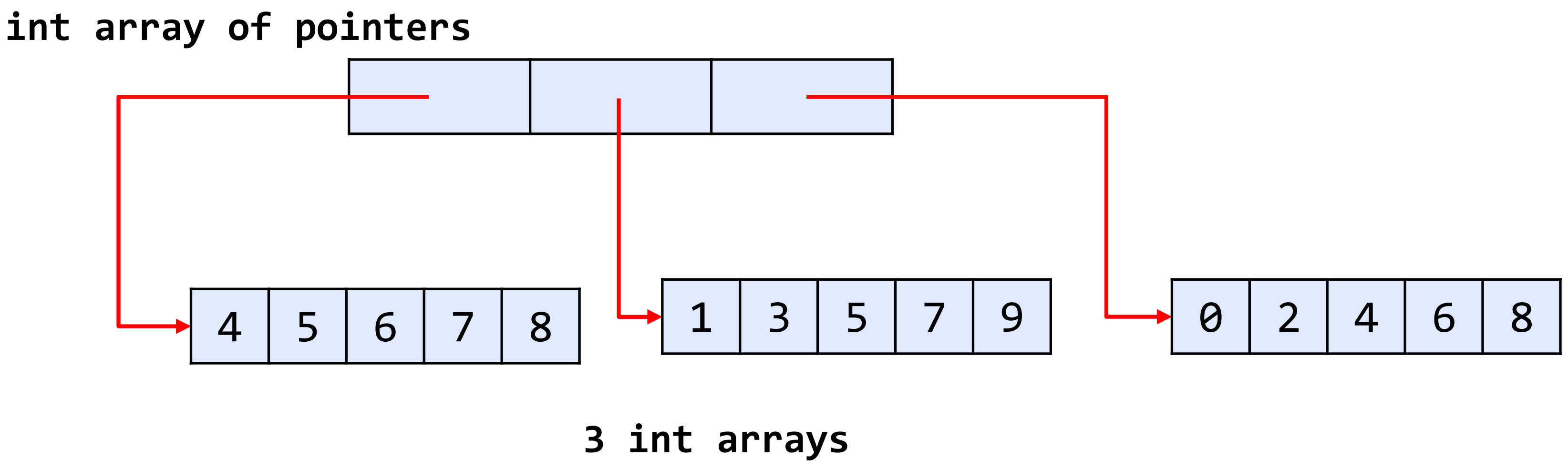
5) Arrays of Pointers

- **Memory diagram**



5) Arrays of Pointers

- **[Practice] Write a program**
  - Declare 3 int arrays (each: size 5), initialize them as shown below
  - Declare 1 int **array of pointers** (size 3)
  - Assign int array to **array of pointers**

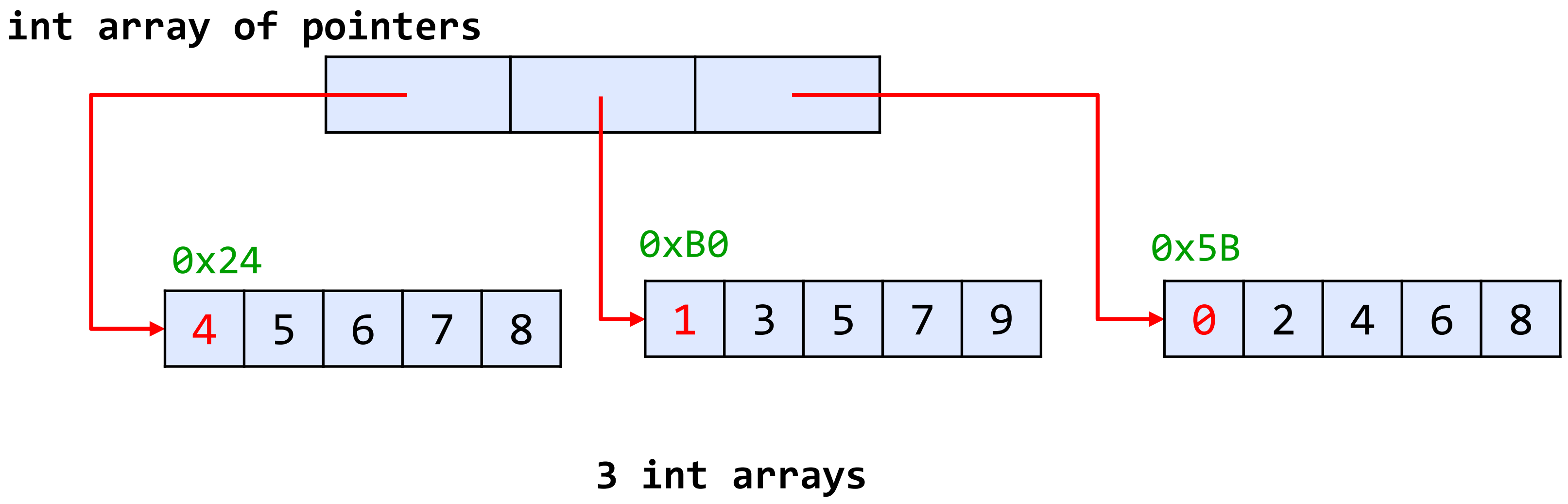


5) Arrays of Pointers

- Use int array of pointers, Print the address and value of the 1st element of each int array
  - ✓ Cannot use the name of int array

Result

```
0x24 4 0xB0 1 0x5B 0
```


int array of pointers
0x24
4 5 6 7 8
0xB0
1 3 5 7 9
0x5B
0 2 4 6 8
3 int arrays