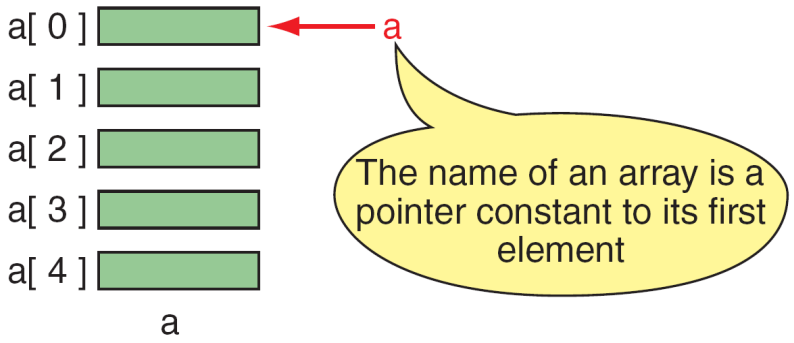
C Programming (CSE2035)
(Chap9. Pointer Applications)

Sungwon Jung, Ph.D.

Bigdata Processing & DB LAB
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Arrays and pointers

- 배열 이름은 첫 번째 요소의 주소 값을 나타낸다.
- **int a[5];** 와 같이 선언되었을 때 메모리의 상황은 다음과 같다.



- 배열 이름인 a는 배열의 첫 번째 원소인 a[0]을 가리키고 있다.
- 즉, a 는 &a[0] 값을 가지고 있다.

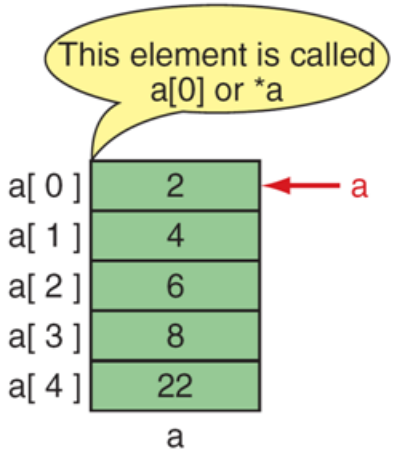
Arrays and pointers

- 다음은 a와 &a[0]이 같은 값을 가짐을 보이는 예제이다.
- a가 a[0]의 주소값을 가지고 있기 때문에, 당연히 a[0]은 *a를 통해서도 접근할 수 있다.

```c
#include <stdio.h>

int main(void)
{
        int a[5]={2, 4, 6, 8, 22};

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

        return 0;
}
```

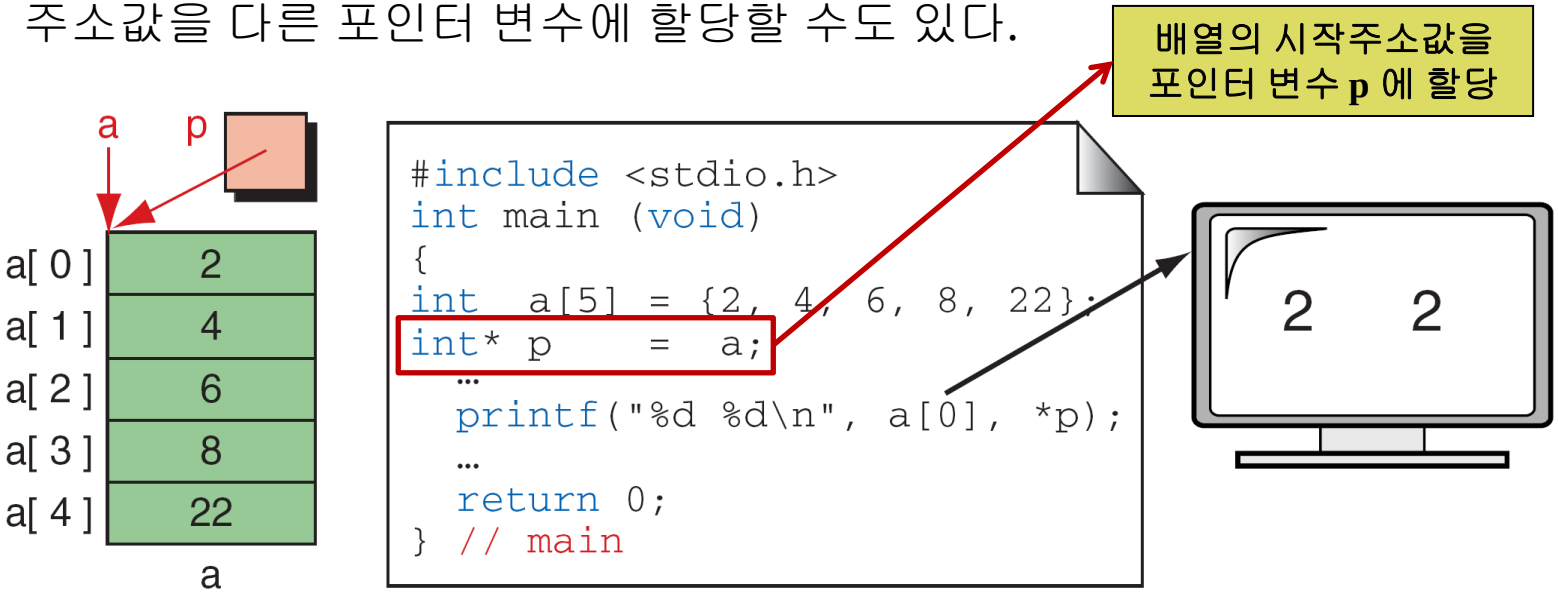
```
[root@mclab chap10]# ./chap10-1
a : -1076213348,      &a[0] : -1076213348
*a : 2,       a[0] : 2
[root@mclab chap10]#
```

*a와 a[0] 모두 배열 a의 첫번째 원소를 가리킨다.



Arrays and pointers

- 배열 이름 a는 주소값을 갖는 포인터이다.
  - 주소값을 다른 포인터 변수에 할당할 수도 있다.

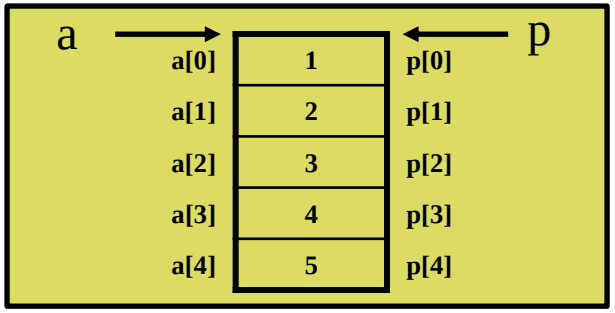


- 단, 배열 이름 a는 배열의 시작 주소를 갖도록 고정되어 있기 때문에 a의 값을 변경할 수는 없다.

Arrays and pointers

- 포인터 변수가 배열을 참조하도록 함으로써 포인터 변수를 배열처럼 사용하는 것도 가능하다.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          int i;
6          int sum1=0;
7          int sum2=0;
8          int a[5]={1,2,3,4,5};
9          int *p=a;
10
11         for(i=0; i<5; i++) {
12                 sum1 += a[i];
13                 sum2 += p[i];
14         }
15
16         printf("%d == %d\n", sum1, sum2);
17
18         return 0;
19 }
```



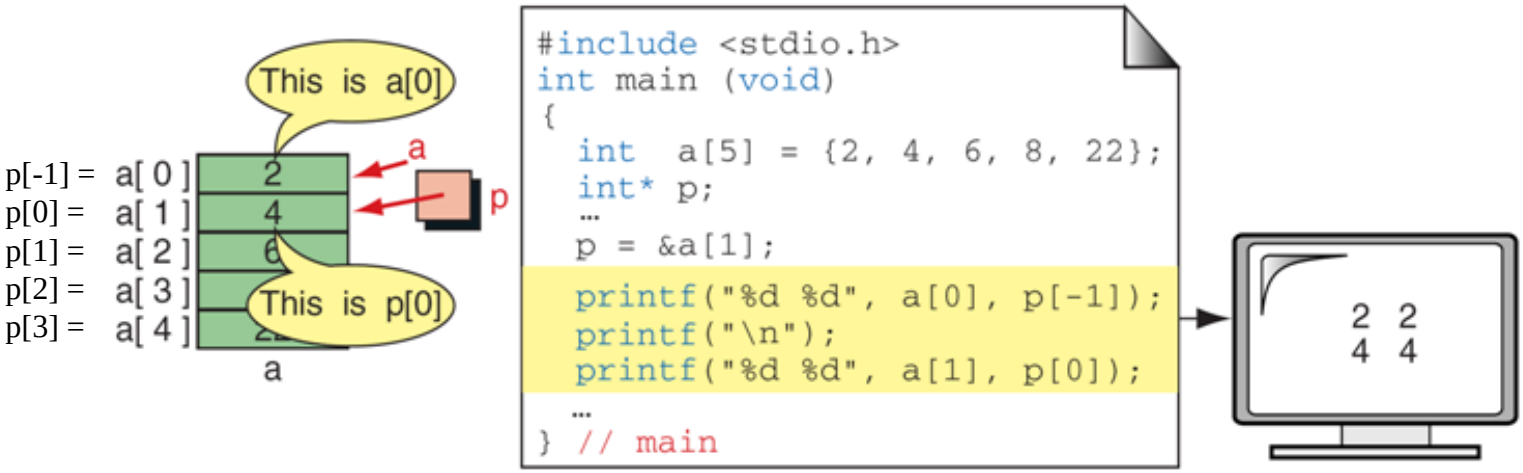
포인터변수 **p**도 **a**와 같은 값을 갖는다. 즉, 배열의 시작 주소를 갖는다.

a[i]와 **p[i]**는 같은 결과를 갖는다.
결국 **p**를 이용해서도 배열에 접근할 수 있음을 알 수 있다.

```
[root@mclab chap10]# ./chap10-2
15 == 15
[root@mclab chap10]#
```

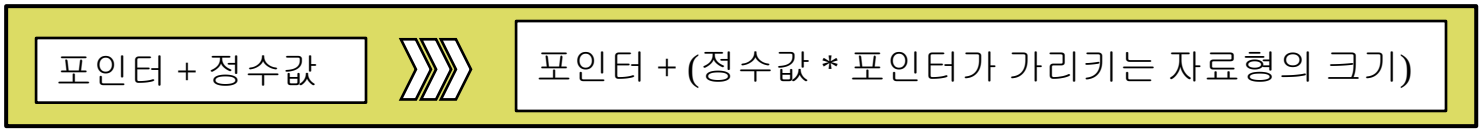
Arrays and pointers

- 배열의 중간 부분의 주소를 **pointer** 변수에 할당할 수도 있다.
  - 이 경우 같은 메모리 공간을 access하기 위해서는 서로 다른 index를 사용해야 한다.

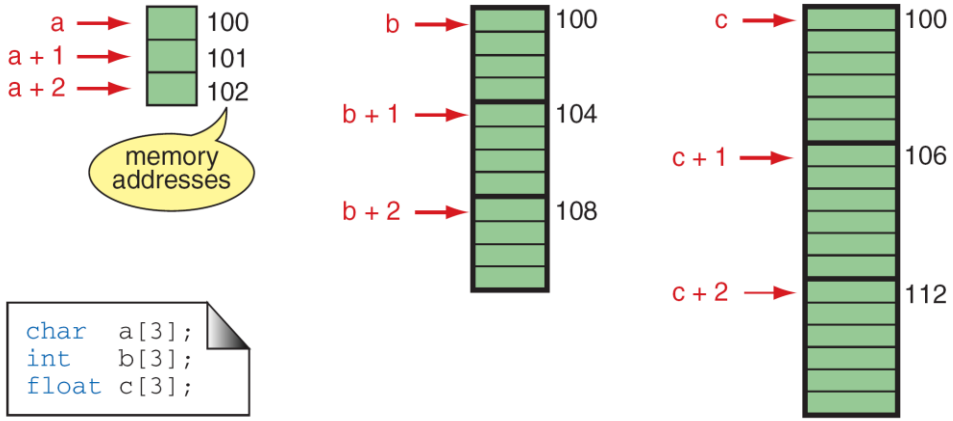


Pointer arithmetic and arrays

- **Pointers and One-Dimensional Arrays**
  - p가 특정 데이터타입에 대한 포인터 변수일 때
    - 수식 p + 1은 해당 데이터타입의 다음 변수를 나타낸다.
  - 포인터 연산 p + 1에서 더해지는 값 1은 현재 포인터의 data type의 size만큼을 더하는 것이다.

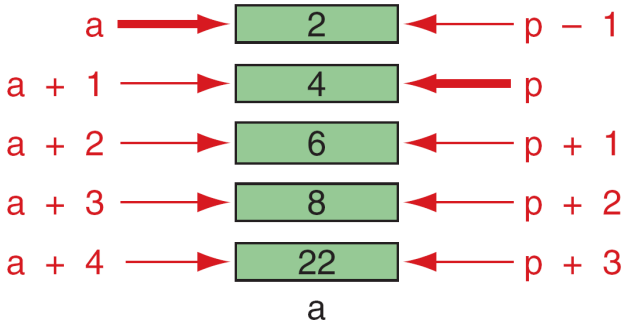


  - 다른 타입을 갖는 배열에 대한 포인터 연산의 결과

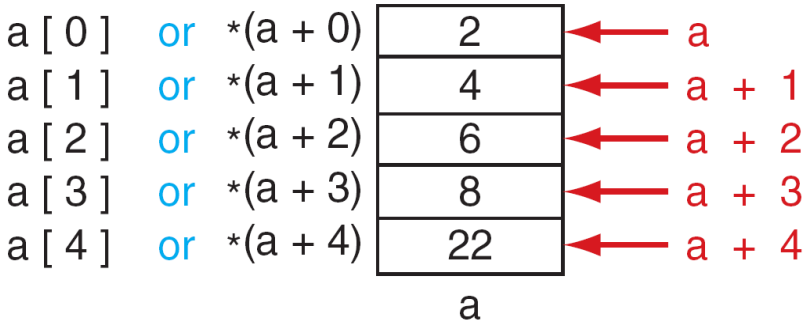


Pointer arithmetic and arrays

- 포인터 변수 p가 배열의 두 번째 원소를 참조하도록 초기화 한 경우
  - (-) 연산은 (+) 연산과 마찬가지로 해당되는 type의 하나 이전의 element를 가리키게 된다.

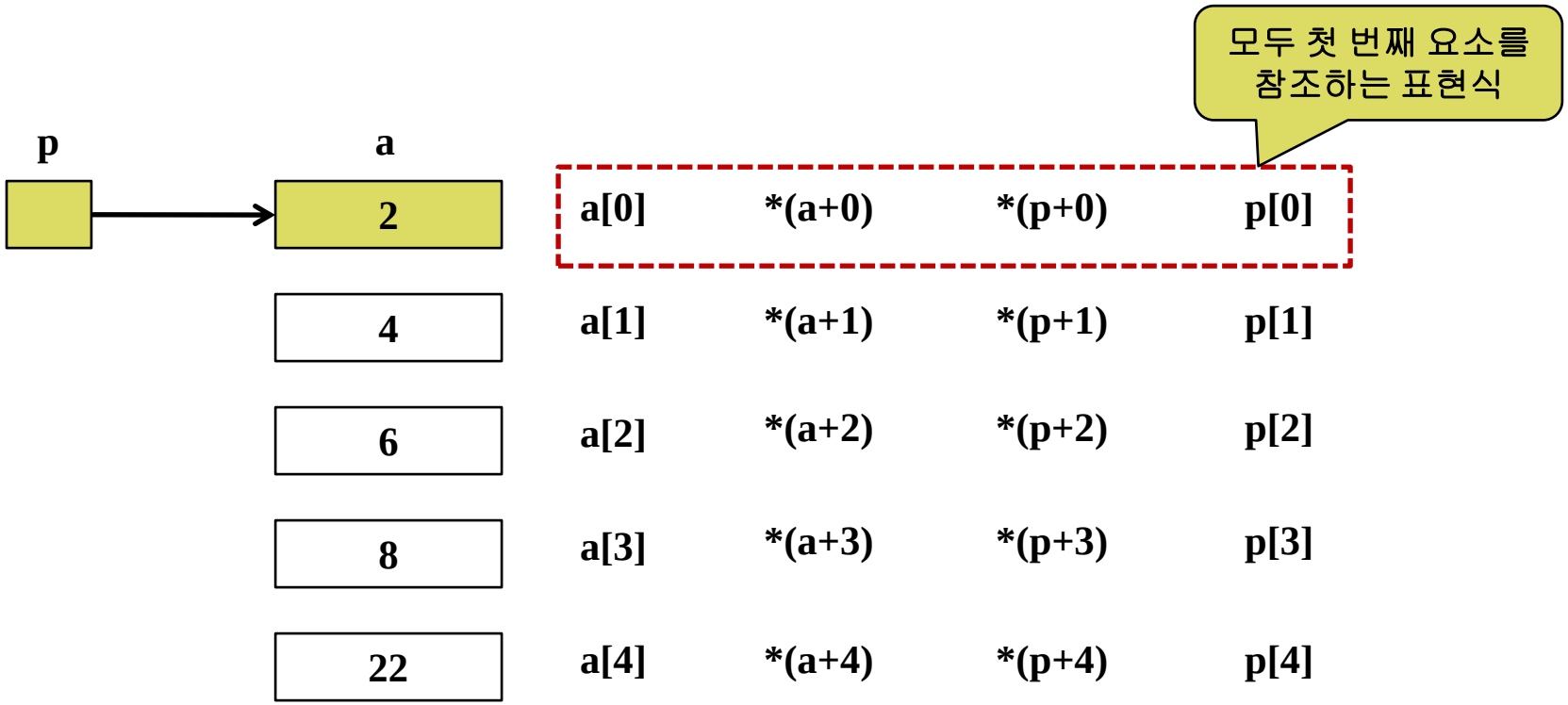


- 역참조 연산자를 이용한 사용도 가능하다.
  - a+1은 &a[1]와 같으므로 *(a+1)은 a[1]와 같다.



Pointer arithmetic and arrays

- 배열의 요소를 참조하는 여러가지 방법



Pointer arithmetic and arrays

- **Pointers And Other Operators**
  - 포인터 연산이란 포인터 값을 증가 혹은 감소시키는 연산을 말한다.
  - 포인터 연산이 가능한 연산은 제한되어 있다.
    - 포인터 연산에 따른 실질적인 값의 변화는 포인터 타입에 따라 다르다.
    - 포인터를 나누거나 곱하는 연산은 불가능하다.
    - 예제 프로그램 – 포인터 연산을 이용한 값의 변화를 출력한다.

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5         char* ptr1=0;
 6         int* ptr2=0;
 7         double* ptr3=0;
 8
 9         printf("%d, %d, %d\n", ptr1++, ptr2++, ptr3++);
10         printf("%d, %d, %d\n", ptr1, ptr2, ptr3);
11
12         return 0;
13 }
```

```
[root@mclab chap10]# ./chap10-3
0, 0, 0
1, 4, 8
[root@mclab chap10]#
```

Pointer arithmetic and arrays

- **Pointers And Other Operators**
  - **관계 연산자(Relational operators)**는 양쪽의 피 연산자가 모두 포인터인 경우에만 사용이 가능하다.

```
p1 >= p2        p1 != p2
```

  - 포인터에서 관계연산자를 사용하는 가장 일반적인 경우는 포인터와 NULL 상수를 비교하는 경우이다.

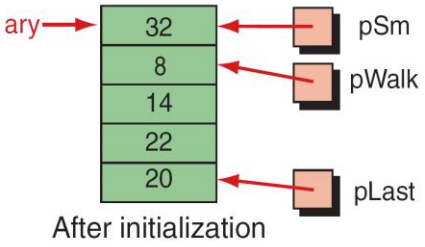
Long Form	Short Form
if (ptr == NULL) if (ptr != NULL)	if (!ptr) if (ptr)

Pointer arithmetic and arrays

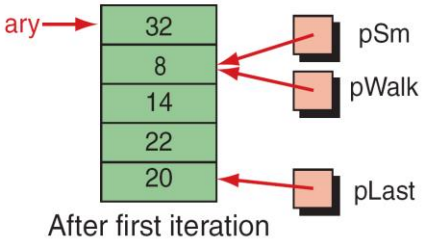
- 포인터를 이용하여 배열의 원소 중 가장 작은 값을 가지는 원소를 찾는 프로그램의 일부이다. 그림은 프로그램이 실행되는 과정을 보인다.

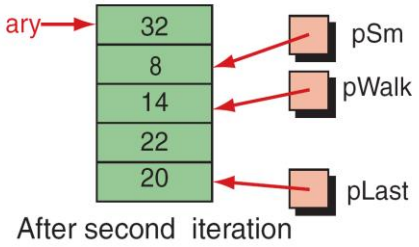
```
pLast = ary + arySize - 1;
for (pSm = ary, pWalk = ary + 1;
     pWalk <= pLast;
     pWalk++)
   if (*pWalk < *pSm)
       pSm = pWalk;
```

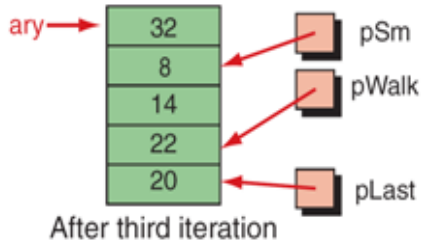
- pSm은 가장 작은 원소를 가리키기 위해 사용된다.
- pWalk은 가장 작은 원소를 찾기 위해 배열을 순서대로 탐색한다.
- pLast는 마지막 원소를 가리킨다.

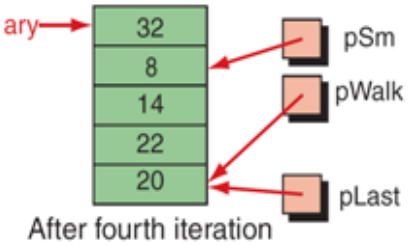


pSm is smallest. It tracks the smallest value. | **pWalk** is walker. It moves to find smallest.









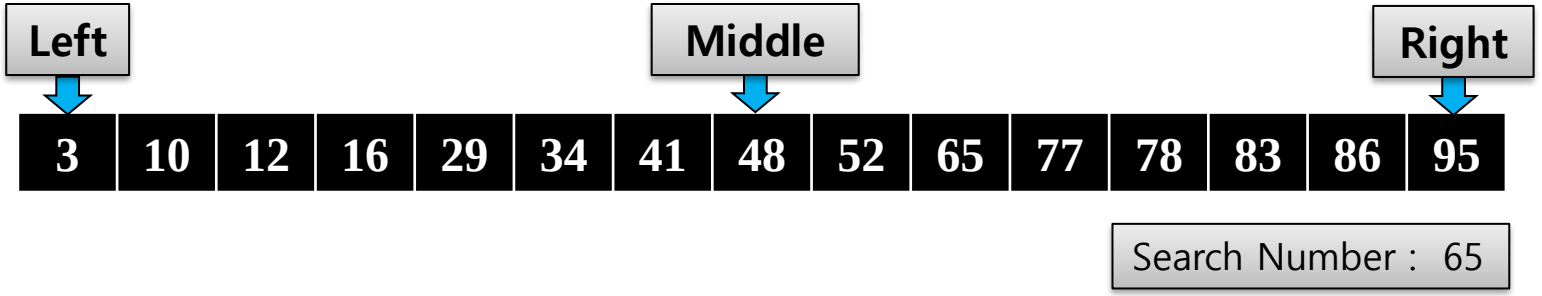
Pointer arithmetic and arrays

- **Pointer subtraction**
  - 포인터 값끼리의 뺄셈은 두 개의 주소 값의 차를 자료형의 크기로 나누어 반환한다.
  - 포인터 값끼리의 덧셈 연산은 불가능하다
  - 포인터 값의 곱셈 및 나눗셈 연산은 불가능하다

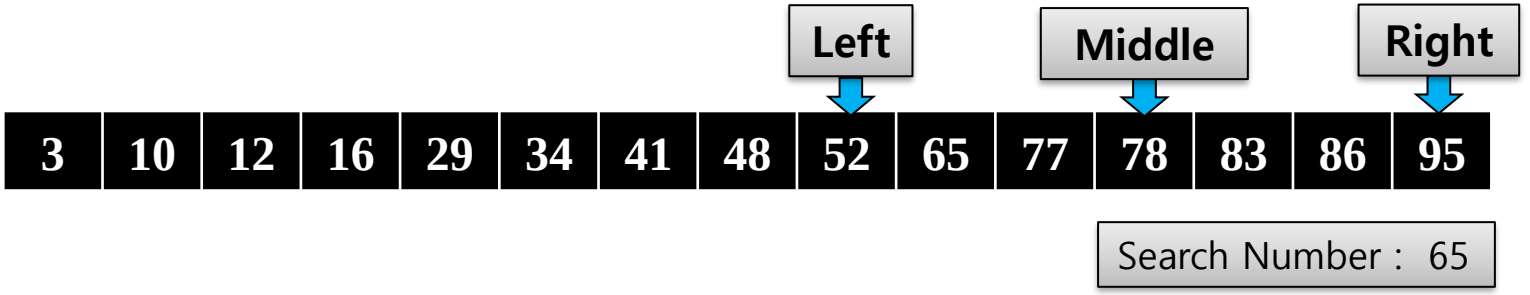
Type of Left Operand	Operator	Type of Right Operand	Type of result
Pointer	+	Pointer	Error
Pointer	-	Pointer	Integer(offset)
Pointer	+	Int	Address
Pointer	-	Int	Address

Pointer arithmetic and arrays

- **Binary Search**
  - 오름차순으로 정렬된 배열에서 원하는 숫자의 위치를 탐색하는 방법
  - Left는 배열의 첫 번째 자리, Right는 마지막 자리, Middle은 중간 위치를 나타낸다.

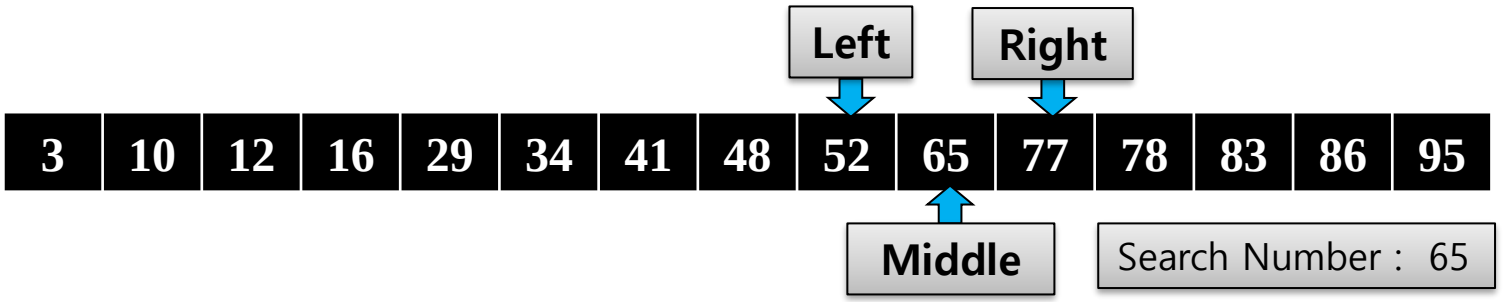


  - Middle 위치의 값과 Search Number를 비교하여 Middle 위치의 값이 작을 경우에는 Left의 위치를 Middle 위치보다 1 크게 잡는다.

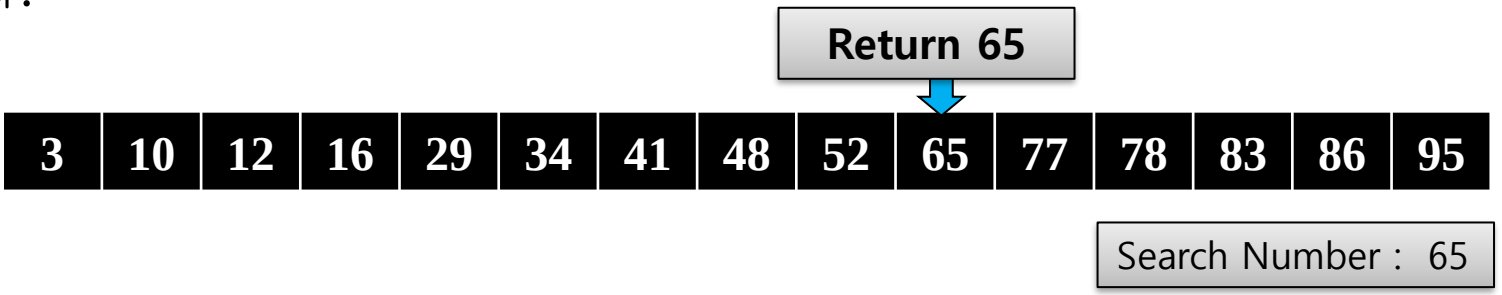


Pointer arithmetic and arrays

- Search Number가 Middle에 위치한 값 보다 작을 경우에는 Right의 위치를 Middle 위치보다 1 작게 잡는다.



- Middle에 위치한 값과 Search Number의 값이 같으면 Middle을 반환한다.



Pointer arithmetic and arrays

- 배열과 index syntax를 사용할 경우, Binary search의 middle index의 계산은 다음과 같이 수행할 수 있다.
  - Middle = (Left + Right) / 2;

- 그러나 middle, left, right가 배열이 가진 특정 원소의 주소 값이라면, 두 포인터의 덧셈과 나눗셈 연산이 불가능하기 때문에 다음과 같은 주소 값 계산 방식이 필요하다.
  - midPtr = firstPtr + (LastPtr – FirstPtr) / 2;

- 위 표현식에서 피연산자들의 자료형은 다음과 같은 순서로 계산된다

Address = Address + (offset) / 2

= Address+ (int) / int

= Address + int

Pointer arithmetic and arrays

- **Using Pointer Arithmetic**
  - 다음은 포인터를 이용한 binary search 함수이다.

```
1  /* ====================binary Search====================
2     Search an ordered list using Binary Search
3        Pre   list must contain at least one element
4              endPtr is pointer to largest element in list
5              target is value of element being sought
6        Post  FOUND: locnPtr pointer to target element
7                     return 1 (found)
8              !FOUND: locnPtr = element below or above target
9                      return 0 (not found)
10 */
11 int binarySearch (int list[], int*  endPtr,
12                    int target, int** locnPtr)
13 {
14 // Local Declarations
15    int* firstPtr;
16    int* midPtr;
17    int* lastPtr;
18
```

```
int main() {

 int a[10] = {1,2,3, ... ,10};
 int *p, i, key;
 ...

 key = 7;
 i = binarySearch(a, a+9, key, &p);
 ...
 retrun 0;
}
```

- **list[ ]**는 정렬된 data set
- **endPtr**은 현재 list배열의 마지막 원소를 참조
- **target**은 list에서 찾아야 할 값
- **locnPtr**은 찾은 원소를 참조할 포인터 변수

Pointer arithmetic and arrays

```
19  // Statements
20     firstPtr = list;
21     lastPtr  = endPtr;
22     while (firstPtr <= lastPtr)
23        {
24        midPtr = firstPtr + (lastPtr - firstPtr) / 2;
25        if (target > *midPtr)
26           // look in upper half
27           firstPtr = midPtr + 1;
28        else if (target < *midPtr)
29           // look in lower half
30           lastPtr = midPtr - 1;
31        else
32           // found equal: force exit
33           firstPtr = lastPtr + 1;
34        } // end while
35     *locnPtr = midPtr;
36     return (target == *midPtr);
37  }  // binarySearch
```

list의 중간 원소를 지정한다

list의 중간 이후 부분일 경우

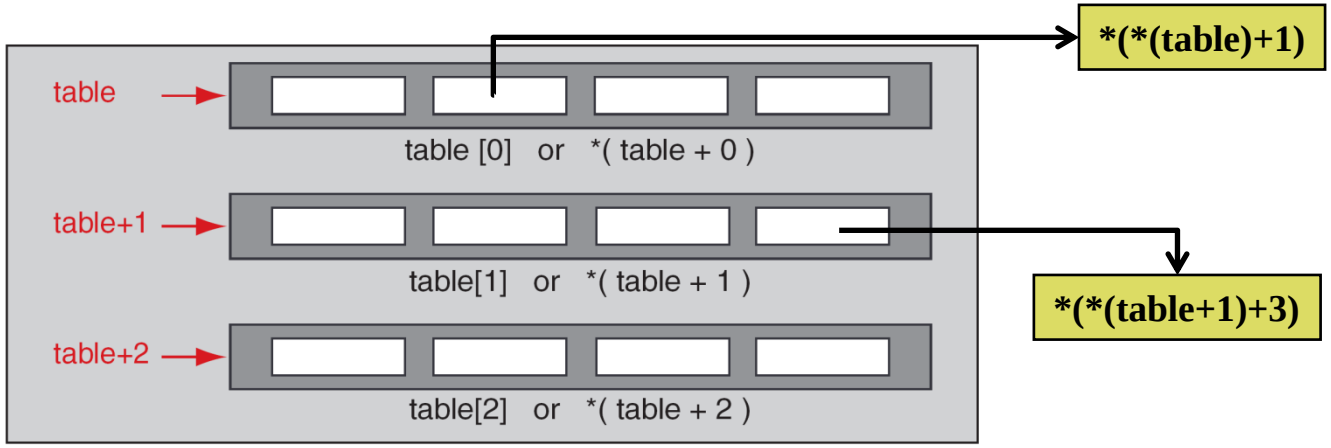
list의 중간 이전 부분일 경우

target == *midPtr일 경우

찾았는지 여부를 리턴 (찾았다면 target과 *midPtr 이 같으므로 1을 리턴)

Pointer arithmetic and arrays

- **Pointers And Two-Dimensional Arrays**
  - 이차원 배열 table의 원소 table[1][3]을 포인터를 사용하여 접근한다면 다음과 같은 표현이 가능하다 ➔ *(*(table +i) + j)



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