# CS265
# Advanced Programming Techniques

## C Pointers and Arrays

# C Arrays

- Declare an array using

```
int a[10];
```

or

```
#DEFINE MAX 10

int a[MAX];
```

or with a variable-length

```
int i, n;
printf(”Enter the size of the array? ");
scanf("%d", &n);
int a[n];   /* C99 has variable-length arrays */
```

- Use subscripts `0` to `n-1` to access array elements, for array of size `n`

```
a[0], … , a[n-1]
```

# Typical Array Loops

Examples of typical operations on an array `a` of length `N`:

- Clear the array

```
for (i = 0; i < N; i++)
  a[i] = 0;
```

- Read data into an array

```
for (i = 0; i < N; i++)
  scanf("%d", &a[i]);
```

- Sum the elements of an array

```
for (i = 0; i < N; i++)
  sum += a[i];
```

**WARNING**
**C does not require that array subscript bounds be checked – it is up to the programmer**

# C Arrays – How to initialize

- We can use an array initializer

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- If the initializer is shorter than the array, the remaining elements will get 0

```
int a[10] = {1, 2, 3, 4, 5, 6};
```

- Using the trick above, we can easily initialize all elements to 0s

```
int a[10] = {0}; // define 1 value only
```

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

# C Arrays – How to Initialize with Designated Initializers

- With relatively few elements and a large array, it is tedious and error prone to write initializers
- C99 offers designated initializers
- C99 offers initializers that only specify specific elements

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- The order in which the elements are listed does not matter

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

- If the length of the array is omitted, the compiler will deduce the length of the array from the largest designator

```
int a[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29}; // size = 24
```

- An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

```
int a[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6}; // mixed mode
```

# Arrays – the `sizeof` operator

- The `sizeof` operator can determine the size of an array (in bytes)

- If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires 4 bytes)

- Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```

- Some programmers use this expression when the length of the array is needed.

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

**length of the array**

# Arrays – `sizeof` operator

- Defining a macro for the size calculation is often helpful:

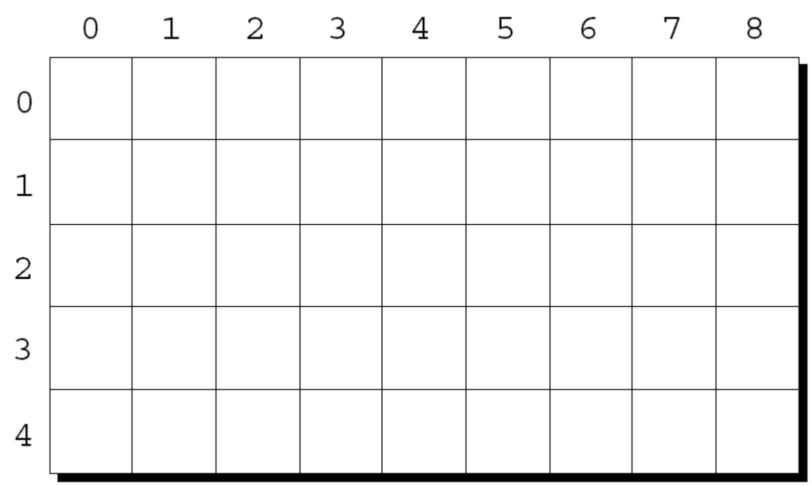
```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
  a[i] = 0;
```

# Multi-Dimensional Arrays

- An array may have any number of dimensions
- The following declaration creates a two-dimensional array

```
int m[5][9];
```

- `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:



# Multi-Dimensional arrays – How To Access

- To access the element of `m` in row `i`, column `j`, we must write

```
m[i][j]
```

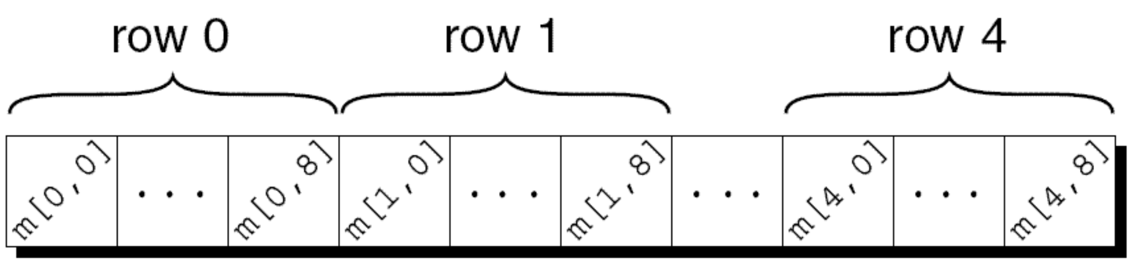
- Be careful not to write

```
m[i,j]
```

# Multi-Dimensional Arrays – How They Are Stored

- C stores arrays in ***row-major order***
  - with row 0 first
  - then row 1
  - and so forth

- How the `m` array is stored:



# Multi-Dimensional Arrays – How To Initialize

- We can use one-dimensional initializers

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- If an initializer isn't large enough to fill all rows in a multidimensional array, the remaining elements are given the value 0.

```
/* fills the first 3 rows only

int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
//fills the first 3 rows and some columns only

int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1}};
```

# Multi-Dimensional Arrays – How To Initialize

- We can even omit the inner braces (this is risky!)

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

- We can use C99 initializers, where all elements not specified default to 0
- Good for sparse arrays

```
int ident[3][3] = {[0][0] = 1, [1][1] = 1, [2][2]=1};
```

# Constant Arrays

- An array can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that's been declared `const` should not be modified by the program

**Any variable defined as `const` tells the compiler that the value of the variable should not change in the program.**

✓ **Good documentation**
✓ **Helps the compiler catch errors**

# Pointers and Arrays

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

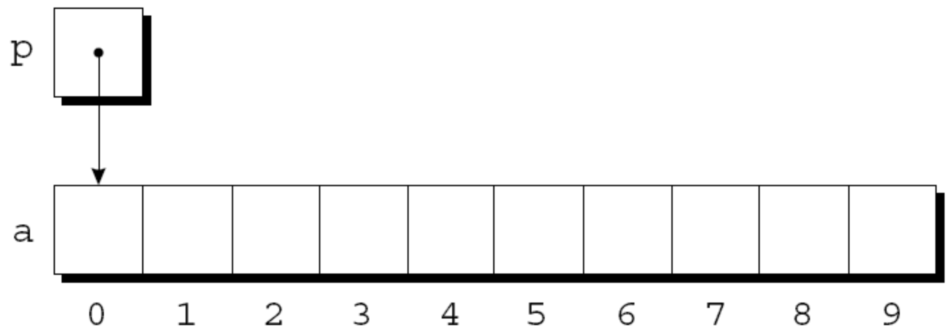
**To traverse all elements in an array**

| Option 1 | Option 2 |
| --- | --- |
| **Use array subscripts** | **Use pointers** |

# Updating via pointers

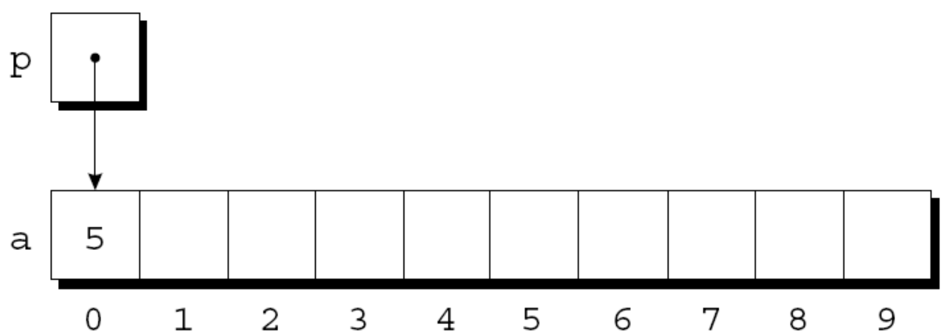
- Pointers can point to array elements:

```
int a[10], *p;
p = &a[0];
```




- We can update via a pointer

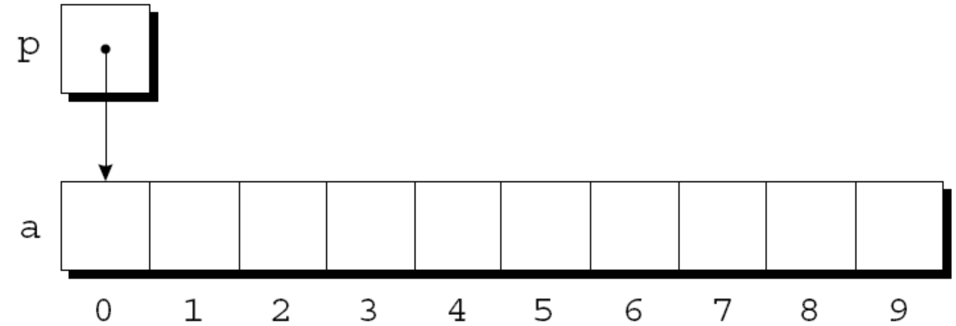
```
*p=5
```



# Pointer Arithmetic

- We can see the other elements of the array using a pointer `p` by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`.
- C supports three (and only three) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

# Pointer Arithmetic




Assume that the following:

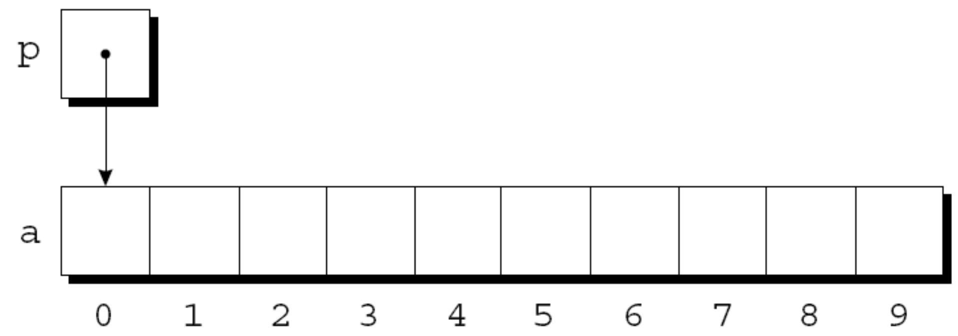
```
int a[10], *p, *q, i;
```

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

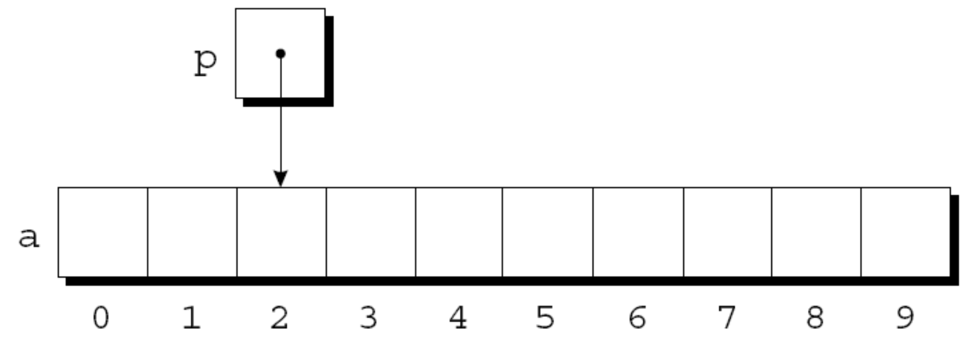
# Pointer Arithmetic

Assume that the following:

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




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

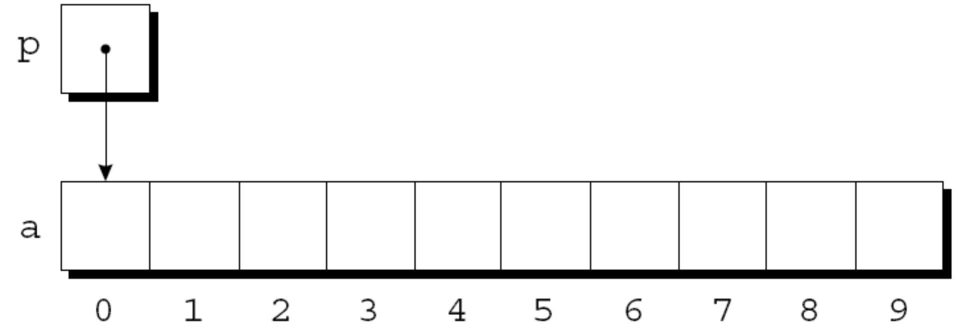



```
q = p + 3;
```

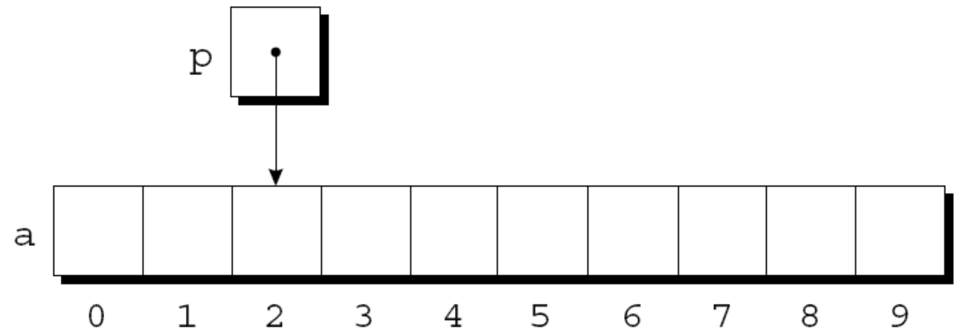
# Pointer Arithmetic

Assume that the following:

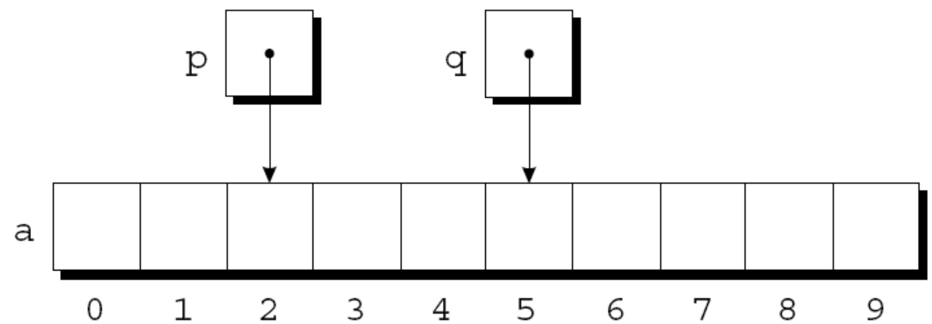
```
int a[10], *p, *q, i;
```




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




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

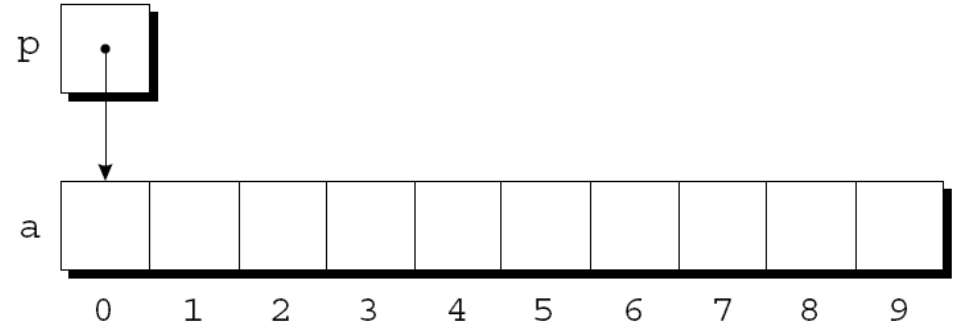



```
p += 6;
```

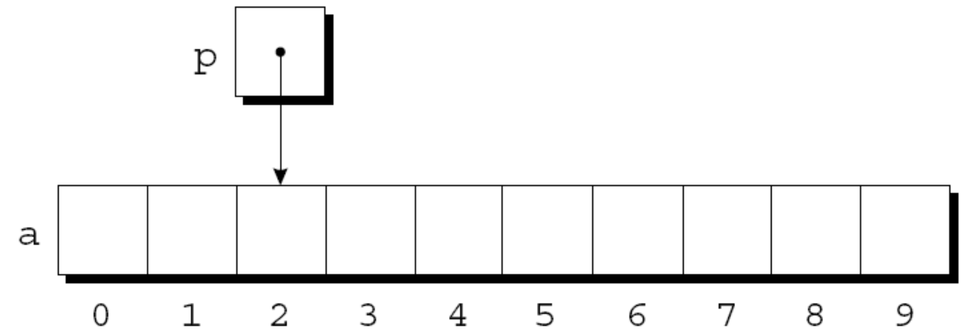
# Pointer Arithmetic

Assume that the following:

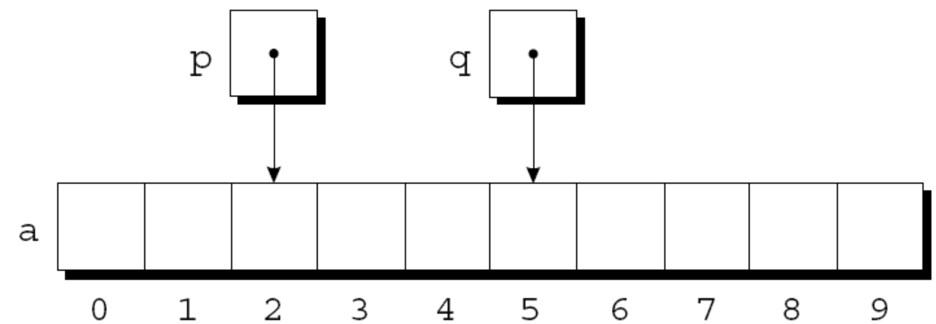
```
int a[10], *p, *q, i;
```



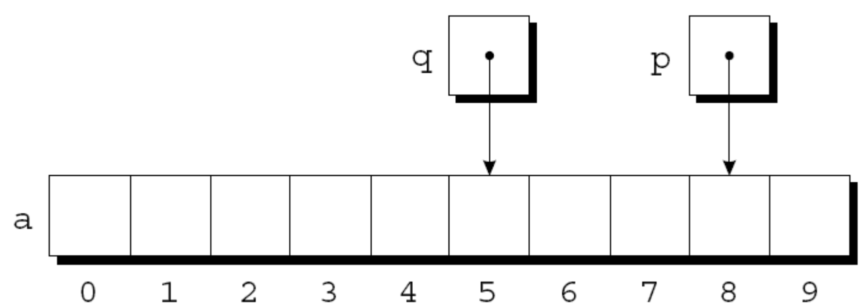

```
p = &a[2];
```




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

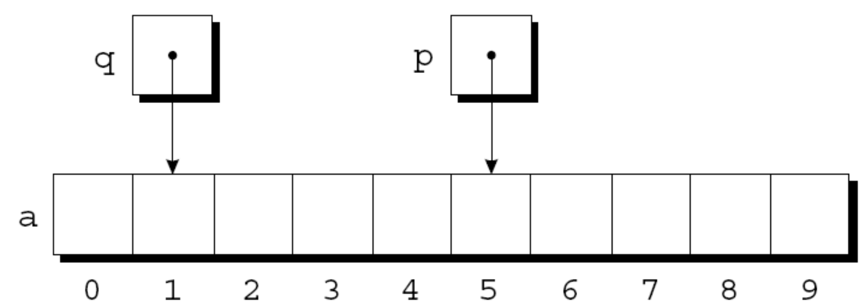



```
p += 6;
```



# Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.




- Example:

```
p = &a[5];
q = &a[1];
```

```
i = p - q;   /* i is 4  */
i = q - p;   /* i is -4 */
```

# Note: You cannot add two pointers

If p and q are pointers,

- This is supported

  p – q

- This is NOT supported

  p + q

**WARNING**

**Only difference is supported**

# Note: Can a pointer value be negative?

- The valid values for a pointer are entirely implementation-dependent, so, yes, a pointer could be negative

# Comparing Pointers

- Pointers can be compared using
  - the relational operators `<`, `<=`, `>`, `>=`
  - the equality operators `==` and `!=`
- Comparisons are meaningful only for pointers to elements of the same array
- The outcome of the comparison depends on the relative positions of the two elements in the array
- After the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p <= q` is 0 and the value of `p >= q` is 1

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

# Using Pointers for Array Processing

```
#define N 10
   …
   int a[N], sum, *p;
   …
   sum = 0;
   for (p = &a[0]; p < &a[N]; p++)
     sum += *p;
```

**This element does not exist**

- The condition `p < &a[N]` in the `for` statement deserves special mention
- It’s legal to apply the address operator to `a[N]`, even though this element doesn’t exist

# Pointers or Subscripts?

- Pointer arithmetic may save execution time
- However, some C compilers produce better code for loops that rely on subscripting

# Combining the * and ++ Operators

- C programmers often combine the `*` (indirection) and `++` operators.
- A statement that modifies an array element and then advances to the next element:

```
a[i++] = j;
```

- The corresponding pointer version:

```
*p++ = j;
```

- Because the postfix version of `++` takes precedence over `*`, the compiler sees this as

```
*(p++) = j;
```

- Not

```
(*p)++ = j;
```

# Combining the * and ++ Operators

- Possible combinations of `*` and `++`:

| *Expression* | *Meaning* |
|---|---|
| `*p++` or `*(p++)` | Value of expression is `*p` before increment; increment `p` later |
| `(*p)++` | Value of expression is `*p` before increment; increment `*p` later |
| `*++p` or `*(++p)` | Increment `p` first; value of expression is `*p` after increment |
| `++*p` or `++(*p)` | Increment `*p` first; value of expression is `*p` after increment |

**When in doubt, use parentheses!**

# Combining the * and ++ Operators

- The most common combination of `*` and `++` is `*p++`, which is handy in loops.
- Instead of writing

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

- to sum the elements of the array `a`, we could write

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

# Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related.
- Another key relationship:

***The name of an array can be used as a pointer to the first element***

- This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

# Using an Array Name as a Pointer

- Suppose that `a` is declared as follows:

```
int a[10];
```

- Examples of using `a` as a pointer:

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

- In general, `a + i` is the same as `&a[i]`.
  - Both represent a pointer to element `i` of `a`.
- Also, `*(a+i)` is equivalent to `a[i]`.
  - Both represent element `i` itself.

# Using an Array Name as a Pointer

- The fact that an array name can serve as a pointer makes it easier to write loops that step through an array.

- Original loop:

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

- Simplified version:

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

# Using an Array Name as a Pointer

- Although an array name can be used as a pointer, it's not possible to assign it a new value.
- Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;           /*** WRONG ***/
```

- This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```

# Typical Array Loops - Using Pointers

Examples of typical operations on an array `a` of length `N`:

- Clear the array

```
for (i = 0; i < N; i++)
  a[i] = 0;
```

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

- Read data into an array

```
for (i = 0; i < N; i++)
  scanf("%d", &a[i]);
```

```
for (p=a; p<a+N; p++)
  scanf("%d", p);
```

- Sum the elements of an array

```
for (i = 0; i < N; i++)
  sum += a[i];
```

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

# Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages

# Resources

- These notes
- *C Programming: A modern Approach* by K.N. King, 2008
- Chapters 8, 11 and 12