# Arrays in *C*

# Definition – *Array*

- A collection of objects of the *same type* stored contiguously in memory under one name
  - May be type of any kind of variable
  - May even be collection of arrays!

- For ease of access to any member of array
- For passing to functions as a group

# Examples

- **`int A[10]`**
  - An array of ten integers
  - **`A[0]`**, **`A[1]`**, …, **`A[9]`**
- **`double B[20]`**
  - An array of twenty long floating point numbers
  - **`B[0]`**, **`B[1]`**, …, **`B[19]`**
- Arrays of **`structs`**, **`unions`**, **`pointers`**, etc., are also allowed
- Array indexes *always* start at zero in *C*

# Examples (continued)

- **`int C[]`**
  - An array of an unknown number of integers (allowable in a parameter of a function)
  - `C[0]`, `C[1]`, …, `C[`*max-1*`]`
- **`int D[10][20]`**
  - An array of ten rows, each of which is an array of twenty integers
  - `D[0][0]`, `D[0][1]`, …, `D[1][0]`, `D[1][1]`, …, `D[9][19]`
  - Not used so often as arrays of pointers

# Array Element

- May be used wherever a variable of the same type may be used
  - In an expression (including arguments)
  - On left side of assignment

- Examples:–

```
A[3] = x + y;
x = y - A[3];
z = sin(A[i]) + cos(B[j]);
```

# Array Elements (continued)

- Generic form:–
  - *ArrayName*[*integer-expression*]
  - *ArrayName*[*integer-expression*] [*integer-expression*]
  - Same type as the underlying type of the array
- Definition:– *Array Index* – the expression between the square brackets

# Array Elements (continued)

- Array elements are commonly used in loops
- E.g.,

```
for(i=0; i < max; i++)
 A[i] = i*i;

sum = 0; for(j=0; j < max; j++)
 sum += B[j];

for (count=0;rc!=EOF;count++)
 rc=scanf("%f", &A[count]);
```

# Remember while using an Array

- It is the programmer's responsibility to avoid indexing off the end of an array
  - *Likely* to corrupt data
  - May cause a *segmentation fault*
  - Could expose system to a *security hole!*
- *C* does NOT check *array bounds*
  - I.e., whether index points to an element within the array
  - Might be high (beyond the end) or negative (before the array starts)

# Declaring Arrays

- Static or automatic
- Array size determined explicitly or implicitly
- Array size may be determined at run-time
    - Automatic only
    - Not in textbook

# Declaring Arrays (continued)

- Outside of any function – always static

```
int A[13];

#define CLASS_SIZE 73
double B[CLASS_SIZE];

const int nElements = 25
float C[nElements];

static char[256]; /*not visible to
  linker */
```

# Declaring Arrays (continued)

- Outside of any function – always static

```
int A[13];

#define CLASS_SIZE 73
double B[CLASS_SIZE];

const int nElements = 25
float C[nElements];

static char D[256];    /*not visible to
  linker */
```

*Static* ⇒ retains values across function calls

# Declaring Arrays (continued)

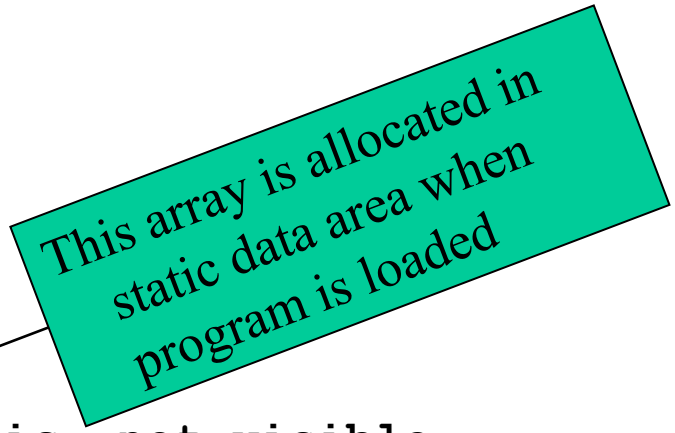
- Inside function or compound statement – usually automatic

```
void f( …) {
    int A[13];

    #define CLASS_SIZE 73
    double B[CLASS_SIZE];

    const int nElements = 25
    float C[nElements];

    static char D[256]; /*static, not visible
      outside function */

} //f
```

# Declaring Arrays (continued)

- Inside function or compound statement – usually automatic

```
void f( …) {
    int A[13];

    #define CLASS_SIZE 73
    double B[CLASS_SIZE];

    const int nElements = 25
    float C[nElements];

    static char D[256]; /*static, not visible
      outside function */

} //f
```



# Dynamic Array Size Determination

- **`gcc`** supports the following:–

```
void func(<other parameters>, const int n) {
  double Arr[2*n];


} //func
```

- I.e., array size is determined by evaluating an expression at run-time
  - Automatic allocation on *The Stack*
  - Not in C88 ANSI standard, not in Kernighan & Ritchie
  - Part of C99

# Array Initialization

- `int A[5] = {2, 4, 8, 16, 32};`
  - Static or automatic
- `int B[20] = {2, 4, 8, 16, 32};`
  - Unspecified elements are guaranteed to be zero
- `int C[4] = {2, 4, 8, 16, 32};`
  - Error — compiler detects too many initial values
- `int D[5] = {2*n, 4*n, 8*n, 16*n, 32*n};`
  - Automatically only; array initialized to expressions
- `int E[n] = {1};`
  - **`gcc`**, C99, C++
  - Dynamically allocated array (automatic only). Zeroth element initialized to *1*; all other elements initialized to *0*

# Implicit Array Size Determination

- `int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`
  - Array is created with as many elements as initial values
    - In this case, 12 elements
  - Values must be compile-time constants (for static arrays)
  - Values may be run-time expressions (for automatic arrays)

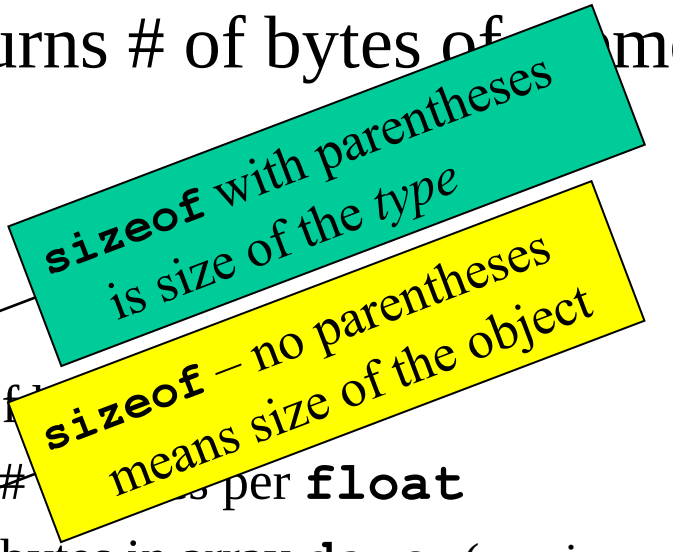
# Getting Size of Implicit Array

- **`sizeof`** operator – returns # of bytes of memory required by operand

- Examples:–
  - **`sizeof (int)`** – # of bytes per **`int`**
  - **`sizeof (float)`** – # of bytes per **`float`**
  - **`sizeof days`** – # of bytes in array **`days`** (previous slide)
  - # of elements in **`days`** = **`(sizeof days)/sizeof(int)`**
- Must be able to be determined at compile time
  - Dynamically allocated arrays not supported

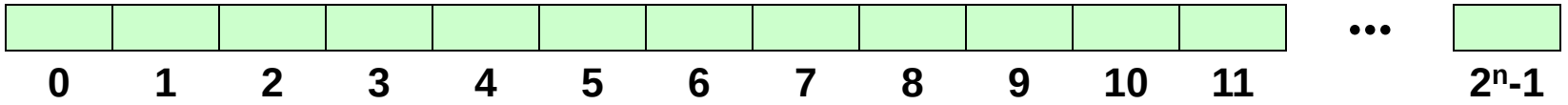
# Getting Size of Implicit Array

- **`sizeof`** operator – returns # of bytes of memory required by operand



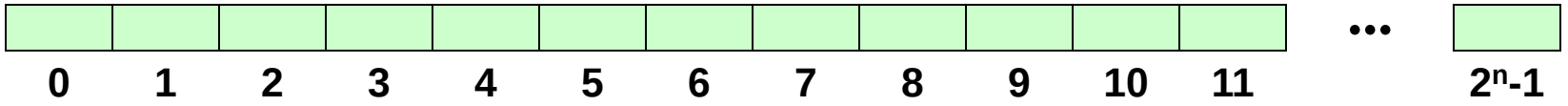

- Examples:–
  - **`sizeof (int)`** – # of
  - **`sizeof (float)`** – # per **`float`**
  - **`sizeof days`** – # of bytes in array **`days`** (previous slide)
  - # of elements in **`days`** = **`(sizeof days)/sizeof(int)`**
- Must be able to be determined at compile time
  - Dynamically allocated arrays not supported

# Digression – Memory Organization




- All modern processors have memories organized as sequence of *numbered bytes*
  - Many (but not all) are linear sequences
  - Notable exception – *Pentium!*
- Definitions:–
  - *Byte:* an 8-bit memory cell capable of storing a value in range 0 … 255
  - *Address:* number by which a memory cell is identified

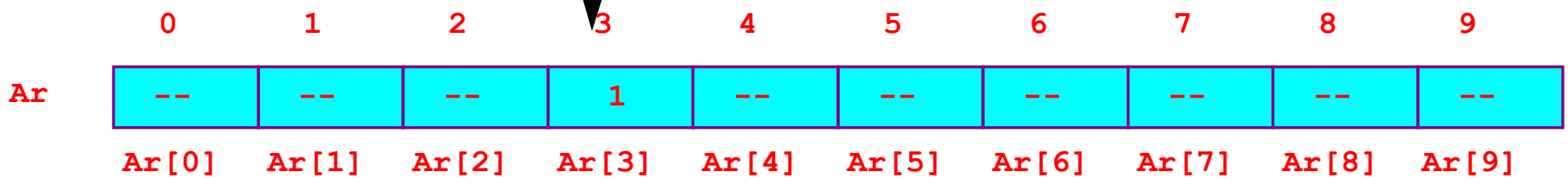
# Memory Organization (continued)




- Larger data types are sequences of bytes – e.g.,
  - **`short int`** – 2 bytes
  - **`int`** – 2 or 4 bytes
  - **`long`** – 4 or 8 bytes
  - **`float`** – 4 bytes
  - **`double`** – 8 bytes
- (Almost) always aligned to multiple of size in bytes
- Address is "first" byte of sequence (i.e., byte zero)
  - May be low-order or high-order byte
  - *Big endian* or *Little endian*

# Subscripting

```
// array of 10 uninitialized ints
int Ar[10];

Ar[3] = 1;
int x = Ar[3];
```



# Change Value of Array elements

```
int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1
mark[2] = -1;

// make the value of the fifth element to 0
mark[4] = 0;
```

# Input and Output Array Elements

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);
// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

```
// print the first element of the array
printf("%d", mark[0]);
// print the third element of the array
printf("%d", mark[2]);
// print ith element of the array
printf("%d", mark[i-1]);
```

# Example 1: Array Input/Output

```
#include <stdio.h>
int main()
{
  int values[5];
  printf("Enter 5 integers: ");
  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i)
      {
       scanf("%d", &values[i]);
      }
  printf("Displaying integers: ");
   // printing elements of an array
  for(int i = 0; i < 5; ++i)
      {
       printf("%d\n", values[i]);
       }
 return 0;
}
```

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

# Example 2: Calculate Average

```
#include <stdio.h>
int main()
{
    int marks[10], i, n, sum = 0, average;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    for(i=0; i<n; ++i)
    {
        printf("Enter number%d: ",i+1);
        scanf("%d", &marks[i]);

        // adding integers entered by the user to the sum variable
        sum += marks[i];
    }

    average = sum/n;
    printf("Average = %d", average);

    return 0;
}
```

```
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39
```

# Multidimensional Arrays

float x[3][4];

| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| Row 1 | x[0][0] | x[0][1] | x[0][2] | x[0][3] |
| Row 2 | x[1][0] | x[1][1] | x[1][2] | x[1][3] |
| Row 3 | x[2][0] | x[2][1] | x[2][2] | x[2][3] |

float y[2][4][3];

| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| Row 1 | x[0][0] | x[0][1] | x[0][2] | x[0][3] |
| Row 2 | x[1][0] | x[0][0] | x[0][1] | x[0][2] | x[0][3] |
| Row 3 | x[2][0] | x[1][0] | x[1][1] | x[1][2] | x[1][3] |
| | | x[2][0] | x[2][1] | x[2][2] | x[2][3] |

# Initializing a multidimensional array

Initialization of a 2d array

// Different ways to initialize two-dimensional array

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Initialization of a 3d array

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

# Example: Sum of two matrices

```
#include <stdio.h>
int main()
{
  float a[2][2], b[2][2], result[2][2];

  // Taking input using nested for loop
  printf("Enter elements of 1st matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter a%d%d: ", i + 1, j + 1);
      scanf("%f", &a[i][j]);
    }

  // Taking input using nested for loop
  printf("Enter elements of 2nd matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter b%d%d: ", i + 1, j + 1);
      scanf("%f", &b[i][j]);
    }

  // adding corresponding elements of
two arrays
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      result[i][j] = a[i][j] + b[i][j];
    }

  // Displaying the sum
  printf("\nSum Of Matrix:");

  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("%.1f\t", result[i][j]);

      if (j == 1)
        printf("\n");
    }
  return 0;
}
```

# Example: Sum of two matrices

Enter elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

Sum Of Matrix:
2.2  0.5
-0.9  25.0

# Example: Three-dimensional array

```
#include <stdio.h>
int main()
{
  int test[2][3][2];

  printf("Enter 12 values: \n");

  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        scanf("%d", &test[i][j][k]);
      }
    }
  }
```

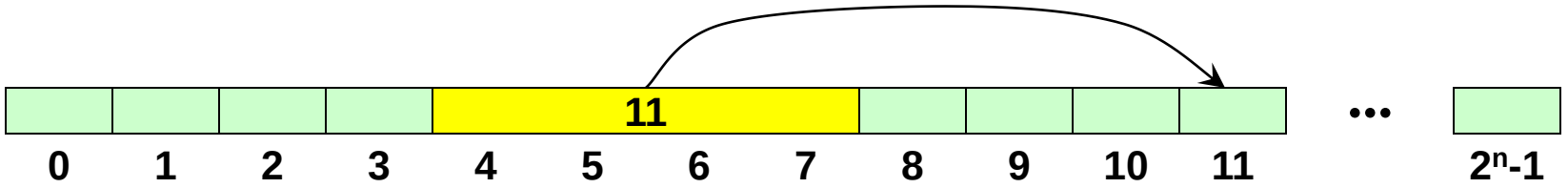
```
  // Printing values with proper index.

  printf("\nDisplaying values:\n");
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        printf("test[%d][%d][%d] = %d\n", i, j, k,
                                              test[i][j][k]);
      }
    }
  }

  return 0;
}
```

# Example: Three-dimensional array

Enter 12 values:
1
2
3
4
5
6
7
8
9
10
11
12

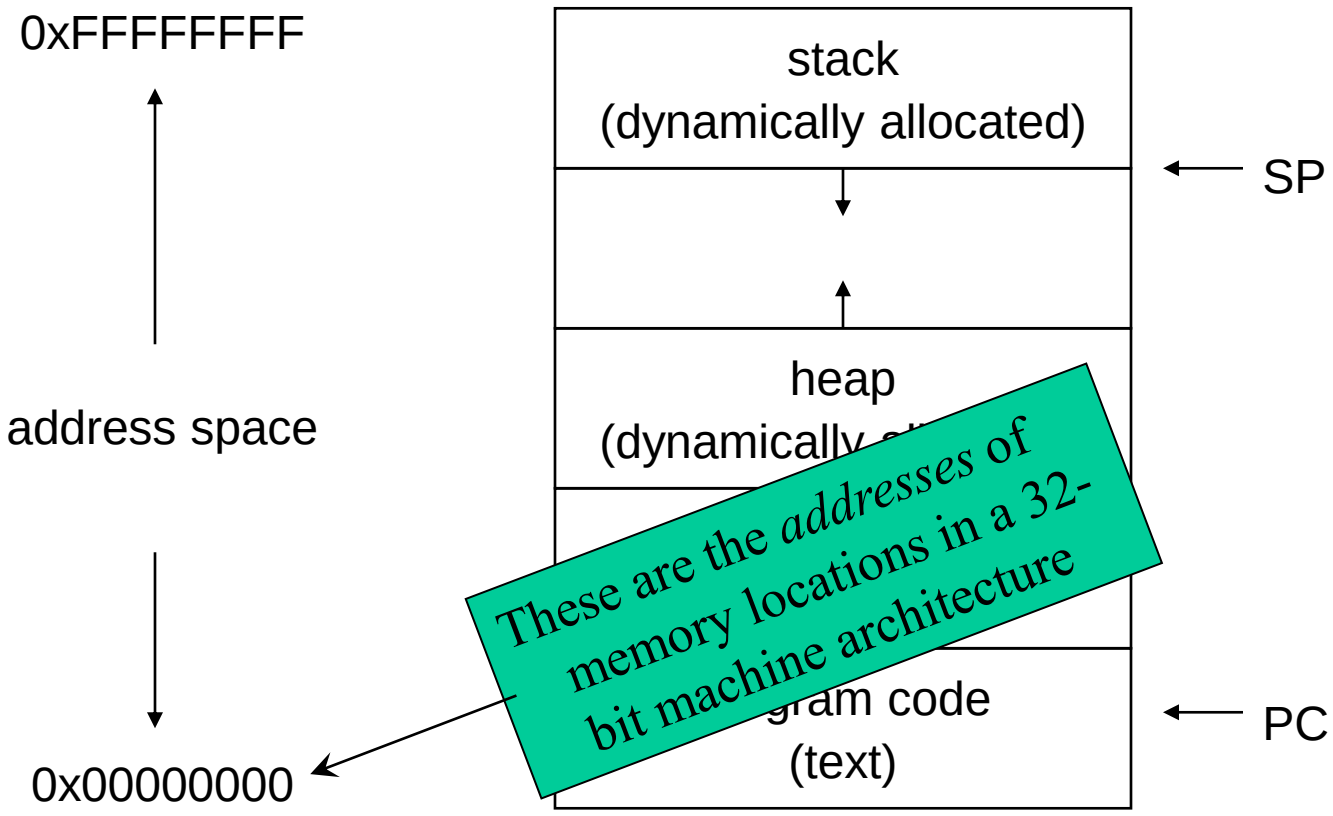
Displaying Values:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12

# Definition – *Pointer*




- A *value* indicating the *number* of (the first byte of) a data object
  - Also called an *Address* or a *Location*
- Used in machine language to identify which data to access
    - E.g., *stack pointer* is address of most recent entry of *The Stack*
- Usually 2, 4, or 8 bytes, depending upon machine architecture

# Memory Addressing



# Pointers in *C*

- Used *everywhere*
  - For building us[illegible], data structures
  - For return[illegible] functions
  - For manag[illegible] arrays
- **`'&'`** unary operator generates a *pointer* to **`x`**
  - E.g., **`scanf("%d", &x);`**
  - E.g., **`p = &c;`**
  - Operand of **`'&'`** must be an *l-value* — *i.e.*, a legal object on left of assignment operator (**`'='`**)
- Unary **`'*'`** operator *dereferences* a pointer
  - i.e., gets value pointed to
  - E.g. **`*p`** refers to value of **`c`** (above)
  - E.g., **`*p = x + y; *p = *q;`**

Not the same as binary **`'&'`** operator (bitwise AND)

# Declaring Pointers in *C*

- **`int *p;`** — a pointer to an **`int`**
- **`double *q;`** — a pointer to a **`double`**
- **`char **r;`** — a pointer to a pointer to a **`char`**
- *type* **`*s;`** — a pointer to an object of type *type*
  - E.g, a **`struct`**, **`union`**, **`function`**, something defined by a **`typedef`**, etc.

# Declaring Pointers in *C* (continued)

- Pointer declarations:–read from *right* to *left*
- **`const int *p;`**
  - **`p`** is a pointer to an integer constant
  - I.e., pointer can change, thing it points to cannot
- **`int * const q;`**
  - **`q`** is a constant pointer to an integer variable
  - I.e., pointer cannot change, thing it points to can!
- **`const int * const r;`**
  - **`r`** is a constant pointer to an integer constant

# Pointer Arithmetic

- **`int *p, *q;`**
  **`q = p + 1;`**
  - Construct a pointer to the next *integer* after **`*p`** and assign it to **`q`**
- **`double *p, *r;`**
  **`int n;`**
  **`r = p + n;`**
  - Construct a pointer to a *double* that is **`n`** *doubles* beyond **`*p`**, and assign it to **`r`**
  - **`n`** may be negative

# Pointer Arithmetic (continued)

- **`long int *p, *q;`**
  **`p++; q--;`**
  - Increment **`p`** to point to the next **`long int`**; decrement **`q`** to point to the previous **`long int`**
- **`float *p, *q;`**
  **`int n;`**
  **`n = p - q;`**
  - **`n`** is the number of floats between **`*p`** and **`*q`**; i.e., what would be added to **`q`** to get **`p`**

# Pointer Arithmetic (continued)

C never checks that the resulting pointer is valid

- **`long int *p, *q;`**
  **`p++; q--;`**
  - Increment **`p`** to point to the next **`long int`**; decrement **`q`** to point to the previous **`long int`**
- **`float *p, *q;`**
  **`int n;`**
  **`n = p - q;`**
  - **`n`** is the number of floats between **`*p`** and **`*q`**; i.e., what would be added to **`q`** to get **`p`**

# Why introduce pointers in the middle of a lesson on arrays?

- Arrays and pointers are *closely related* in *C*
  - In fact, they are essentially the same thing!
  - Esp. when used as parameters of functions
- **`int A[10];`**
  **`int *p;`**
  - *Type* of **`A`** is **`int *`**
  - **`p = A;`** and **`A = p;`** are legal assignments
  - **`*p`** refers to **`A[0]`**
    **`*(p + n)`** refers to **`A[n]`**
  - **`p = &A[5];`** is the same as **`p = A + 5;`**

# Arrays and Pointers (continued)

- **`double A[10];`** *vs.* **`double *A;`**
- *Only* difference:–
  - **`double A[10]`** sets aside *ten* units of memory, each large enough to hold a **`double`**
  - **`double *A`** sets aside *one* pointer-sized unit of memory
    - You are expected to come up with the memory elsewhere!
  - Note:– all pointer variables are the same size in any given machine architecture
    - Regardless of what types they point to

# Note

- *C* does *not* assign arrays to each other
- *E.g*,
  - 
```
double A[10];
double B[10];

A = B;
```
    - assigns the pointer value **B** to the pointer value **A**
    - Contents of array **A** are untouched

# Arrays as Function Parameters

- **`void init(float A[], int arraySize);`**
  **`void init(float *A, int arraySize);`**

- Are identical function prototypes!
- Pointer is passed by value
- I.e. caller copies the *value* of a pointer to **`float`** into the parameter **`A`**
- Called function can reference *through* that pointer to reach thing pointed to

# Arrays as Function Parameters (continued)

```
• void init(float A[], int arraySize){
      int n;

      for(n = 0; n < arraySize; n++)
           A[n] = (float)n;

  }  //init
```

- Assigns values to the array A *in place*
  - So that caller can see the changes!

# Examples

```
while ((rc = scanf("%lf", &array[count]))
  !=EOF && rc==0)
      ...

double getLargest(const double A[], const
           int sizeA) {
  double d;
  if (sizeA > 0) {
     d = getLargest(&A[1], sizeA-1);
     return (d > A[0]) ? d : A[0];
  } else
     return A[0];
}    // getLargest
```

# Result

- Even though all arguments are passed *by value* to functions …
- … pointers allow functions to assign back to data of caller

- Arrays are pointers passed by value

# Safety Note

- When passing arrays to functions, *always* specify **`const`** if you don't want function changing the value of any elements

- Reason:– you don't know whether your function would pass array to another before returning to you
    - Exception – many software packages don't specify **`const`** in their own headers, so you can't either!

# Reading Assignment

Chapter 5 of Kernighan & Ritchie

# Questions?