Arrays in C & C++
(including a brief introduction to pointers)

Modified from WPI CS-2303

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)

THE UNIVERSITY OF KANSAS

Common I/O funcitons

`printf()` function - show output

`scanf()` function - take input

```
Example: scanf("%d", &n);
```

`getchar()` and `putchar()`

`gets()` and `puts()` function

Formatting output

Description	Code	Result
At least five wide	printf("'%5d'", 10);	' 10'
At least five-wide, left-justified	printf("'%-5d'", 10);	'10 '
At least five-wide, zero-filled	printf("'%05d'", 10);	'00010'
At least five-wide, with a plus sign	printf("'%+5d'", 10);	' +10'
Five-wide, plus sign, left-justified	printf("'%-+5d'", 10);	'+10 '

Description	Code	Result
Print one position after the decimal	printf("'%.1f'", 10.3456);	'10.3'
Two positions after the decimal	printf("'%.2f'", 10.3456);	'10.35'
Eight-wide, two positions after the decimal	printf("'%8.2f'", 10.3456);	' 10.35'
Eight-wide, four positions after the decimal	printf("'%8.4f'", 10.3456);	' 10.3456'
Eight-wide, two positions after the decimal, zero-filled	printf("'%08.2f'", 10.3456);	'00010.35'
Eight-wide, two positions after the decimal, left-justified	printf("'%-8.2f'", 10.3456);	'10.35 '

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

Arrays in C

- By far, the dominant kind of data structure in C programs
- Many, many uses of all kinds
    * Collections of all kinds of data
    * Instant access to any element

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]`**

Two-dimensional Arrays

- `int D[10][20]`
  - A *one-dimensional array* with 10 elements, each of which is an array with 20 elements
- `int D[10][20]     /*[row][col]*/`
- Last subscript varies the fastest
  - I.e., elements of last subscript are stored contiguously in memory
- Also, three or more dimensions

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
  * Also called an *array subscript*

Array Elements (continued)

- Array elements are commonly used in loops
- Example

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

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

Caution! Caution! Caution!

- 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 (continued)

- Outside of any function – always static

```
int A[13];

#define CLASS_SIZE 73
#define MAX 150

double B[CLASS_SIZE];

const int nElements = 25
float C[nElements];
```

Declaring Arrays (continued)

- Outside of any function – always static

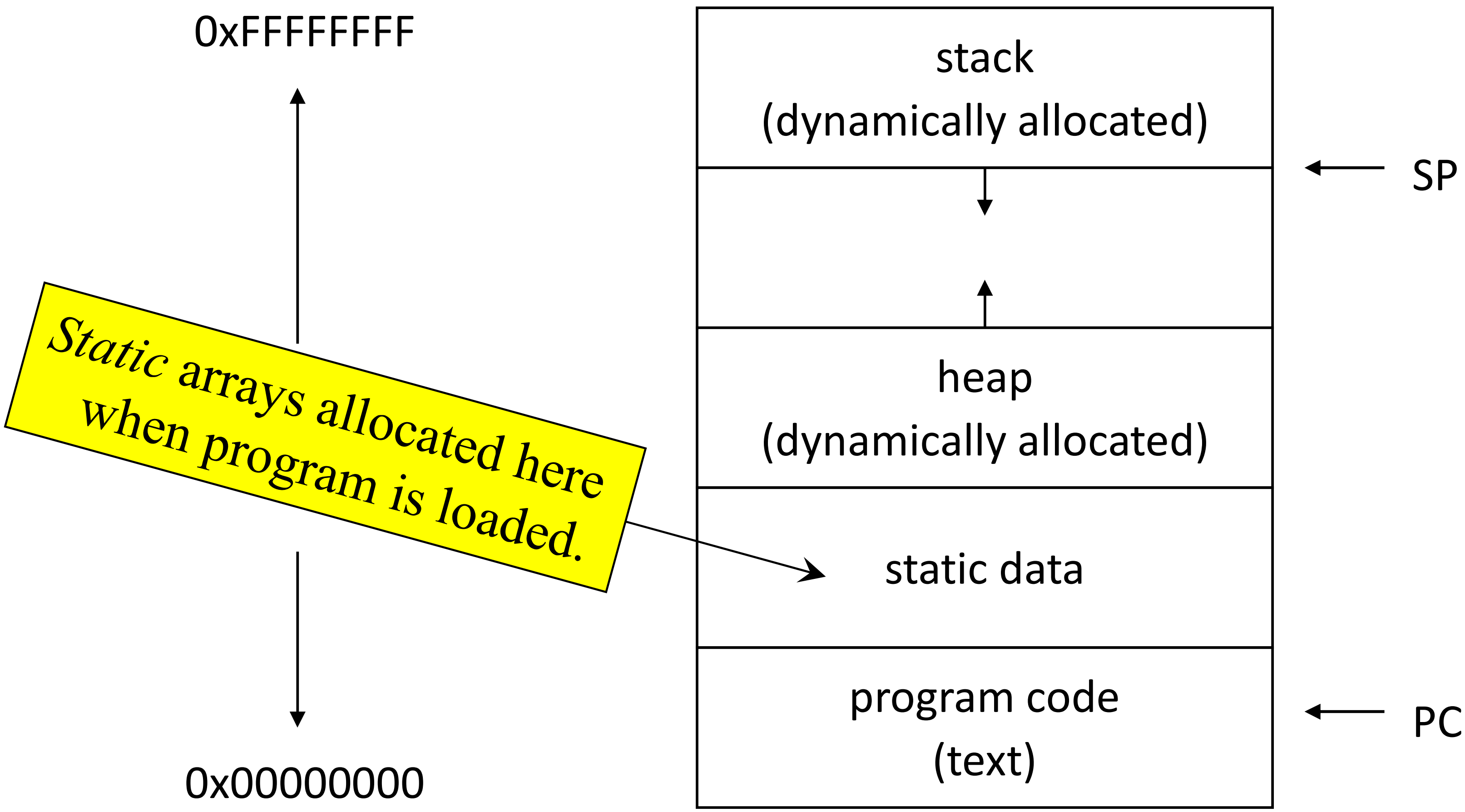
```
int A[13];

#define CLASS_SIZE 73
double B[CLASS_SIZE];

const int nElements = 25
float C[nElements];
```

Static ⇒ retains values across function calls

Static Data Allocation



Declaring Arrays (continued)

- Inside function or compound statement – usually automatic

```
#define CLASS_SIZE 100

void f( …) {
  int A[13];

  double B[CLASS_SIZE];

  const int nElements = 25
  float C[nElements];


} //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* and C++

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 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)
  - See p. 86 of K&R

Getting Size of Implicit Array

- sizeof operator – returns # of bytes of memory required by operand
  - See p.135 of K&R, §7.7 of D&D
- Examples:–
  - sizeof (int) – # of bytes per int
  - sizeof (float) – # of bytes per float
  - sizeof days – # of bytes in array days (previous slide)
- Must be able to be determined at compile time
  - Getting size of dynamically allocated arrays not supported

Getting Size of Implicit Array

- sizeof operator – returns # of bytes of memory required by operand
  - See p.135
- 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
  - Getting size of dynamically allocated arrays not supported

`sizeof` with parentheses is size of the *type*

`sizeof` – no parentheses means size of the object

Initializing a Two-Dimensional Array

```
static char daytab[2][12] = {
  {31,28,31,30,31,30,31,31,30,31,30,31},
  {31,29,31,30,31,30,31,31,30,31,30,31}
};   //   daytab
```

OR

```
static char daytab[2][12] = {
  31,28,31,30,31,30,31,31,30,31,30,31,
  31,29,31,30,31,30,31,31,30,31,30,31
};   //   daytab
```

Pointers in C

- Used *everywhere*
  - For building useful, interesting data structures
  - For returning data from functions
  - For managing 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*

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`, and `A` is initialized to point to the zeroth unit.
  - `double *A` sets aside *one* pointer-sized unit of memory, not initialized
    - 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**
    * Original contents of array **A** are untouched (and possibly unreachable!)
    ```

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, *it is recommended to* 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!