CSC / CPE 357

Systems Programming

Topics

- Arrays

Reference: Chapter 5 in The C Programming Language

These slides are based on materials from Professors Aaron Keen and Andrew Migler

Arrays in C

- `type name[size]` allocates `size x sizeof(type)` bytes of contiguous memory
    - `int months[12];`
- By default, array values are “mystery” data (i.e., uninitialized)
- Size of an array is not tracked automatically
    - An array does not know its own size!
- Recent versions of C allow for variable-length arrays (VLA)

    -
      ```
      int n = 12;
      int months[n];
      ```
    - We will avoid the use of VLAs (for now)

Array Initialization

- `type name[size] = { val0, val1, …, valN };`
    - Initialization via { } can be used only at time of definition
    - If no `size` supplied, array size is inferred from length of array initializer

- Example:

```
char vowels[] = { 'a', 'e', 'i', 'o', 'u' };
```

- `name[index]` specifies an element of the array, beginning with 0
    - `vowels[0]; // 'a'`

Memory Organization

- Simplified model: array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups
    - single byte: `char`
    - two contiguous bytes: `short`
    - four contiguous bytes: `int`

- A **pointer** is a group of cells that can hold a memory address

Pointers in C

Syntax:

```
int *p;  // variable p can contain the address of an int
```

Caution:

`int *p1, p2;` is not the same as `int *p1, *p2;`

Pointer Operators

- The unary operator & gives the address of an object:
  - `&var` represents the address of a variable named `var`
  - Example:
    ```
    int i = 357;
    int *p = &i;
    ```

- The unary operator * (asterisk) is the indirection or dereferencing operator:
  - Access the memory value referred to by a pointer
  - Example:
    ```
    printf("This is %d\n", *p);
    ```

Pointer Operators

- If `ip` points to the integer `x`, then `*ip` can occur in any context where `x` could:
  - `*ip = *ip + 10; // increments *ip (x) by 10.`

- Combining unary operators (* and &) and arithmetic operators
  - `y = *ip + 1; // add 1 to value referenced by ip, assign the result to y`
  - Increment the value that `ip` references:
    - `*ip += 1`
    - `++*ip`
    - `(*ip)++  // parentheses are necessary here!`

- `(*ip)++` versus `*ip++`
  - The latter increments the pointer `ip` (unary operators associate right to left)

`Pointer Arithmetic`

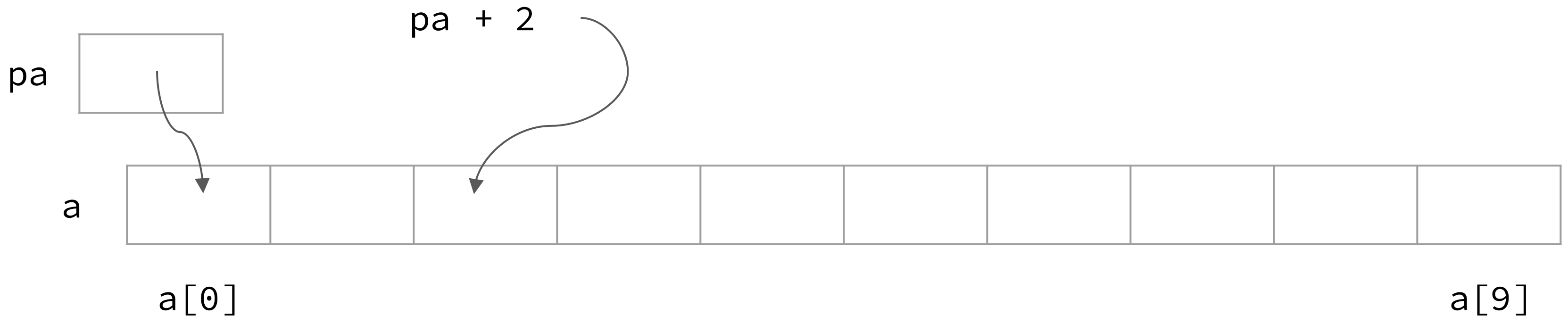
- Pointers are declared with a type
  - Compiler is aware of the size of the data you are pointing to
  - Exception: `void *` is a generic pointer (i.e., a placeholder)

- Pointer arithmetic is scaled by `sizeof(*p)`

- Closely aligned with arrays

Pointers and Arrays

```
int a[10];
int *pa;
pa = &a[0];  // equivalent to: pa = a;
```



Pointers and Arrays

- A pointer can point to an array element
  - Array indexing notation on pointers
  - Pointer arithmetic may be used to access array elements
  - `ptr[i]` is equivlanet to `*(ptr+i)`

- Array name refers to the beginning address of the array
  - Pointer to the first element of array. Unlike pointers, this can’t be changed (array name always points to first element)
    - Valid: `pa = a;` and `pa++;`
    - Invalid: `a = pa;` and `a++;`

Pointers and Arrays: Example

```
int a[] = { 10, 20, 30, 40, 50 };
int *p1 = &a[3]; // 4th element in array a
int *p2 = &a[0]; // 1st element in array a
int *p3 = a;     // 1st element in array a
*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;
// what values does array a contain after these steps?

// a now contains: 200, 400, 500, 100, 300
```

Character Pointers (Strings)

- A string constant, for example `"Hello"` is an array of characters.

- Internal representation of a string: character array terminated with the null character `'\0'`
    - Allows programs to find the end
    - The length in memory is thus one more than the visible characters

- A string is accessed by a pointer to its first character
    - C has no operators for processing an entire string of characters as a unit

`char * versus char[]`

```
char *s = "abc";
char t[] = "abc";
```

- `char *` results in an <u>unmodifiable</u> string constant in a memory area that should be considered read-only
  - `s[0] = 'A'; // invalid`
  - It is possible to change the *pointer* s to point elsewhere (for example: `s = t;`)
- `char t[]` allocates a modifiable array of characters
  - `t[0] = 'A'; // valid`
  - t is an array name -- <u>not a pointer variable</u> in the sense that s is
  - `t = s; // invalid`

Copying a String

```
char *s = "one";
char *t = "two";
```

- To copy `t` to `s`, it is <u>not</u> sufficient to use: `s = t`
    - `s = t` copies the *pointer* only, not the string content
- Instead, a loop is required

Pointers as Function Arguments

Consider a swap() function to support a sorting algorithm, used to exchange to out-of-order values in an array

```
swap(a,b);

// recall: call by value
void swap(int x, int y) {
    int temp;
        temp = x;
        x = y;
        y = temp;
}
```

```
swap(&a, &b);

// pointer arguments
void swap(int *px, int *py) {
    int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
```

Pointers to Pointers

- Pointers are variables themselves
- Can be stored and manipulated just like other variables

Pointers to Pointers

```
int a = 357;
int b = 453;
int *ip = &a;
int *ip2 = &b;
int **ipp = &ip;

ipp = &ip2;
```

Pointers Example 2

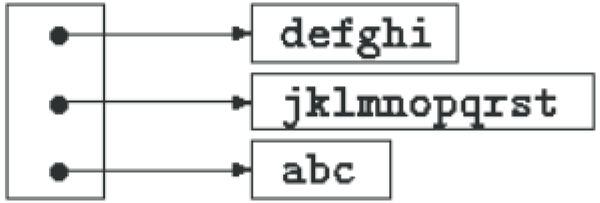
```
int arr[3] = { 2, 3, 4 };
int *p = &arr[1];
int **dp = &p; // pointer to a pointer

*(*dp) += 1;
p += 1;
*(*dp) += 1;

// What values does arr hold at this point?
// Poll:
// A. {2, 3, 4}
// B. {3, 4, 5}
// C. {2, 6, 4}
// D. {2, 4, 5}
// E. Not sure
```

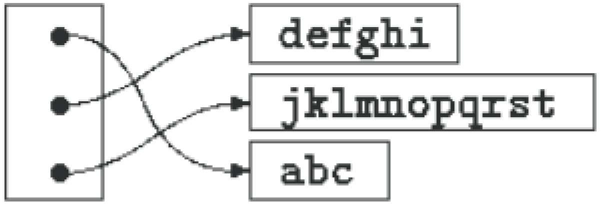
Pointers to Pointers

- Example: sorting strings

- Array of character pointers:

```
char *strings[];
```





Two-dimensional Arrays

- A two-dimensional array is a one-dimensional array, each of whose elements is an array.

- To define an array of integers with 3 rows and 4 columns:

```
int tda[3][4];
```

- To initialize the array values:

```
int tda[3][4] = { {1,2,3,4}, {2,3,4,5}, {4,5,6,7} };
```

Two-dimensional Array Subscripts

Subscripts are written as

```
tda[i][j]  /* [row][col] */
```

rather than

```
tda[i,j]   /* invalid */
```

Two-dimensional Array Indexing

```
int tda[3][4];
```

tda[0][0]	tda[0][1]	tda[0][2]	tda[0][3]
tda[1][0]	tda[1][1]	tda[1][2]	tda[1][3]
tda[2][0]	tda[2][1]	tda[2][2]	tda[2][3]

Multi-Dimensional Arrays as Parameters

- When passed as a function parameter, the parameter declaration must include, at a minimum, the number of columns

- Number of rows is not necessary

- Three possibilities:
  - `void fun(int tda[3][4]) { ... }`
  - `void fun(int tda[][4]) { ... }`
  - `void fun(int (*tda)[4]) { ... }`

> 2 Dimensional Arrays

- 2-D arrays are commonly used
- C supports arrays with any number of dimensions
- For example, a 3-dimensional array:
  - `int arr[5][2][3];  // essentially, 5 2x3 arrays`
- We will defer conversion about > 2-D arrays