Arrays and Strings

CS449 Fall 2015
Arrays

- Data type for a sequence of variables of a given type in consecutive memory
- Just as for pointers, there are array types for each primitive data type (e.g. char, int, float)
Running Example

```c
#include <stdio.h>

int main()
{
    int nums[5], i;    /* declarations */
    printf("Enter nums: ");
    for(i=0; i<5; ++i) {
        scanf("%d", &nums[i]);    /* write to array */
    }
    printf("Your nums: ");
    for(i=0; i<5; ++i) {
        printf("%d ", nums[i]);    /* read from array */
    }
    return 0;
}
```

>> ./a.out
Enter nums: 10 20 30 40 50
Your nums: 10 20 30 40 50
Declaring Arrays

```c
#include <stdio.h>

int main()
{
    int nums[5], i; /* declarations */
    printf("Enter nums: ");
    for(i=0; i<5; ++i) {
        scanf("%d", &nums[i]); /* write to array */
    }
    printf("Your nums: ");
    for(i=0; i<5; ++i) {
        printf("%d ", nums[i]); /* read from array */
    }
    return 0;
}
```

• Syntax: `<type> <name> [ <length> ]`
  – E.g. “int num[5];”, “char c[10];”

• Combination of type and length tells the compiler amount of memory to reserve (sizeof(type) * length)
  – Length must always be declared (explicitly or implicitly)

• Array initializers
  – Handy way to initialize elements of array
  – E.g. “int num[3] = {1, 2, 3};”
  – If initializer shorter than length, rest initialized to 0 (e.g. “int num[3] = {1};”, “int num[3] = {};”)
  – Initializers implicitly gives size of array (e.g. “int num[] = {1, 2, 3};”
  – Initialized arrays can still be modified
#include <stdio.h>

int main()
{
    int nums[5], i; /* declarations */
    printf("Enter nums: ");
    for(i=0; i<5; ++i) {
        scanf("%d", &nums[i]); /* write to array */
    }
    printf("Your nums: ");
    for(i=0; i<5; ++i) {
        printf("%d ", nums[i]); /* read from array */
    }
    return 0;
}

- Syntax: `<name> [ <index> ]`
  - E.g. “nums[5]”, “c[10]”
- Index starts with 0
  - “nums[i]” accesses the “i+1 th” element
- Value of array name with no index (e.g. “nums”) is the address of first element of array (equivalent to “&nums[0]”)
  - Is interchangeable with pointer
  - “int *p = nums;” is perfectly valid
  - “*p == nums[0]”
- Difference between array and pointer
  - Array has allocated memory statically bound to the name at compile time
  - “nums” not part of data of program
  - “nums” cannot point to new address (e.g. “nums = p” results in compile error)
Declaring Multidimensional Arrays

- Syntax: `<type> <name> [ <length1> ] [ <length2> ] ... [ <lengthN> ]`
  - E.g. “int nums[2][3];”
- Array initializers
  - E.g. “int nums[2][3] = { {0, 1, 2}, {3, 4, 5} };”
- Conceptual layout

<table>
<thead>
<tr>
<th></th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Row 2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- Physical layout in linear memory

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>[0][0]</td>
<td>[0][1]</td>
<td>[0][2]</td>
<td>[1][0]</td>
<td>[1][1]</td>
<td>[1][2]</td>
<td></td>
</tr>
</tbody>
</table>
Accessing Multidimensional Arrays

• **Syntax:** `<type> <name> [ <index1> ] [ <index2> ] ... [ <indexN> ]`
  
  – E.g. “`nums[1][1];`” accesses offset 4 (1 * 3 + 1) in linear memory

• **Pointer type of “nums”?**
  
  – “`int (*p)[3] = nums;`”
  – “`p[1][1] == 4`”
  – Interpretation: “`p` is a pointer to a 3-column array of integers”
  – Not “`int *p[3]`” => “`p` is a 3-column array of integer pointers”

• **Pointer type of “nums[1]”?**
  
  – “`int *p = nums[1];`”

• **Neither “nums” or “nums[1]” can be target of assignment**
Strings

• There is no “string” data type in C
• Sequence of characters in consecutive memory ending with a null character (‘\0’);
  – Null character: character constant with ASCII value 0
  – In short, a string is a null-terminated character array
• String variables are declared as character arrays
  – e.g. “char s[10];” can hold a string 9 characters long (excluding the null character)
  – Character pointers (“char *”) can point to strings
  – “const char*” points to an immutable string
String Functions

- Defined in C standard library, declared in `<string.h>`

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t strlen(const char *s);</code></td>
<td>Calculates the length of the string <code>s</code>, not including the terminating '\0' character.</td>
</tr>
<tr>
<td><code>int strcmp(const char *s1, const char *s2);</code></td>
<td>Compares the two strings <code>s1</code> and <code>s2</code>. It returns an integer less than, equal to, or greater than zero if <code>s1</code> is found, respectively, to be less than, to match, or be greater than <code>s2</code>.</td>
</tr>
<tr>
<td><code>int strncmp(const char *s1, const char *s2, size_t n);</code></td>
<td>Same as above, except only compares the first (at most) <code>n</code> characters of <code>s1</code> and <code>s2</code>.</td>
</tr>
</tbody>
</table>
# String Functions

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strcpy(char *dest, const char *src);</code></td>
<td>Copies the string pointed to by <code>src</code> (including the terminating \0 character) to the array pointed to by <code>dest</code>.</td>
</tr>
<tr>
<td><code>char *strncpy(char *dest, const char *src, size_t n);</code></td>
<td>Same as above, except that not more than <code>n</code> bytes of <code>src</code> are copied.</td>
</tr>
<tr>
<td><code>char *strcat(char *dest, const char *src);</code></td>
<td>Appends the <code>src</code> string to the <code>dest</code> string overwriting the \0 character at the end of <code>dest</code>, and then adds a terminating \0 character.</td>
</tr>
<tr>
<td><code>char *strncat(char *dest, const char *src, size_t n);</code></td>
<td>Same as above, except that it will use at most <code>n</code> characters from <code>src</code>.</td>
</tr>
</tbody>
</table>

- Always try to use the “n” version to prevent buffer overruns
# String Functions

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *strchr(const char *s, int c);</td>
<td>Returns a pointer to the first occurrence of the character c in the string s.</td>
</tr>
<tr>
<td>char *strstr(const char *haystack, const char *needle);</td>
<td>Finds the first occurrence of the substring needle in the string haystack.</td>
</tr>
</tbody>
</table>
# String Conversion Functions

- String to number conversion functions declared in `<stdlib.h>`

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int atoi(const char *nptr);</code></td>
<td>Converts string pointed to by nptr to int.</td>
</tr>
<tr>
<td><code>double atof(const char *nptr);</code></td>
<td>Converts string pointed to by nptr to double.</td>
</tr>
</tbody>
</table>

- String formatting functions declared in `<stdio.h>`

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int sprintf(char *str, const char *format, ...);</code></td>
<td>Same as printf except instead of writing to stdout formatted string is written to str</td>
</tr>
<tr>
<td><code>int snprintf(char *str, size_t size, const char *format, ...);</code></td>
<td>Same as above, except no more than size bytes are written to str</td>
</tr>
</tbody>
</table>

- Not an exhaustive list. Use your manpages. (e.g. “man string”, “man stdio”)
Pitfall 1: Arrays

• What’s wrong with the following code?

```c
int a[5];
a[5] = 0;
```

• “a[5]” is attempting to get a value from unallocated memory so result is undefined

• Remember index always starts with 0
Pitfall 2: String buffer allocation

• What’s wrong with the following code?
  char *s1;
  strcpy(s1, s2);
  “s1” points to random unallocated memory
• Change pointer to array: “char s1[100];”
• Better yet, change strcpy to strncpy
  char s1[100];
  strncpy(s1, s2, 100); // to prevent buffer overflow
  s1[99] = ‘\0’; // in case s2 is longer than 100 chars
Pitfall 3: String comparison

• What’s wrong with the following code?
    char s[100];
    strcpy(s, “Hello”);
    if(s == “Hello”) printf(“Hello\n”);

• The value of “s” is just the address &s[0]
• Should do instead:
    if(strcmp(s, “Hello”) == 0) printf(“Hello\n”);
Review of Data Types

- **Primitive data types**
  - integers: char, short, int, long, long (signed and unsigned)
  - reals: float, double, long double

- **Pointers (derived type that points to another type)**
  - char *p, int *p, int (*p)[3], int (*p)[2][3], int **p

- **Arrays (derived type that is a sequence of a given type)**
  - char a[3], int a[3], int a[2][3], int *a[3] (same as int *(a[3]))

- **Given array int a[2][3][4], the following are valid**
  - int (*p)[3][4] = a; (same as int (*p)[3][4] = &a[;])
  - int (*p)[4] = a[1]; (same as int (*p)[4] = &a[1][0];)
  - int *p = a[1][0]; (same as int *p = &a[1][0][0])
  - int p = a[1][0][2];
Pitfall 4: No L-Value

• What’s wrong with the following code?
int n;
int **p = &(&n);
• &n has no storage location (a.k.a. l-value)
• The following code is valid:
int n;
int *p = &n;
int **p2 = &(p);
• By the same token, the following is invalid
int a[3], b[3];
a = b; // a is not an l-value