Arrays and Strings

CS449 Fall 2017
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)
```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 Array Variables

```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];”
- **Meaning:** Declares a sequence of `<length>` variables of `<type>` and associates them with `<name>`
- **Length must always be declared**
  - Combination of type and length tells the compiler amount of memory to reserve: `<length> * sizeof(<type>)`
  - E.g. `sizeof(nums) == 20`
    (Assuming `sizeof(int) == 4`)
#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: <array decl> = { <values> }
  - E.g. “int num[3] = {1, 2, 3};”
- Array initializers
  - Can only be used at the point of declaration (and not later)
  - If initializer shorter than length, rest initialized to 0. E.g.
    • int num[3] = {1}; (or {1,0,0})
    • int num[3] = {}; (or {0,0,0})
  - Initializers can implicitly give the size of array. E.g.
    • In int num[] = {1, 2, 3};, length does not need to specified since it’s implicitly 3
Accessing Array Variables

#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;
}
The [] Operator

• More generally, takes the form “<expr> [index]”
  – Where <expr> can be any expression that provides two pieces of information:
    • Starting **address** (of array)
    • **Type** of each element (in array)
  – Semantics of [] operator:
    • <starting addr> + index * sizeof(<type of element>)
  – <expr> can be name of an array, but also:
    • Pointer variable (e.g. In “char *p;”, p can provide a starting address and a char element type)
    • Computation resulting in a pointer value (we’ll see this later)
  – Arrays and pointers can be used interchangeably!
Arrays are Really Pointer Constants

#include <stdio.h>

int main()
{
    int a[3] = {0, 1, 2}, b[3], *p; /* declarations */
    printf("a = 0x%08x\n", a);
    p = a;
    printf("p = 0x%08x\n", p);
    printf("p[2] = 0x%d\n", p[2]);
    /* a = b; */ /* Results in compile error! */
    return 0;
}

• When array name is used in an expression, the value is:
  Pointer to first element in array
  (e.g. a == &a[0])

• Pointer value can be assigned to a pointer variable
  — Allows arrays to be passed around in a program easily and efficiently

• Arrays are “pointer constants”
  — “a” has a value but no storage location so cannot be assigned to
Declaring Multidimensional Arrays

- Syntax: `<type> <name> [ <length1> ] [ <length2> ] ... [ <lengthN> ]`
  - E.g. “int nums[2][3];”
- Conceptual layout

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

- Physical layout in linear memory

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[0][0]</td>
<td>[0][1]</td>
<td>[0][2]</td>
<td>[1][0]</td>
<td>[1][1]</td>
<td>[1][2]</td>
</tr>
</tbody>
</table>

- Array initializers
  - E.g. “int nums[2][3] = { {0, 1, 2}, {3, 4, 5} };”
Accessing Multidimensional Arrays

• Syntax: `<type> <name> [ <index 1> ] [ <index 2> ] ...`
  – E.g. “nums[1][1];” accesses offset 4 (1 * 3 + 1) in linear memory

• Value of “nums”?
  – Pointer to first element in nums array
  – `int (*p)[3] = nums; // same as p = &nums[0];`
  – Each element is an int [3] (a row of 3 integers)

• Value of “nums[0]”?
  – Pointer to first element in nums[0] array
  – `int *p = nums[0]; // same as p = &nums[0][0];`
  – Each element is an int (an integer)

• Value of “nums[0][0]”?
  – Array variable at row 0, column 0
Accessing Multidimensional Arrays

• Are values for “nums” and “nums[0]” identical?
  – Yes, in the sense that they point to the same addresses
  – But, they have different element types! (int [3] vs. int)

• Consider the following two examples:

```c
int nums[2][3];
int (*p)[3] = nums;
p[0] == &nums[0][0];
p[0][0] == nums[0][0];

int nums[2][3];
int *q = nums[0];
q[0] == nums[0][0];
q[0][0] == ??? /* Compile error! */
```

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

**nums**  **nums[0]**
There is no “string” data type in C

Sequence of characters in consecutive memory ending with a null character (‘\0’);

- e.g. ”HELLO”:

| ‘H’ | ‘E’ | ‘L’ | ‘L’ | ‘O’ | ‘\0’ |

- Character pointers (char *) can point to strings
- String variables are char * variables in C

Declaring a char * variable does not allocate memory for a string
- e.g. “char *str; str[0] = ‘H’;” will crash your program
Strings

• Two ways to (have your compiler) allocate memory for a string
  – String constants (e.g. “HELLO”)
    • Memory allocated is immutable
    • For strings whose contents will never change
  – Character arrays
    • e.g. “char s[10]” can hold a string 9 characters long (excluding the null character)
    • Memory allocated is mutable
    • For declaring memory that can store different strings

• char * variable can point to various string constants or character arrays at different times
## Basic 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 <code>'\0'</code> 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>char *strcpy(char *dest, const char *src);</code></td>
<td>Copies the string pointed to by <code>src</code> (including the terminating <code>'\0'</code> character) to the array pointed to by <code>dest</code>.</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 <code>'\0'</code> character at the end of <code>dest</code>, and then adds a terminating <code>'\0'</code> character.</td>
</tr>
</tbody>
</table>
# String Functions w/ Limited Length

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int strcmp(const char *s1, const char *s2, size_t n);</code></td>
<td>Same as strcmp, except only compares the first (at most) n characters of s1 and s2</td>
</tr>
<tr>
<td><code>char *strncpy(char *dest, const char *src, size_t n);</code></td>
<td>Same as strcpy, except that not more than n bytes of src are copied. If src is longer than n bytes, make sure to append '\0' character to dest.</td>
</tr>
<tr>
<td><code>char *strncat(char *dest, const char *src, size_t n);</code></td>
<td>Same as strcat, except that it will use at most n characters from src. If src is longer than n bytes, make sure to append '\0' character to dest.</td>
</tr>
</tbody>
</table>

- Use the “n” version to prevent memory overruns
  - E.g. “char str[5]; strcpy(str, “Hello World”);” will cause overrun
  - Use “char str[5]; strncpy(str, “Hello World”, 4); str[4] = '\0’;” instead
# String Search Functions

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

- **String -> 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>

- **Number -> string:** 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”)

---

**Prototype:**

- `int atoi(const char *nptr);` Converts string pointed to by nptr to int.
- `double atof(const char *nptr);` Converts string pointed to by nptr to double.

**Description:**

- **Prototype:**
  - `int sprintf(char *str, const char *format, ...);` Same as printf except instead of writing to stdout formatted string is written to str.
  - `int snprintf(char *str, size_t size, const char *format, ...);` Same as above, except no more than size bytes are written to str.
Pitfall 1: Array bounds

• What’s wrong with the following code?
  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 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]
- Solution:
  if(strcmp(s, “Hello”) == 0) printf(“Hello\n”);
Pitfall 3: Random Pointer Access

• What’s wrong with the following code?
  
  int *a;
  
  a[5] = 0;

• Since “a” is not initialized, it may point to any random memory location

• “a[5]” will access a random location leading to
  – Program crash (segmentation fault) or
  – Memory corruption
Pitfall 4: String buffer allocation

• What’s wrong with the following code?
  char *s1;
  strcpy(s1, s2);
• “s1” points to random memory address
• Solution: “char s1[100];”
• While we are at it, change strcpy to strncpy
  char s1[100];
  strncpy(s1, s2, 99); // to prevent buffer overflow
  s1[99] = '\0'; // in case s2 is longer than 100 chars

>> ./a.out
Segmentation fault (core dumped)
Pitfall 5: String Constants are Immutable

• What’s wrong with the following code?
char *str = "Hello";
str[0] = 'Y';
• “str” points to a string constant
• Solution:
char str[6]; // 1 character more for null character
strcpy(str, “Hello”); // length known; no ‘n’ needed
str[0] = ‘Y’;
Review of Data Types

• Primitive data types
  – integers: short, int, long, long long
  – reals: float, double, long double
  – text: char

• 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]
  – also used to hold strings

• Given array int a[2][3], the following are valid
  – int (*p)[3] = a; (same as int (*p)[3] = &a[0];)
  – int *p = a[0]; (same as int *p = &a[0][0];)
  – int p = a[0][0];