Arrays and Strings

CS449 Fall 2017
Arrays

• Data type for a sequence of variables of the given element type in consecutive memory

• Element type can be any data type. E.g.
  – char A[10]; // sequence of chars
  – int A[10]; // sequence of ints
  – int *A[10]; // sequence of int pointers
#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;
}
Declaring 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;
}

• Syntax: <elem type> <name> [<length>]
  – E.g. “int num[5];”, “char c[10];”
• Meaning: Declares a sequence of <length> variables of <elem type> and associates them with <name>
• Length must always be specified
  – Combination of element type and length tells the compiler amount of memory to reserve:
    <length> * sizeof(<elem 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 is implicitly specified as 3
# Accessing Array Elements

```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:** `<array> [ <index> ]`
  - E.g. “nums[5]”, “c[10]”
- **Meaning:** `<index>` th variable in `<array>`
- First variable starts with index 0
- **Warning:** accessing index beyond length of array will result in access to memory not reserved for array. This may cause:
  - Program crash (segmentation fault)
  - Data corruption (overwriting other variables)
#include <stdio.h>

int main()
{
    int a[3] = {0, 1, 2}; /* int array */
    int *p; /* int pointer */
    printf("a = 0x%08x\n", a);
    printf("&a[0] = 0x%08x\n", &a[0]);
    p = a;
    printf("p = 0x%08x\n", p);
    a = p; /* ILLEGAL */
}
Why Pointer to First Element?

• Let’s look at the [] operator: “<array> [ <index> ]”
  – Operands: <array>, <index>
  – Result: memory location of <index>th element starting from element 0

• How is the address for the memory location calculated?
  – <address of element 0> + <index> * <element size>

• Therefore, the value of <array> must provide the following info:
  – Address of element 0
  – Element size

• Therefore, the value of an array must be a pointer to the first element
  – Value of pointer provides the address of element 0
  – Type of pointer provides element size (size of base type of pointer)

• <array> does not have to be an actual array. Can be …
  – An expression returning a pointer value. E.g. “int x; (&x)[0] = 0;”
Using Pointers in Place of Arrays

```c
#include <stdio.h>

int main()
{
    int a[3] = {0, 1, 2}; /* int array */
    int *p; /* int pointer */
    printf("a = 0x%08x\n", a);
    p = a; /* p == &a[0] */
    printf("p = 0x%08x\n", p);
    printf("p[1] = %d\n", p[1]);
    p = &a[1];
    printf("p = 0x%08x\n", p);
    printf("p[1] = %d\n", p[1]);
    return 0;
}
```

```
> ./a.out
a = 0x6ae5b400
p = 0x6ae5b400
p[1] = 1
p = 0x6ae5b404
p[1] = 2
```

- [] operator does not care whether operand is actual array or pointer
  - Pointers can be used in place of arrays (e.g. p in place of a)
- Pointers can even point to elements in the middle of an array
  - And still be accessed like arrays
**Passing Arrays as Pointers**

```c
#include <stdio.h>

int print_array (int *p) {
    printf("p = 0x%08x\n", p);
    printf("p[1] = %d\n", p[1]);
    return 0;
}

int main() {
    int a[3] = {0, 1, 2}; /* int array */
    printf("a = 0x%08x\n", a);
    print_array(a);
    print_array(&a[1]);
    return 0;
}
```

```
>> ./a.out
a = 0x6ae5b400
p = 0x6ae5b400
p[1] = 1
p = 0x6ae5b404
p[1] = 2
```

- Arrays can be passed to functions in the form of pointers
Declaring Multidimensional Arrays

- Syntax: `<type> <name> [ <length1> ] [ <length2> ] ... [ <lengthN> ]`
  - E.g. “int nums[2][3];”
- Array initialization
  - E.g. “int nums[2][3] = {{0, 1, 2}, {3, 4, 5}};”
- 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></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>
- E.g. “nums[1][2];” accesses offset 5 (1 * 3 + 2) in linear memory
- Row major (Row 0, row 1, ...)
Value of a Multidimensional Array

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

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

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

- Consider the following two examples:

  ```c
  int nums[2][3];
  int (*p)[3] = nums;
  p[0] == nums[0];
  p[0][0] == nums[0][0];
  int nums[2][3];
  int *q = nums[0];
  q[0] == nums[0][0];
  q[0][0] == ??? /* ILLEGAL! */
  ```

<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>
Strings

• String: sequence of characters in consecutive memory ending with a null character (‘\0’);
  – e.g. ”HELLO”:
    "H" "E" "L" "L" "O" "\0"

• There is no separate “string” data type in C
  – char * variable is used to refer to strings. E.g.
    char *str = “HELLO”;
  – Strings can be accessed in the same way as arrays. E.g.
    str[0] == ‘H’; /* in above example */

• Declaring a char * variable does not allocate memory for a string. E.g.
  char *str;  /* Dangling pointer */
  str[0] = ‘H’; /* CRASH. Access to unallocated memory. */
Strings

• Two ways to allocate memory for a string
  – Declaring character array variables
    • E.g. “char str[10];” can hold a string 9 characters long (excluding the null character)
    • Memory reserved is mutable: can modify string
  – Using string constants (e.g. “HELLO”)
    • Memory reserved is immutable: cannot modify string
    • For strings whose contents will never change

• Example:
```c
char str[10];
char *p = str;
p[0] = 'H';  /* str[0] == 'H' */
p = "HELLO";
p[0] = 'H';  /* CRASH. Writing immutable memory */
```
# 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>int strncmp(const char *s1, const char *s2, size_t n);</td>
<td>Same as strcmp, except only compares the first (at most) n characters of s1 and s2</td>
</tr>
<tr>
<td>char *strncpy(char *dest, const char *src, size_t n);</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>char *strncat(char *dest, const char *src, size_t n);</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, src);” may cause overrun
  - Use “char str[5]; strncpy(str, src, 4); str[4] = ‘\0’;” instead
# String Search 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 -> number:** Conversion functions declared in `<stdlib.h>`

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int atoi(const char *nptr);</td>
<td>Converts string pointed to by nptr to int.</td>
</tr>
<tr>
<td>double atof(const char *nptr);</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>int sprintf(char *str, const char *format, ...);</td>
<td>Same as printf except instead of writing to stdout formatted string is written to str</td>
</tr>
<tr>
<td>int snprintf(char *str, size_t size, const char *format, ...);</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: Array bounds

• What’s wrong with the following code?

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

• “a[5]” accesses a location beyond the array bounds so the value is undefined (whatever happens to be there)

• 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 pointer &s[0]
• Solution:
if(strcmp(s, “Hello”) == 0) printf(“Hello\n”);
Pitfall 3: Dangling 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” is a dangling pointer
• 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]

• Arrays (derived type that is a sequence of the given type)
  – char a[3], int a[3], int *a[3]
  – Char arrays can be used to hold strings

• Given array int a[2][3], the following are true
  – a == &a[0]; // type: int (*)(3], pointer to array of 3 ints
  – a[0] == &a[0][0]; // type: int *, pointer to int