CS61C – Machine Structures

Lecture 4 – C Pointers and Arrays

1/25/2006

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Common C Error

- There is a difference between assignment and equality
  - `a = b` is assignment
  - `a == b` is an equality test
- This is one of the most common errors for beginning C programmers!

Pointers & Allocation (1/2)

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet *(well actually points somewhere - but don't know where!).* We can either:

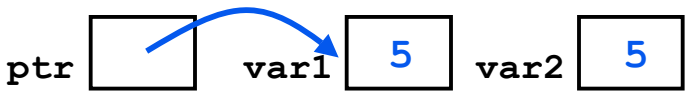
  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to… (next time)

Pointers & Allocation (2/2)

- Pointing to something that already exists:

```
int *ptr, var1, var2;
var1 = 5;
ptr  = &var1;
var2 = *ptr;
```

- `var1` and `var2` have room implicitly allocated for them.



More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.
- What does the following code do?

```
void f()
{
    int *ptr;
    *ptr = 5;
}
```


Pointers in C

- Why use pointers?
  - If we want to pass a huge struct or array, it's easier to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.

Arrays (1/6)

- **Declaration**:

```
int ar[2];
```

declares a 2-element integer array. *An array is really just a block of memory.*

```
int ar[] = {795, 635};
```

declares and fills a 2-elt integer array.

- **Accessing elements**:

```
ar[num];
```

returns the `num`th element.

Arrays (2/6)

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - They differ in very subtle ways: incrementing, declaration of filled arrays
- **Key Concept**: An array variable is a "pointer" to the first element.

Arrays (3/6)

- **Consequences:** `int ar[10];`
  - `ar` is an array variable but looks like a pointer in many respects (though not all)
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.
- **Declared arrays are only allocated while the scope is valid**

```
char *foo() {
   char string[32]; ...;
   return string;
}
```
is incorrect

Arrays (4/6)

- **Array size `n`; want to access from `0` to `n-1`, can test for exit by comparing to address one element past the array**

```
int ar[10], *p, *q, sum = 0;
...
p = &ar[0]; q = &ar[10];
while (p != q)
    /* sum = sum + *p; p = p + 1; */
    sum += *p++;
```

  - Is this legal?
- **C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error**

Arrays (5/6)

- Array size `n`; want to access from `0` to `n-1`, so you should use counter AND utilize a constant for declaration & incr
  - Wrong style

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```

  - Right style

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? SINGLE SOURCE OF TRUTH
  - You're avoiding maintaining two copies of the number 10

Arrays (6/6)

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find; be careful! (You'll learn how to debug these in lab…)

Segmentation Fault vs Bus Error?

° http://www.hyperdictionary.com/

° **Segmentation Fault**

- A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a "SIGBUS" signal which, if not caught, will terminate the current process.

° **Bus Error**

- An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump.

Pointer Arithmetic (1/3)

° Since a pointer is just a mem address, we can add to it to traverse an array.

`p+1` returns a ptr to the next array elt.

`(*p)+1` vs `*p++` vs `*(p+1)` vs `(*p)++` ?

```
x = *p++ ⇒ x = *p ; p =  p + 1;
x = (*p)++ ⇒ x = *p ; *p = *p + 1;
```

° What if we have an array of large structs (objects)?

C takes care of it: In reality, `p+1` doesn't add `1` to the memory address, it adds the size of the array element.

Pointer Arithmetic (2/3)

- So what's valid pointer arithmetic?
  - Add an integer to a pointer.
  - Subtract 2 pointers (in the same array).
  - Compare pointers (<, <=, ==, !=, >, >=)
  - Compare pointer to `NULL` (indicates that the pointer points to nothing).
- Everything else is illegal since it makes no sense:
  - adding two pointers
  - multiplying pointers
  - subtract pointer from integer

Pointer Arithmetic (3/3)

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
- So the following are equivalent:

```
int get(int array[], int n)
{
    return  (array[n]);
    /* OR */
    return *(array + n);
}
```

C Strings

- A string in C is an array of characters.

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (null terminator)

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```


C Strings Headaches

- One common mistake is to forget to allocate an extra byte for the null terminator.
- More generally, C requires the programmer to manage memory manually (unlike Java or C++).
  - When creating a long string by concatenating several smaller strings, the programmer must insure there is enough space to store the full string!
  - What if you don't know ahead of time how big your string will be?
  - Buffer overrun security holes!

Pointer Arithmetic Question:

How many of the following are invalid?

I. pointer + integer
II. integer + pointer
III. pointer + pointer
IV. pointer – integer
V. integer – pointer
VI. pointer – pointer
VII. compare pointer to pointer
VIII. compare pointer to integer
IX. compare pointer to 0
X. compare pointer to `NULL`

"And in Conclusion…"

- Pointers and arrays are virtually same
- C knows how to increment pointers
- C is an efficient language, with little protection
  - Array bounds not checked
  - Variables not automatically initialized
- (Beware) The cost of efficiency is more overhead for the programmer.
  - "C gives you a lot of extra rope but be careful not to hang yourself with it!"

Bonus Slide: Arrays/Pointers

- An array name is a read-only pointer to the 0th element of the array.
- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0)
        n++;
    return n;
}
```

```
int strlen(char *s)
{
    int n = 0;
    while (s[n] != 0)
        n++;
    return n;
}
```

Could be written:
`while (s[n])`

Bonus Slide: Pointer Arithmetic

- We can use pointer arithmetic to "walk" through memory:

```
void copy(int *from, int *to, int n) {
    int i;
    for (i=0; i<n; i++) {
        *to++ = *from++;
    }
}
```

- C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a `char`, 4 bytes for an `int`, etc.)