CS61C : Machine Structures

Lecture 1.2.1
C Pointers, Arrays, and Strings

2004-06-23

Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

CSUA Helpsessions

CSUA
(Computer Science Undergraduate Association)

Helpsessions:

How to Use Unix
Thursday 6/24 6pm in 306 Soda

How to use Emacs!
Wednesday 6/30 7pm in 306 Soda

Compilation : Overview

C *compilers* take C and convert it into an architecture specific machine code (string of 1s and 0s).

- **Unlike Java which converts to architecture independent bytecode.**
- **Unlike most Scheme environments which interpret the code.**
- **Generally a 2 part process of compiling .c files to .o files, then linking the .o files into executables**
  - **"gcc –g –Wall –o myprog myprog.c"**

C vs. Java™ Overview (1/3)

Java	C
• Object-oriented (OOP)	• No built-in object abstraction. Data separate from methods.
• "Methods"	• "Functions"
• Automatic memory management	• Manual memory management
	• Pointers

C vs. Java™ Overview (2/3)

Java	C
• Arrays initialize to zero	• Arrays initialize to garbage

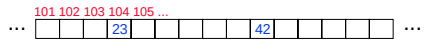
• Syntax:
`/* comment */`
`//comment`
`System.out.print`
	• Syntax:
	`/* comment */`
	`printf`

C vs. Java™ Overview (3/3)

Java	C
• Declare vars pretty much anywhere.	• Var declarations only at top of block.
• Explicit Boolean Type with NO Type Coercion	• No Booleans, only 0/NULL are false, but with coercion.
• main (String argv[])	• main (int argc, char *argv[])

Address vs. Value

- What good is a bunch of memory if you can't select parts of it?
  - Each memory cell has an address associated with it.
  - Each cell also stores some value.
- Don't confuse the address referring to a memory location with the value stored in that location.



Pointers

- A pointer is just a C variable whose value is the address of another variable!
- After declaring a pointer:

```
int *ptr;
```

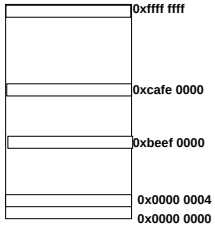
`ptr` doesn't actually point to anything yet. We can either:

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

Pointers

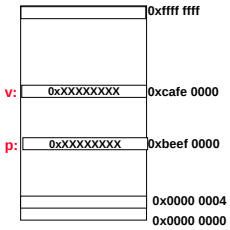
- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.

Pointer Usage Example



Memory and Pointers:

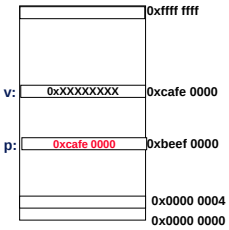
Pointer Usage Example



Memory and Pointers:

int *p, v;

Pointer Usage Example

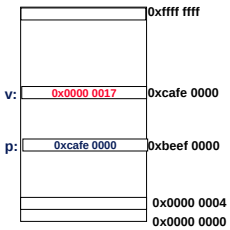


Memory and Pointers:

int *p, v;

p = &v;

Pointer Usage Example



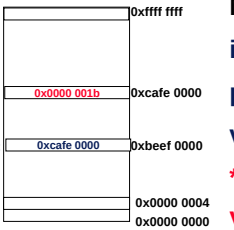
Memory and Pointers:

int *p, v;

p = &v;

v = 0x17;

Pointer Usage Example



Memory and Pointers:

int *p, v;

p = &v;

v = 0x17;

*p = *p + 4;

V = *p + 4

C Pointer Dangers

- What does the following code do?

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

- S E G F A U L T ! (on my machine/os)
  - (Not a nice compiler error like you would hope!)

Pointers and Parameter Passing

- Java and C pass a parameter "by value"
  - procedure/function gets a copy of the parameter, so changing the copy cannot change the original

```
void addOne (int x) {
   x = x + 1;
}
int y = 3;
addOne(y);
```

- y is still = 3

Pointers and Parameter Passing

- How to get a function to change a value?

```
void addOne (int *p) {
  *p = *p + 1;
}
int y = 3;

addOne(&y);
```

- y is now = 4

Arrays (1/7)

- Declaration:

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

declares a 2-element integer array.

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

declares and fills a 2-elt integer array.

- Accessing elements:

```
ar[num];
```

returns the `num`th element from 0.

Arrays (2/7)

- 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 Difference:

  An array variable is a CONSTANT pointer to the first element.

Arrays (3/7)

- Consequences:
  - `ar` is a pointer
  - `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 (4/7)

- Array size `n`; want to access from `0` to `n-1`:

```
int ar[10], i=0, sum = 0;
...
while (i < 10)
    /* sum = sum+ar[i];
        i = i + 1; */

    sum += ar[i++];
```

Arrays (5/7)

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

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

  - Right

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

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

Arrays (6/7)

- 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…

Arrays 7/7: In Functions

- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.
  - Can be incremented

```
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])`

Pointer Arithmetic (1/5)

- Since a pointer is just a memory address, we can add to it to traverse an array.
- `ptr+1` will return a pointer to the next array element (nomatter how big).
- `(*ptr)+1` vs. `*ptr++` vs. `*(ptr+1)` ?
- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, `ptr+1` doesn't add `1` to the memory address, it adds the size of the array element.

Pointer Arithmetic (2/5)

- 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 (3/5)

- 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.)

Pointer Arithmetic (4/5)

- 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);
}
```

Pointer Arithmetic (5/5)

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

```
int ar[10], *p, *q, sum = 0;
...
p = ar; 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