Agenda

- Review
- Pointers
- Administrivia
- Arrays
- Technology Break
- Summary

Review

- C is a popular and efficient compiled language used for systems programming and application development
  - Derivatives include C++, Objective-C, C#
  - Java borrowed much of the basic syntax
- Very close to machine, so sometimes difficult to program and debug
- Key concepts last time: compilation, preprocessor, typed data and functions, main(), structs, enums, control flow

Address vs. Value

- Consider memory to be a single huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don’t confuse the address referring to a memory location with the value stored there

Pointers

- An address refers to a particular memory location; e.g., it points to a memory location
- Pointer: A variable that contains the address of a variable

Pointer Syntax

- int *x;
  - Tells compiler that variable x is address of an int
- x = &y;
  - Tells compiler to assign address of y to x
    - & called the “address operator” in this context
- z = *x;
  - Tells compiler to assign value at address in x to z
    - * called the “dereference operator” in this context
Creating and Using Pointers

• How to create a pointer:
  
  & operator: get address of a variable
  
  int *p, x;  
  
  p = &x;  
  
  x = 3;  
  
  p = &x;  

• How to get a value pointed to?

  * (dereference operator): get the value that the pointer points to
  
  printf("p points to \%d\n", *p);

Types of Pointers

• Pointers are used to point to any kind of data (int, char, struct, etc.)
• Normally a pointer only points to one type (int, char, struct, etc.).
  – void * is a type that can point to anything (generic pointer)
  – Use sparingly to help avoid program bugs, and security issues, and other bad things!
Pointers and Structures

```c
struct Point {
    int x;
    int y;
};

Point p1;
Point p2;
Point *paddr;
```

/* dot notation */
```c
int h = p1.x;
p2.y = p1.y;
```

/* arrow notation */
```c
int h = paddr->x;
int h = (*paddr).x;
```

/* This works too */
```c
p1 = p2;
```

Why Pointers in C?

• Why use pointers?
  - If we want to pass a large struct or array, it’s easier / faster / etc. 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 C, so be careful anytime you deal with them
  - Most problematic with dynamic memory management—which you will to know by the end of the semester, but not for the projects there will be a lab later in the semester
  - Dangling references and memory leaks

Peer Instruction Answer

```c
#include <stdio.h>
void main()
{
    int *p, x=-5, y="init
    y = *(p = &x) + 1;
    int z;
    flip-sign(p);
    printf("x=\%d,y=\%d,p=\%d\n",x,y,p);
} flip-sign(int *n){*n = -*n;
```  

5 syntax + logic errors in this C code

How many logic and syntax errors?

- 1
- 2
- 3
- 4

What is output after correct errors?

```c
void main()
{
    int *p, x=5, y="init
    y = *(p = &x) + 1;
    int z;
    flip-sign(p);
    printf("x=\%d,y=\%d,p=\%d\n",x,y,p);
} flip-sign(int *n){*n = -*n;
```  

x=5, y=6, p=-5
x=-5, y=6, p=-5
x=-5, y=4, p=-5
x=-5, y=-6, p=-5

9/9/12 Fall 2012 - Lecture #5 14
**Administrivia**

- HW #2
- Lab #2
- No Discussion sections this week, but extra office hours...

**Extra Office Hours This Week**

(Alan Christopher: Tue 5-7 320 Soda)  
Loc Do: Fri 12-2pm Qualcomm Cyber Café  
(James Ferguson: Tue 10-11 AM & 1-2 PM, 320 Soda)  
Anirudh Garg: Wed 4-6 Qualcomm Cyber Café  
William Ku: 6-8 Thursday 283E Soda  
Brandon Luong: Wed 6-8 283E Soda  
Sung Roa Yoon: Thu 7-9 283E Soda.

*Watch Piazza!*

**The Rules**  
(and we really mean it!)

**Arrays (1/5)**

- Declaration:  
  ```c
  int ar[2];
  ```
  declares a 2-element integer array: just a block of memory  
- Accessing elements:  
  ```c
  int ar[] = {795, 635};
  ```
  declares and initializes a 2-element integer array  
- Accessing elements:  
  ```c
  ar[num]
  ```
  returns the numth element

**Arrays (2/5)**

- C arrays are (almost) identical to pointers  
  - `char *string` and `char string[]` are nearly identical declarations  
  - Differ in subtle ways: incrementing, declaration of filled arrays  
  - End of C string marked by 0 in last character  
- Key Concept: Array variable is a “pointer” to the first (0th) element

**C Strings**

- String in C is just an array of characters  
  ```c
  char string[] = "abc";
  ```
- How do you tell how long a string is?  
  - Last character is followed by a 0 byte (aka “null terminator”)  
  ```c
  int strlen(char s[])
  {
  int n = 0;
  while (s[n] != 0) n++;
  return n;
  }
  ```
Arrays (3/5)

- Consequences:
  - `ar` is an array variable, but looks like a pointer
  - `ar[0]` is the same as `*(ar+0)`
  - We can use pointer arithmetic to conveniently access arrays
- Declared arrays are only allocated while the scope is valid

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

is incorrect and very very bad

Arrays (4/5)

- Array indexing is syntactic sugar for pointers
  - `a[i]` is treated as `*(a+i)`
  - E.g., three equivalent ways to zero an array:
    - for (i=0; i < size; i++) `a[i] = 0;`
    - for (i=0; i < size; i++) `*(a+i) = 0;`
    - for (p=a; p < a+size; p++) `*p = 0;`

Incrementing a pointer makes it point to the next variable in memory (type of pointer says how big each variable is)

Arrays (5/5)

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

Array Summary

- Array indexing is syntactic sugar for pointers
- `a[i]` is treated as `*(a+i)`
- E.g., three equivalent ways to zero an array:
  - for (i=0; i < size; i++) `a[i] = 0;`
  - for (i=0; i < size; i++) `*(a+i) = 0;`
  - for (p=a; p < a+size; p++) `*p = 0;`

What is TRUE about this function?

```c
void foo(char *s, char *t) {
    while (*s)
        s++;
    while (*s++ = *t++)
        ;
}
```

- No syntax errors; it changes characters in string `t` to next character in the string `a`
- No syntax errors; it copies a string at address `t` to the string at address `a`
- No syntax errors; it appends the string at address `t` to the end of the string at address `a`

Question: Which statement is FALSE regarding C and Java?

- Arrays in C are just pointers to the 0-th element
- As Java was derived from C, it has the same control flow constructs
- Like Java, in C you can check the length of an array `a.length` gives no. elements in `a`
- C has pointers but Java does not allow you to manipulate pointers or memory addresses of any kind
FYI—Update to ANSI C

• "C99" or "C9X" standard
  
  • gcc -std=c99 to compile

• References
  
  
  • http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html

• Highlights
  
  • Declarations in for loops, like Java
  
  • Java-like // comments (to end of line)
  
  • Variable-length non-global arrays
  
  • <inttypes.h>: explicit integer types
  
  • <stdbool.h>: for boolean logic types and definitions

And In Conclusion, ...

• All data is in memory
  
  • Each memory location has an address to use to refer to it and a value stored in it

• Pointer is a C version (abstraction) of a data address
  
  • * “follows” a pointer to its value
  
  • & gets the address of a value
  
  • Arrays and strings are implemented as variations on pointers

• C is an efficient language, but leaves safety to the programmer
  
  • Array bounds not checked
  
  • Variables not automatically initialized
  
  • Use pointers with care: they are a common source of bugs in programs