CS61C
Pointers, Arrays, and
Wrapup of Assembly Language

Lecture 11

February 24, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

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

Review 1/1

- Stored Program concept mean instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  - Compiler Assembler Linker (Loader)
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses

Outline

- Address vs. Value
- C Pointer concepts and MIPS implementations and problems
- Administrivia, “What’s this stuff good for”
- HLL/Asm Wrap up: multidimensional arrays, `sizeof()`, &&, ||, l-format
- Difficulties of pointers for 61C students: OpEds by Harvey, Brewer, Hilfinger
- C pointer difficulties don’t apply to Java? (if time)
- Conclusion

Address vs. Value

- Fundamental concept of Comp. Sci.
- Even in Spreadsheets: select cell A1 for use in cell B1

	A	B
1	100	100
2		

- Do you want to put the address of cell A1 in formula (=A1) or A1's value (100)?
- Difference? When change A1, cell using address changes, but not cell with old value

Address vs. Value in C

- **Pointer: a variable that contains the address of another variable**
  - **HLL version of machine language address**
- **Why use Pointers?**
  - **Sometimes only way to express computation**
  - **Often more compact and efficient code**
- **Why not? (according to Eric Brewer)**
  - **Huge source of bugs in real software, perhaps the largest single source**

  1) Dangling reference (premature free)

  2) Memory leaks (tardy free): can't have long-running jobs without periodic restart of them

C Pointer Operators

- Suppose c has value 100, located in memory at address 0x10000000
- Unary operator & gives address: `p = &c;` gives address of c to p;
  - p "points to" c
  - `p == 0x10000000`
- Unary operator * gives value that pointer points to: if `p = &c;` then
  - "Dereferencing a pointer"
  - `* p == 100`

Assembly Code to Implement Pointers

- deferencing data transfer in asm.
  - ... = ... *p ...; load
    (get value from location pointed to by p)
  - *p = ...; store
    (put value into location pointed to by p)

Assembly Code to Implement Pointers

- c is int, has value 100, in memory at address 0x10000000, p in $a0, x in $s0

```
p = &c;  /* p gets 0x10000000 */
x = *p;  /* x gets 100 */
*p = 200; /* c gets 200 */
```

```
# p = &c;  /* p gets 0x10000000 */
lui $a0,0x1000 # p = 0x10000000

# x = *p;  /* x gets 100 */
lw  $s0, 0($a0) # dereferencing p

# *p = 200; /* c gets 200 */
addi $t0,$0,200
sw   $t0, 0($a0) # dereferencing p
```

Registers and Pointers

- Registers do not have addresses
  - registers cannot be pointed to
  - cannot allocate a variable to a register if it may have a pointer to it

C Pointer Declarations

- C requires pointers be declared to point to particular kind of object (`int`, `char`, ...)
- Why? Safety: fewer problems if cannot point everywhere
- Also, need to know size to determine appropriate data transfer instruction
- Also, enables pointer calculations
  - Easy access to next object: p+1
  - Or to i-th object: p+i
  - Byte address? multiplies i by size of object

C vs. Asm

```
int strlen(char *s) {
 char *p = s; /* p points to chars */

  while (*p != '\0')
    p++; /* points to next char */
 return p - s; /* end - start */

}
```

```
        mov  $t0,$a0
        ldbu $t1,0($t0) /* derefence p */
        beq  $t1,$zero, Exit

Loop:   addi $t0,$t0,1  /* p++ */
        ldbu $t1,0($t0) /* derefence p */
        bne  $t1,$zero, Loop

Exit:   sub $v0,$t1,$a0
        jr $ra
```

C pointer "arithmetic"

- What arithmetic OK for pointers?
  - Add an integer to a pointer: `p+i`
  - Subtract 2 pointers (in same array): `p-s`
  - Comparing pointers (<, <=, ==, !=, >, >=)
  - Comparing pointer to 0: `p == 0`
    (0 used to indicate it points to nothing; used for end of linked list)
- Everything else illegal (adding 2 pointers, multiplying 2 pointers, add float to pointer, ...)
  - Why? Makes no sense in a program

Common Pointer Use

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

```
int a[10], *q, sum = 0;

...
p = &a[0]; q = &a[10];
while (p != q)
  sum = 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

Common Pointer Mistakes

- Common error; Declare and write:

```
int *p;
*p = 10; /* WRONG */
```

  - What address is in p? (NULL)

- C defines that memory location 0 must not be a valid address for pointers
  - NULL defined as 0 in <stdio.h>

Common Pointer Mistakes

- Copy pointers vs. values:

```
int *ip, *iq, a = 100, b = 200;
ip = &a; iq = &b;
*ip = *iq; /* what changed? */
ip = iq;   /* what changed? */
```

Pointers and Heap Allocated Storage

- Need pointers to point to `malloc()` created storage
- What if free storage and still have pointer to storage?
  - "Dangling reference problem"
- What if don't free storage?
  - "Memory leak problem"

Multiple pointers to same object

- Multiple pointers to same object can lead to mysterious behavior

```
int *x, *y, a = 10, b;
...
y = & a;
...
x = y;
...
*x = 30;
...
/* no use of *y */
printf("%d", *y);
```

Administrivia

- Readings: Pointers: COD: 3.11, K&R Ch. 5; I/O 8.1, 8.2, 8.3, 8.5, A.7, A.8
- 6th homework: Due 3/3 7PM
  - Exercises 8.1, 8.5, 8.8
- 3rd Project/5th Lab: MIPS Simulator
  Due Wed. 3/3 7PM; deadline Thurs 8AM

“What’s This Stuff Good For?”

For the Refrigerator, a Silicon Chip Treat

The Internet refrigerator has long been a kind of Silicon Valley joke (killer app: crispy lettuce) but don't tell that to the people at Electrolux. In tandem with the software company ICL, the appliance manufacturer recently unveiled a prototype of the **Screenfridge**, a fridge-freezer equipped with a touch screen and a bar-code scanner.

The touch screen has an Internet link, including e-mail, and the scanner can be used to swipe items, entering them directly into a Web shopping list.

"After PC modems and interactive TV, we were looking for the most appropriate room in the house for electronic commerce," said Sion Roberts, ICL's vice president for interactive retailing.

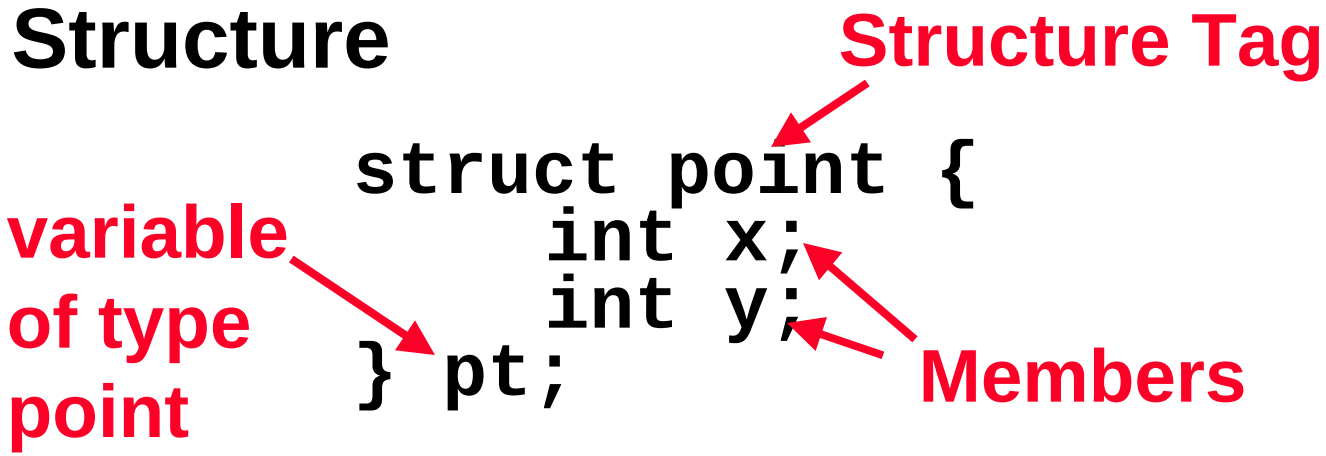
"The kitchen is the focus for the household community."

The Screenfridge will be tested in Europe this year, and the companies are working on a separate screen-only unit that can sit on the doors of existing fridges "like a giant, modern-day fridge magnet."

N.Y.Times, 2/18/99

Structures and Shorthand for Pointers

- Structure



- Structures often used with pointers

```
struct point *pp;
pp = &pt;
z =(*pp).x /* member x of pt*/
```

- Alternative notation: ->

```
z = pp->x /* member x of pt*/
```

- Similar concept in C++, Java

Multidimensional Arrays and Functions

- Bug in code From Lecture 9

```
void mm (double x[][], double y[][],
         double z[][]){
 int i, j, k;

 for (i=0; i!=32; i=i+1)
   for (j=0; j!=32; j=j+1)
     for (k=0; k!=32; k=k+1)
         x[i][j] = x[i][j] + y[i][k]
                   * z[k][j];

}
```

- 2-dimesional array parameter must include number of columns, to know size of row

Multidimensional Arrays and Functions

- Patched code From Lecture 9

```
void mm (double x[][32],
     double y[][32], double z[][32]){
 int i, j, k;

...

}
```

- Allows compiler to calculate address of `x[i][j]` since calculation is `i*rowsize + j`

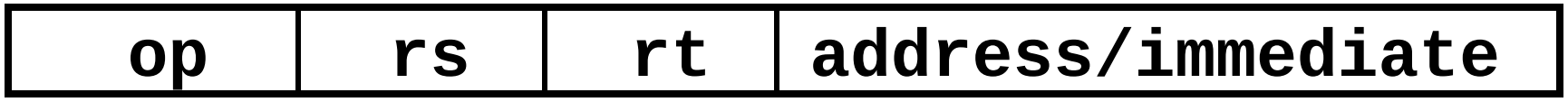
Sizeof and Structures

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading, so use `sizeof(type)`
  - Many years ago int was 16 bits, and programs assumed it was 2 bytes
- How big is `sizeof(p)`?

```
struct p {
            char x;
            int y;
        };
```

  - 5 bytes? 8 bytes?
  - Compiler may word align integer `y`

Number Representation for I-format



- Loads, stores treat the address as a 16-bit 2's complement number: -2^{15} to $2^{15}-1$ or -32768 to +32767 (0x1000 to 0x0111) added to $rs
  - Hence $gp set to 0x10001000 so can easily address from 0x10000000 to 0x10001111
- Most immediates represent same values: `addi`, `addiu`, `slti`, `sltiu`
- `andi`, `ori` consider immediate a 16-bit unsigned number: 0 to $2^{16}-1$, or 0 to 65535 (0x0000 to 0x1111)

&&, || operators in C

- How state “if c is blank or tab character”?
  - `if ( c == ' ') ___;`
    `else if ( c == '\t') ___;`
  - logical OR operator | | for such situations:
    `if ( c >= ' ' || c == '\t') ___;`
- How state “if c is a digit (0<=c<=9)”?
  - `if ( c >= '0') if ( c <= '9') ___;`
  - logical AND operator && for such situations:
    `if ( c >= '0' && c <= '9') ___;`
- Note: C defines numeric value of relational expression to be 1 if true, 0 if false

Floating Point Registers

- Only 16 available for computation, either single or double: $f0, $f2, $f4, ...
  - $f1, $f3, $f5 just used for loading/storing 2nd half of double precision number

```
lwc1 $f0, 32($sp)
lwc1 $f1, 36($sp)
```

What is easier for compiler to optimize?

- Use of pointers reduces computation at source level, so compiled code may be more efficient
- Since compiler must not keep things in registers that may have a pointer to it, its harder for compiler to optimize by code
  - More difficult to restrict where pointer might point than using indices in arrays
- Pointers allow programmer optimization, exacerbate compiler optimization

Common Problems with Pointers: Brewer

- 1) heap-based memory allocation (malloc/free in C or new/delete in C++) is a huge source of bugs in real software, perhaps the largest single source. The worse problem is the dangling reference (premature free), but the lesser one, memory leaks, mean that you can't have long-running jobs without restarting them periodically

Common Problems with Pointers: Brewer

- 2) aliasing: two pointers may have different names but point to the same value. This is really the more fundamental problem of pass by reference (i.e., pointers): people normally assume that they are the only one modifying the an object
- This is often not true for pointers -- there may be other pointers and thus other modifiers to your object. Aliasing is the special case where you have both of the pointers...

Common Problems with Pointers: Brewer

- In general, pointers tend to make it unclear if you are sharing an object or not, and whether you can modify it or not. If I pass you a copy, then it is yours alone and you can modify it if you like. The ambiguity of a reference is bad; particularly for return values such as getting an element from a set -- is the element a copy or the master version, or equivalently do all callers get the same pointer for the same element or do they get a copy. If it is a copy, where did the storage come from and who should deallocate it?

Common Problems with Pointers: Harvey

- Some of them will write C code in which they declare a pointer and then try to dereference it without allocating any memory for it to point to.

Common Problems with Pointers: Hilfinger

- 1. Some people do not understand the distinction between x = y and *x = *y.
- 2. Some simply haven't enough practice in routine pointer-hacking, such as how to splice an element into a list.
- 3. Some do not understand the distinction between struct Foo x; and struct Foo *x;
- 4. Some do not understand the effects of p = &x and subsequent results of assigning through dereferences of p, or of deallocation of x.

Java doesn't have these pointer problems?

- Java has automatic garbage collection, so only when last pointer to object disappears, object is freed
- Point 4 above not a problem in Java:

  "4. Some do not understand the effects of p = &x and subsequent results of assigning through dereferences of p, or of deallocation of x."
- What about 1, 2, 3, according to Hilfinger?

Java vs. C. vs. C++ Semantics?

1. The semantics of pointers in Java, C, and C++ are IDENTICAL. The difference is in what Java lacks: it does not allow pointers to variables, fields, or array elements. Since Java has no pointers to array elements, it has no "pointer arithmetic" in the C sense (a bad term, really, since it only means the ability to refer to neighboring array locations).

- When considering what Java, C, and C++ have in common, the semantics are the same.

Java pointer is different from meaning in C?

- 2. The meaning of "pointer semantics" is that after

```
y.foo = 3;
x = y;
x.foo += 1;
```

- y.foo is 4, whereas after

```
x = z;
x.foo += 1;
```

- y and y.foo are unaffected.
- This is true in Java, as it is true in C/C++ (except for their use of -> instead of '.').

Java vs. C pass by reference?

- 3. NOTHING is passed by reference in Java, just as nothing is passed by reference in C. A parameter is "passed by reference" when assignments to a parameter within the body means assignment to the actual value. In Java and legal C, for a local variable x (whose address is never taken) the initial and final values of x before and after

  f(x)

- are identical, which is the definition of "pass by value".

What about Arrays, Paul?

- 3. There is a common misstatement that "arrays are passed by reference in C". The language specification is quite clear, however: arrays are not passed in C at all.
  - (If you want to “pass an array” you must pass a pointer to the array, since you can’t pass an array at all)
- The semantics are really very clean---ALL values, whether primitive or reference---obey EXACTLY the same rule. We really HAVE to refrain from saying that "objects are passed by reference", since students have a hard enough time understanding that f(x) can't reassign x as it is.

"And in Conclusion.." 1/1

- Pointer is high level language version of address
  - Powerful, dangerous concept
- Like goto, with self-imposed discipline can achieve clarity and simplicity
  - Also can cause difficult to fix bugs
- C supports pointers, pointer arithmetic
- Java structure pointers have many of the same potential problems!
- Next: Input/Output (COD chapter 8)