CS 61C: Great Ideas in Computer Architecture

Lecture 3: *Pointers*

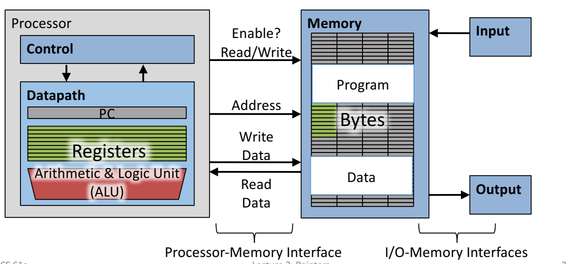
Krste Asanović & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c

Agenda

- **Pointers in C**
- Arrays in C
- This is not on the test
- Pointer arithmetic
- Strings, main
- And in Conclusion, …

Components of a Computer



Computer Memory

```
int a;

a = -85;

printf("%d", a);
```

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Do not confuse memory address and value.
Nor a street address with the person living there.

Pointers

- C speak for "memory addresses"
- Notation

```
int *x;      // variable x is an address to an int
int y = 9;   // y is an int
x = &y;      // assign address of y to x
             // "address operator"
int z = *x;  // assign what x is pointing to to z
             // "dereference operator"
*x = -7;     // assign -7 to what x is pointing to
```

What are the values of x, y, z?

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Pointer Type

- Pointers have types, like other variables
  - "type of object" the pointer is "pointing to"
- Examples:
  - `int *pi;     // pointer to int`
  - `double *pd; // pointer to double`
  - `char *pc;    // pointer to char`

Generic Pointer (void *)

- Generic pointer
  - Points to any object (int, double, …)
  - Does not "know" type of object it references (e.g. compiler does not know)
- Example:
  - `void *vp;` `// vp holds an address to` `// object of "arbitrary" type`
- Applications
  - Generic functions e.g. to allocate memory
  - `malloc, free`
    - accept and return pointers of any type
    - see next lecture

Pointer to `struct`

```
// type declaration
typedef struct { int x, y; } Point;

// declare (and initialize) Point "object"
Point pt = { 0, 5 };

// declare (and initialize) pointer to Point
Point *pt_ptr = &pt;

// access elements
(*pt_ptr).x = (*pt_ptr).y;

// alternative syntax
pp->x = pp->y;
```


Your Turn!

```
#include <stdio.h>

int main(void) {
    int a = 3, b = -7;
    int *pa = &a, *pb = &b;
    *pb = 5;
    if (*pb > *pa) a = *pa - b;
    printf("a=%d b=%d\n", a, b);
}
```

Answer	a	b
RED	3	-7
GREEN	4	5
ORANGE	-4	5
YELLOW	-2	5

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

What's wrong with this Code?

```
#include <stdio.h>

int main(void) {
    int a;
    int *p;
    printf("a = %d,  p = %p,  *p = %d\n",
           a, p, *p);
    return 0;
}
```

Output:

a = 1853161526,
p = 0x7fff5be57c08,
*p = 0

Pointers as Function Arguments

```
#include <stdio.h>

void f(int x, int *p) {
    x = 5;  *p = -9;
}

int main(void) {
    int a = 1, b = -3;
    f(a, &b);
    printf("a=%d b=%d\n", a, b);
}
```

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

- C passes arguments by value
  - i.e. it passes a copy
  - value does not change outside function
- To pass by reference use a pointer

Parameter Passing in Java

- "primitive types" (int, char, double)
  - by value (i.e. passes a copy)
- Objects
  - by reference (i.e. passes a pointer)
  - Java uses pointers internally
    - But hides them from the programmer
  - Mapping of variables to addresses is not defined in Java language
    - No address operator (&)
    - Gives JVM flexibility to move stuff around

Your Turn!

```c
#include <stdio.h>

void foo(int *x, int *y) {
    if ( *x < *y ) {
        int t = *x;
        *x = *y;
        *y = t;
    }
}

int main(void) {
    int a=3, b=1, c=5;
    foo(&a, &b);
    foo(&b, &c);
    printf("a=%d b=%d\n", a, b);
}
```

Type	Name	Addr	Value
		...	
		105	
		104	
		103	
		102	
		101	
		100	
		...	

Answer	a	b	c
RED	5	3	1
GREEN	1	5	3
ORANGE	3	3	1
YELLOW	3	5	1

Agenda

- Pointers in C
- **Arrays in C**
- This is not on the test
- Pointer arithmetic
- Strings, main
- And in Conclusion, …

C Arrays

- Declaration:
  - `// allocate space`
    `// unknown content`
    `int a[5];`
  - `// allocate & initialize`
    `int b = { 3, 2, 1 };`
- Element access:
  - `b[1];`
  - `a[2] = 7;`
- Index of first element: 0

Type	Name	Addr	Value
		...	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		...	

Beware: no array bound checking!

```c
#include <stdio.h>

int main(void) {
    int a[] = { 1, 2, 3 };

    for (int i=0;  i<4;  i++)
        printf("a[%d] = %d\n", i, a[i]);
}
```

Output:

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = -1870523725
```

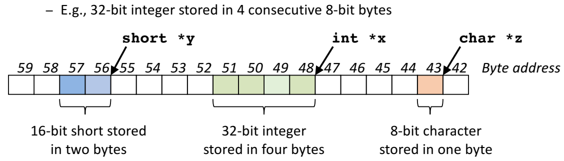
Often the result is much worse:
- erratic behavior
- segmentation fault, etc.
- C does not know array length!
- Pass as argument into functions

Use Constants, Not Literals

- Assign size to constant
  - Bad pattern
    ```c
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Better pattern
    ```c
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```
- "Single source of truth"
  - Avoiding maintaining two copies of the number 10
  - And the chance of changing only one
  - DRY: "Don't Repeat Yourself"

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  - Hardware's memory composed of 8-bit storage cells, each has a unique address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes



sizeof() operator

```c
#include <stdio.h>

int main(void) {
    double d;
    int array[5];
    struct { short a; char c; } s;

    printf("double: %2lu\n", sizeof(d));
    printf("array:  %2lu\n", sizeof(array));
    printf("s:      %2lu\n", sizeof(s));
}
```

Output:

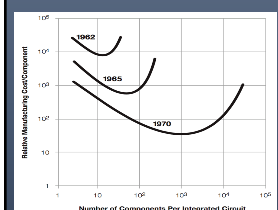
```
double:   8
array:   20
s:        4
```

- `sizeof(type)`
  - Returns number of bytes in object
  - Number of bits in a byte is not standardized
    - All modern computers: 8 bits per byte
    - Some "old" computers use other values, e.g. 6 bits per "byte"
- By definition, in C
  - `sizeof(char)==1`
- For all other types result is ***hardware and compiler dependent***
  - Do not assume - Use `sizeof`!

Agenda

- Pointers in C
- Arrays in C
- **This is not on the test**
- Pointer arithmetic
- Strings, main
- And in Conclusion, …

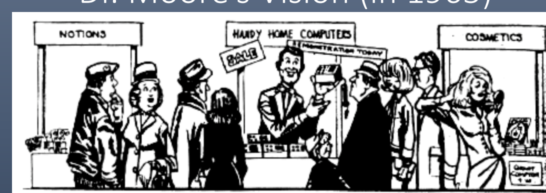
So what did Dr. Moore Predict?



- Transistor* cost as a function of components per chip
  - Minimum
  - Shifts to right:
    - As time passes, cost decreases provided we get more
    - Fortunately we always had good ideas to use more:
      - Computers
      - Memory
      - Smartphones
      - Internet of Things?
- Why a minimum?
  - If too small, some don't work!

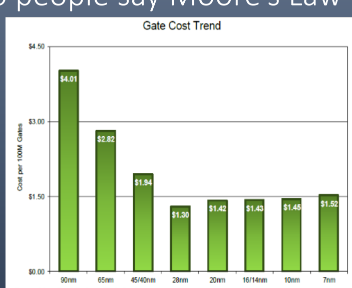
* Transistors: basic elements making up computers (see later)

Dr. Moore's Vision (in 1965)

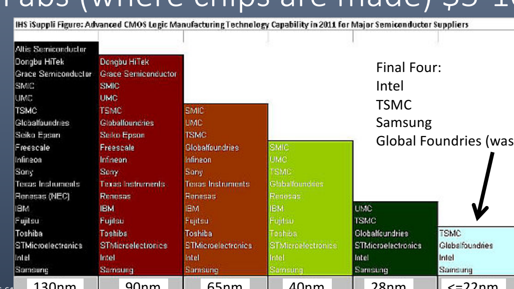


- Something useful that is getting always better and less expensive is good for
  - Society
  - Business

Why do people say Moore's Law is over?



Fabs (where chips are made) $5-10B



Final Four:
Intel
TSMC
Samsung
Global Foundries (was IBM)

Break!

Agenda

- Pointers in C
- Arrays in C
- This is not on the test
- **Pointer arithmetic**
- Strings, main
- And in Conclusion, …

Pointer Arithmetic - `char`

```
#include <stdio.h>

int main(void) {
    char c[] = { 'a', 'b' };
    char *pc = c;
    pc++;
    printf("*pc=%c\n c=%p\npc=%p\npc-c=%ld\n",
            *pc, c, pc, pc-c);

    int i[] = { 10, 20 };
    int *pi = i;
    pi++;
    printf("*pi=%d\n i=%p\npi=%p\npi-i=%ld\n",
            *pi, i, pi, pi-i);
}
```

```
*pc  = b
c    = 0x7fff50f54b3e
pc   = 0x7fff50f54b3f
pc-c = 1
```

Type	Name	Byte Addr*	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

*Computer only uses byte addresses. Tables with blue headers are simplifications.

Pointer Arithmetic - `int`

```
#include <stdio.h>

int main(void) {
    char c[] = { 'a', 'b' };
    char *pc = c;
    pc++;
    printf("*pc=%c\n c=%p\npc=%p\npc-c=%ld\n",
            *pc, c, pc, pc-c);

    int i[] = { 10, 20 };
    int *pi = i;
    pi++;
    printf("*pi=%d\n i=%p\npi=%p\npi-i=%ld\n",
            *pi, i, pi, pi-i);
}
```

```
*pi  = 20
i    = 0x7fff50f54b40
pi   = 0x7fff50f54b44
pi-i = 1
```

Type	Name	Byte Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Array Name / Pointer Duality

- Array variable is a "pointer" to the first (0th) element
- Can use pointers to access array elements
  - `char *pstr` and `char astr[]` are nearly identical declarations
  - Differ in subtle ways: `astr++` is illegal
- Consequences:
  - `astr` is an array variable, but works like a pointer
  - `astr[0]` is the same as `*astr`
  - `astr[2]` is the same as `*(astr+2)`
  - Can use pointer arithmetic to access array elements

Arrays versus Pointer Example

```
#include <stdio.h>

int main(void) {
    // array indexing
    int a[] = { 10, 20, 30 };
    printf("a[1]=%d, *(p+1)=%d, p[2]=%d\n",
            a[1], *(a+1), *(&a[2]));
    // pointer arithmetic
    int *p = a;
    p++;
    *p = 22;
    p[1] = 33;
    p[-1] = 11;
    for (int i=0;  i<3;  i++)
        printf("a[%d] = %d, ", i, a[i]);
}
```

Type	Name	Addr	Value
		…	
		104	
		103	
		102	
		101	
		100	
		…	

Output:

```
a[1]=20, *(p+1)=20, p[2]=30
a[0]=11, a[1]=22, a[2]=33
```

Mixing pointer and array notation can be confusing → avoid.

Pointer Arithmetic

- Example:

```
int n = 3;
int *p;
p += n;     // adds n*sizeof(int) to p
p -= n;     // subtracts n*sizeof(int) from p
```

- Use only for arrays. **Never**:

```
char *p;
char a, b;
p = &a;
p += 1;     // may point to b, or not
```


Arrays and Pointers

- Array ≈ pointer to the initial (0th) array element

  `a[i]  ≡  *(a+i)`

- An array is passed to a function as a pointer
  - The array size (# of bytes) is lost!
- Usually bad style to interchange arrays and pointers

Passing arrays:

Really `int *array` — explicitly pass size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```


Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

… because `array` is really a pointer (and a pointer is architecture-dependent, but likely to be 8 on modern 64-bit machines!)

What does this print? **40**

(provided `sizeof(int)==4`)

Arrays and Pointers

These code sequences have the same effect:

```
int  i;
int  array[5];

for (i = 0; i < 5; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[5];

for (p = array; p < &array[5]; p++)
{
  *p = …;
}
```

Name	Type	Addr	Value
		…	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Point past end of array?

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

```
const int SZ = 10;
int ar[SZ], *p, *q, sum = 0;
p = &ar[0]; q = &ar[SZ];
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 error

Valid Pointer Arithmetic

- Add/subtract an integer to/from a pointer
- Difference of 2 pointers (must both point to elements in same array)
- Compare pointers (<, <=, ==, !=, >, >=)
- Compare pointer to NULL (indicates that the pointer points to nothing)

Everything makes no sense & is illegal:

- adding two pointers
- multiplying pointers
- subtract pointer from integer

Pointers to Pointers

```
#include <stdio.h>

// changes value of pointer
void next_el(int **h) {
    *h = *h + 1;
}

int main(void) {
    int A[] = { 10, 20, 30 };
    // p points to first element of A
    int *p = A;
    next_el(&p);
    // now p points to 2nd element of A
    printf("*p = %d\n", *p);
}
```


Your Turn …

```
int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);
```

Name	Type	Addr	Value
		…	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Answer	
RED	2
GREEN	3
ORANGE	4
YELLOW	5

Administrivia

- Homework 0 and Mini-bio will be released by tonight
- Lab swap policy is posted on Piazza and the website
- Guerrilla Session and mini-tutoring session details will be posted soon

Break!

Agenda

- Pointers in C
- Arrays in C
- This is not on the test
- Pointer arithmetic
- **Strings, main**
- And in Conclusion, …

C Strings

- C strings are null-terminated character arrays
  - `char s[] = "abc";`

Type	Name	Byte Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

String Example

```
#include <stdio.h>

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

int main(void) {
    char str[] = "abc";
    printf("str = %s,  length = %d\n", str, slen(str));
}
```

Output: str = abc, length = 3

Concise strlen()

```
int strlen(char *s) {
    char *p = s;
    while (*p++)
            ; /* Null body of while */
    return (p – s – 1);
}
```

What happens if there is no zero character at end of string?

Arguments in `main()`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[])`
  - `argc` is the *number* of strings on the command line
  - `argv` is a *pointer* to an array containing the arguments as strings

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i=0;  i<argc;  i++)
        printf("arg[%d] = %s\n", i, argv[i]);
}
```


Example

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i=0;  i<argc;  i++)
        printf("arg[%d] = %s\n", i, argv[i]);
}
```

UNIX:

```
$ gcc –o ex Argc.c
$ ./ex –g a "d e f"
arg[0] = ./ex
arg[1] = -g
arg[2] = a
arg[3] = d e f
```


Agenda

- Pointers in C
- Arrays in C
- This is not on the test
- Pointer arithmetic
- Strings, main
- **And in Conclusion, …**

And in Conclusion, …

- Pointers are "C speak" for machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type & size of the object they point to (except void *)
- Like most things, pointers can be used for
  - Pointers are powerful
  - But, without good planning, a major source of errors
  - Plenty of examples in the next lecture!