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);

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

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

Applications
- Generic functions e.g. to allocate memory
  - malloc, free
  - accept and return pointers of any type
  - see next lecture

What’s wrong with this Code?

# include <stdio.h>

int main(void)
int a = 3, b = -7;
int *p = &a, *q = &b;
p = q;
printf("a = %d, p = %p, q = %d, \n", a, p, q);
return 0;

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

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

Pointers to struct

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

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

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

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

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

Your Turn!

```c
#include <stdio.h>

int main(void)
int a = 3, b = -7;
int *p = &a, *q = &b;
p = q;
printf("a = %d, p = %p, q = %d, \n", a, p, q);
return 0;
```

```
<table>
<thead>
<tr>
<th>Answer</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>3</td>
<td>-7</td>
</tr>
<tr>
<td>GREEN</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>ORANGE</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>YELLOW</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
```

Pointers as Function Arguments

```c
#include <stdio.h>

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

```c
int main(void)
int a = 1, b = -3;
int *p, *q;
p = &a, *q = &b;
printf("a=%d b=%d\n", a, b);
```

```c
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1004</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1004</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1004</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1004</td>
<td>3</td>
</tr>
</tbody>
</table>
```
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[3], c[3];
    foo(a[0], b[0]);
    printf("Red is Int %d\n", a[0], b[0]);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>105</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>103</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

Agenda

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

C Arrays

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>105</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>103</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

Beware: no array bound checking!

```c
#include <stdio.h>

int main(void) {
    int a[3] = { 1, 2, 3};
    for (int i=0; i<4; i++)
        printf("%d\n", 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!

Use Constants, Not Literals

- Assign size to constant
  ```c
  int i, ar[10];
  for(j = 0; j < 10; j++)
      ...;
  ```
- Better pattern
  ```c
  const int ARRAY_SIZE = 10;
  for(i = 0; i < ARRAY_SIZE;
      for(i = 0; i < ARRAY_SIZE;
  ```

- "Single source of truth"
  ```c
  void* x;
  short* y;
  char* z;
  ```

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>105</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>103</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  ```c
  int x;
  char* z;
  ```
- 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
  ```c
  char *z;
  ```
- E.g., 32-bit integer stored in 4 consecutive 8-bit bytes
- DRY: "Don't Repeat Yourself!"
sizeof() operator

```c
#include <stdio.h>

int main(void) {
    double d;
    float f;
    printf("double: %lu\n", sizeof(d));
    printf("float: %lu\n", sizeof(f));
    return 0;
}
```

Output:
- double: 8
- float: 4

Size of operator
- Returns number of bytes in object
  - Number of bytes in a byte is not standardized
  - All modern computers have 8 bits per byte, e.g., Intel "all" computers use values, e.g., 8 bits per "byte"
- By definition, in C
  - `sizeof(char) == 1`
- For all other types result is hardware and compiler dependent
- Do not assume `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,

Pointers in C

Arrays in C

This is not on the test

Pointer Arithmetic

Arrays versus Pointer Example

Arrays versus Pointer Example

- Mixing pointer and array notation can be confusing → avoid.

Array Name / Pointer Duality

- Array variable is a "pointer" to the first (0) element
- Can use pointers to access array elements
  - char *astr 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

Type Name Add Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- Example:
  ```c
  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*:
  ```c
  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
  ```c
  a[i] = *(a+i)
  ```

- An array is passed to a function as a pointer
  ```c
  int main(void)
  ```
  ```c
  int a[10], b[5];
  ... foo(a, 10)...
  ```

- Usually bad style to interchange arrays and pointers

---

**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

---

**Arrays and Pointers**

These code sequences have the same effect:

```c
int foo(int array[],
   unsigned int size)
{
    ... printf("%d
", sizeof(array));
    ...
}
```
Pointers to Pointers

```c
#include <stdio.h>

// changes value of pointer
void next_el(int **eh) {
    *eh = *eh + 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("%d\n", *p);
    return 0;
}
```

Your Turn …

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

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!

C Strings

• C strings are null-terminated character arrays
  - char s[] = “abc”;

Agenda

• Pointers in C
• Arrays in C
• This is not on the test
• Pointer arithmetic
• Strings, main
• And in Conclusion, …
### String Example

```c
#include <stdio.h>

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

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

Output: \nL = abc, length = 3
```

### Concise strlen()

```c
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

```c
#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 Argv.c
$ ./ex -g a "d e f"
arg[0] = ./ex
arg[1] = -g
arg[2] = a
arg[3] = d e f
```

### Example

```c
#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 Argv.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!