Pointers, Pointers, Pointers
CSE 333 Autumn 2018

Instructor: Hal Perkins

Teaching Assistants:
Tarkan Al-Kazily Renshu Gu Travis McGaha
Harshita Neti Thai Pham Forrest Timour
Soumya Vasisht Yifan Xu
Administrivia

- Exercise 2 out today and due Wednesday morning

- Exercise grading
  - We will do our best to keep up (some delays as we start the qtr)
  - Things to watch for:
    - Input sanity check
    - No functional abstraction (single blob of code)
    - Formatting funnies (e.g. tabs instead of spaces)
  - Grades:
    - 3 = superior; 2 = fine, some things to improve; 1 = some problems; 0 = hmmm...
    - We expect 3 and 0 to be rare; more 3’s as quarter progresses
Adminstrivia

- Homework 0 due tonight, 11 pm

- Homework 1 out now, due in week and a half (Thu 10/11)
  - Everyone should have pulled starter code by now. Any surprises?

- If you don’t yet have a gradescope account and/or gitlab repo, send mail to cse333-staff@cs right after class with details so we can get that fixed
Administrivia

- Documentation:
  - man pages, books
  - Reference websites: cplusplus.org, man7.org, gcc.gnu.org, etc.

- Folklore:
  - Google-ing, stackoverflow, that rando in lab

- Tradeoffs? Relative strengths & weaknesses?
  - Discuss
Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return 0;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {  
    int x = 1;  
    int arr[3] = {2, 3, 4};  
    int* p = &arr[1];  

    printf("&x: %p; x: %d\n", &x, x);  
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);  
    printf("&arr[1]: %p; arr[1]: %d\n", &arr[1], arr[1]);  
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);  
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);  

    return 0;  
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;x</td>
<td></td>
<td>value</td>
</tr>
<tr>
<td>&amp;arr[0]</td>
<td></td>
<td>value</td>
</tr>
<tr>
<td>&amp;arr[1]</td>
<td></td>
<td>value</td>
</tr>
<tr>
<td>&amp;arr[2]</td>
<td></td>
<td>value</td>
</tr>
<tr>
<td>&amp;p</td>
<td></td>
<td>value</td>
</tr>
<tr>
<td>&amp;x</td>
<td></td>
<td>value</td>
</tr>
</tbody>
</table>

stack frame for `main()`
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p;  *p: %d\n", &p, p, *p);

    return 0;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;arr[0]</td>
<td>arr[0]</td>
<td>2</td>
</tr>
<tr>
<td>&amp;p</td>
<td>p</td>
<td>&amp;arr[1]</td>
</tr>
<tr>
<td>&amp;x</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
Box-and-Arrow Diagrams

```c
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p;  x: %d\n", &x, x);
    printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p;  p: %p;  *p: %d\n", &p, p, *p);

    return 0;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff...78</td>
<td>arr[2]</td>
<td>4</td>
</tr>
<tr>
<td>0x7fff...74</td>
<td>arr[1]</td>
<td>3</td>
</tr>
<tr>
<td>0x7fff...70</td>
<td>arr[0]</td>
<td>2</td>
</tr>
<tr>
<td>0x7fff...68</td>
<td>p</td>
<td>0x7fff...74</td>
</tr>
<tr>
<td>0x7fff...64</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
Pointer Arithmetic

- Pointers are *typed*
  - Tells the compiler the size of the data you are pointing to
  - *Exception*: `void*` is a generic pointer (*i.e.* a placeholder)

- Pointer arithmetic is scaled by `sizeof(*p)`
  - Works nicely for arrays
  - Does not work on `void*`, since `void` doesn’t have a size!

- Valid pointer arithmetic:
  - Add/subtract an integer to a pointer
  - Subtract two pointers (within stack frame or malloc block)
  - Compare pointers (`<`, `<=`, `==`, `!=`, `>`, `>=`), including `NULL`
Practice Question

At this point in the code, what values are stored in `arr[]`?

```
int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p;  // pointer to a pointer

    (*dp) += 1;
    p += 1;
    (*dp) += 1;

    return 0;
}
```
Practice Solution

```c
int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p; // pointer to a pointer

    *(*dp) += 1;
    p += 1;
    *(*dp) += 1;

    return 0;
}
```

Note: arrow points to next instruction to be executed.

`boxarrow2.c`
Practice Solution

```c
int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p;  // pointer to a pointer

    (*dp) += 1;
    p += 1;
    (*dp) += 1;

    return 0;
}
```

Note: arrow points to next instruction to be executed.
Practice Solution

```c
int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p;  // pointer to a pointer

    (*dp) += 1;
    p += 1;
    (*dp) += 1;

    return 0;
}
```

Note: arrow points to next instruction to be executed.
Practice Solution

```c
int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p;  // pointer to a pointer

    (*dp) += 1;
    p += 1;
    (*dp) += 1;

    return 0;
}
```

Note: arrow points to next instruction to be executed.

boxarrow2.c
Endianness

- Memory is byte-addressed, so endianness determines what ordering that multi-byte data gets read and stored in memory
  - **Big-endian**: Least significant byte has *highest* address
  - **Little-endian**: Least significant byte has *lowest* address

- **Example**: 4-byte data 0xa1b2c3d4 at address 0x100

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a1</td>
<td>b2</td>
<td>c3</td>
<td>d4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little-Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d4</td>
<td>c3</td>
<td>b2</td>
<td>a1</td>
</tr>
</tbody>
</table>
Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

Note: Arrow points to next instruction.

Stack (assume x86-64)
Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;
    
    int_ptr += 1;
    int_ptr += 2;  // uh oh
    char_ptr += 1;
    char_ptr += 2;
    return 0;
}
```

Note: Arrow points to next instruction.

Stack (assume x86-64)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[2]</td>
<td>03 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[1]</td>
<td>02 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[0]</td>
<td>01 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char_ptr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int_ptr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

*pointerarithmetic.c*

**Stack** (assume x86-64)

- `arr[0]`: 01 00 00 00 00
- `arr[1]`: 02 00 00 00 00
- `arr[2]`: 03 00 00 00 00

**Note:** Arrow points to *next* instruction.

- `int_ptr`: Stack Pointer
- `char_ptr`: Pointer to `arr[0]`
Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

Note: Arrow points to next instruction.

Stack (assume x86-64)
**Pointer Arithmetic Example**

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

**Stack** (assume x86-64)

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[2]</td>
<td>03 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[1]</td>
<td>02 00 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[0]</td>
<td>01 00 00 00 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Arrow points to next instruction.

**pointerarithmetics.c**

- **int_ptr**: 0x07fffffffde010
- ***int_ptr**: 1
Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

Stack (assume x86-64)

```
arr[0]: 01 00 00 00 00
arr[1]: 02 00 00 00 00
arr[2]: 03 00 00 00 00
char_ptr: 01 00 00 00 00
int_ptr: 0x07ffffffde014
```

Note: Arrow points to next instruction.
Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

Note: Arrow points to next instruction.

Stack (assume x86-64)

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>int_ptr</th>
<th>*int_ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[2]</td>
<td>03 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>arr[1]</td>
<td>02 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>arr[0]</td>
<td>01 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>char_ptr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int_ptr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

int_ptr: 0x07fffffffd01C
*int_ptr: ???
# Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

`pointerarithmetic.c`

**Stack** (assume x86-64)

```
arr[2] 03 00 00 00 00
arr[1] 02 00 00 00 00
arr[0] 01 00 00 00 00
char_ptr
int_ptr
```

**Note**: Arrow points to next instruction.

**char_ptr**: 0x0x7fffffffffde010

***char_ptr***: 1
### Pointer Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

**Stack**
(assume x86-64)

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[2]</td>
<td>03</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>arr[1]</td>
<td>02</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>arr[0]</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```

**char_ptr**: 0x0x7fffffffd01

**int_ptr**: 0

**Note**: Arrow points to next instruction.
Pointers Arithmetic Example

```c
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

Note: Arrow points to next instruction.

Stack (assume x86-64)

- `arr[2]`: 03 00 00 00
- `arr[1]`: 02 00 00 00
- `arr[0]`: 01 00 00 00

char_ptr: 0x07fffffffe013
*char_ptr: 0
Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
C is Call-By-Value

- C (and Java) pass arguments by *value*

  - Callee receives a **local copy** of the argument
    - Register or Stack
  - If the callee modifies a parameter, the caller’s copy *isn’t* modified

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
```
Broken Swap

brokenswap.c

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
}
```

Note: Arrow points to next instruction.
Broken Swap

brokenswap.c

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
}
```
Broken Swap

void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...

brokenswap.c
Broken Swap

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
}
```

brokenswap.c

```
OS kernel [protected]

Stack

main | a | 42 | b | -7 |

swap | a | 42 | b | -7 |

tmp | 42 |

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata
```
**Broken Swap**

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
}
```

[Diagram showing memory allocation and swap function operation]

**brokenswap.c**

- **OS kernel [protected]**
  - Stack
  - Main: a 42, b -7
  - Swap: a -7, b -7
  - Heap
  - Read/Write Segment
    - .data, .bss
  - Read-Only Segment
    - .text, .rodata
Broken Swap

void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...

brokenswap.c
Broken Swap

brokenswap.c

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
}
```

OS kernel [protected]
Stack
main a 42 b -7
Heap
Read/Write Segment .data, .bss
Read-Only Segment .text, .rodata
Faking Call-By-Reference in C

- Can use pointers to *approximate* call-by-reference
  - Callee still receives a *copy* of the pointer (*i.e.* call-by-value), but it can modify something in the caller’s scope by dereferencing the pointer parameter

```c
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...
```
Fixed Swap

```c
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...
}
```

**Note:** Arrow points to next instruction.

OS kernel [protected]

Stack

- main
  - a 42
  - b -7

Heap

Read/Write Segment
- .data, .bss

Read-Only Segment
- .text, .rodata
Fixed Swap

swap.c

```c
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...}
```

OS kernel [protected]

Stack

main

```
<table>
<thead>
<tr>
<th>a</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>-7</td>
</tr>
</tbody>
</table>
```

Heap

swap

```
| a  | 42 |
| b  | -7 |
```

Heap

Read/Write Segment

```
.data, .bss
```

Read-Only Segment

```
.text, .rodata
```

...
Fixed Swap

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...}
```

OS kernel [protected]

Stack

Stack Frame:
- `a`: 42
- `b`: -7

Heap

Heap Frame:
- `a`
- `b`

Read/Write Segment
- `.data`, `.bss`

Read-Only Segment
- `.text`, `.rodata`
Fixed Swap

```c
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...;
}
```
Fixed Swap

swap.c

```c
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...
```
Fixed Swap

void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;  // Swap a and b
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...
}

[OS kernel diagram]
- Stack
- Heap
- Read/Write Segment (.data, .bss)
- Read-Only Segment (.text, .rodata)
- main: a -7, b 42
Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- **Pointers and Arrays**
- Function Pointers
Pointers and Arrays

- A pointer can point to an array element
  - You can use array indexing notation on pointers
    - `ptr[i]` is `*(ptr+i)` with pointer arithmetic – reference the data `i` elements forward from `ptr`
  - An array name’s value is the beginning address of the array
    - *Like* a pointer to the first element of array, but can’t change

```c
int a[] = {10, 20, 30, 40, 50};
int* p1 = &a[3];  // refers to a's 4th element
int* p2 = &a[0];  // refers to a's 1st element
int* p3 = a;     // refers to a's 1st element

*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;  // final: 200, 400, 500, 100, 300
```
Array Parameters

- Array parameters are *actually* passed as pointers to the first array element
  - The [] syntax for parameter types is just for convenience
    - OK to use whichever best helps the reader

This code:

```c
void f(int a[]);

int main( ... ) {
    int a[5];
    ...
    f(a);
    return 0;
}
```

Equivalent to:

```c
void f(int* a);  

int main( ... ) {
    int a[5];
    ...
    f(&a[0]);
    return 0;
}
```
Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
Function Pointers

- Based on what you know about assembly, what is a function name, really?
  - Can use pointers that store addresses of functions!

- Generic format:
  - Looks like a function prototype with extra * in front of name
  - Why are parentheses around (* name) needed?

- Using the function:
  - Calls the pointed-to function with the given arguments and return the return value
Function Pointer Example

- `map()` performs operation on each element of an array.

```c
#define LEN 4

int negate(int num) { return -num; }
int square(int num) { return num * num; }

// perform operation pointed to on each array element
void map(int a[], int len, int (*op)(int n)) {
    for (int i = 0; i < len; i++) {
        a[i] = (*op)(a[i]);  // dereference function pointer
    }
}

int main(int argc, char** argv) {
    int arr[LEN] = {-1, 0, 1, 2};
    int (*op)(int n);  // function pointer called 'op'
    op = square;    // function name returns addr (like array)
    map(arr, LEN, op);
    ...
}
```
Extra Exercise #1

Use a box-and-arrow diagram for the following program and explain what it prints out:

```c
#include <stdio.h>

int foo(int* bar, int** baz) {
    *bar = 5;
    *(bar+1) = 6;
    *baz = bar + 2;
    return *((*baz)+1);
}

int main(int argc, char** argv) {
    int arr[4] = {1, 2, 3, 4};
    int* ptr;

    arr[0] = foo(&arr[0], &ptr);
    printf("%d %d %d %d %d
",
            arr[0], arr[1], arr[2], arr[3], *ptr);
    return 0;
}
```
Extra Exercise #2

- Write a program that determines and prints out whether the computer it is running on is little-endian or big-endian.

  **Hint:** pointerarithmetic.c from today’s lecture or show_bytes.c from 351
Extra Exercise #3

- Write a function that:
  - Malloc’s an int* array of the same element length
  - Initializes each element of the newly-allocated array to point to the corresponding element of the passed-in array
  - Returns a pointer to the newly-allocated array
Extra Exercise #4

- Write a function that:
  - Accepts a function pointer and an integer as arguments
  - Invokes the pointed-to function with the integer as its argument