Pointers, Pointers, Pointers ...
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi              Farrell Fileas        Lukas Joswiak
Nathan Lipiarski    Renshu Gu           Travis McGaha
Yibo Cao            Yifan Bai           Yifan Xu

“Pointers are merely variables that contain memory addresses”
About how long did Exercise 1 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn’t finish / I prefer not to say
Administrivia (1 of 2)

- Exercise 2 out today and due Wednesday morning

- Exercise grading
  - We will do our best to keep up
  - Things to watch for:
    - Input sanity check
    - No functional abstraction (single blob of code)
    - Formatting funnies (e.g. tabs instead of spaces)
Administrivia (2 of 2)

❖ Homework 0 due TONIGHT
  ▪ Logistics and infrastructure for projects
    • clint and valgrind are useful for exercises, too

❖ Homework 1 already out, due in 2 weeks (Thu 10/10)
  ▪ Linked list and hash table implementations in C
  ▪ Get starter code using git pull in your course repo
    • Might have a merge if your local copy has unpushed changes
    • If git drops you into vim, :q to quit or :wq if you want to save changes
Lecture Outline

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

“Pointers are merely variables that contain memory addresses”
Box-and-Arrow Diagrams

int main(int argc, char **argv) {
    int32_t x = 1;
    int32_t arr[3] = {2, 3, 4};
    int32_t *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) {
    int32_t x = 1;
    int32_t arr[3] = {2, 3, 4};
    int32_t *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></td>
<td>x</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>arr[2]</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>arr[1]</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>arr[0]</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>value</td>
</tr>
</tbody>
</table>
Box-and-Arrow Diagrams

int main(int argc, char **argv) {
    int32_t x = 1;
    int32_t arr[3] = {2, 3, 4};
    int32_t *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>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;x</td>
<td>x</td>
</tr>
<tr>
<td>&amp;arr[0]</td>
<td>arr[0]</td>
</tr>
<tr>
<td>&amp;p</td>
<td>p</td>
</tr>
</tbody>
</table>
int main(int argc, char **argv) {
    int32_t x = 1;
    int32_t arr[3] = {2, 3, 4};
    int32_t *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;
}

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>arr[2]</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>arr[1]</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>arr[0]</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0x7fff...d4</td>
<td></td>
</tr>
</tbody>
</table>

address | name | value
--------|------|------
0x7fff...dc |       |      
0x7fff...d8 |       |      
0x7fff...d4 |       |      
0x7fff...d0 |       |      
0x7fff...c8 |       |      

Box-and-Arrow Diagrams

boxarrow.c
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/from a pointer
  ▪ Subtract two pointers (within stack frame or malloc block)
  ▪ Compare pointers (`<, <=, ==, !=, >, >=`), including `NULL`
  ▪ ... plenty of valid-but-inadvisable operations, too!
Inadvisable Pointer Examples 🙄

// Leave them uninitialized!
int *int_ptr;
*int_ptr = 333;

// Use garbage values!
int *int_ptr = rand();
*int_ptr = 333;

// Reinterpret raw bytes!
double d = 3.14;
int *int_ptr = (int *) &d;
*int_ptr = 333;

[inadvisable_pointers.c]

“Pointers are merely variables that contain memory addresses”
Inadvisable Pointer-Specific Examples

// Uninitialized!
int ***ipp;
***ipp = 333;

// Garbage values!
ipp = rand();
***ipp = 333;

// Reinterpret raw bytes!
double d = 3.14;
double *dp = &d;
ipp = (int **) &dp;
*ip = 333;

void *vp = (void*) ip;
void **vpp = &vp;

vpp = vp; // lol typechecking

// Since pointers are variables, we can do all these things recursively!

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

A. {2, 3, 4}
B. {3, 4, 5}
C. {2, 6, 4}
D. {2, 4, 5}
E. I'm not sure ...
PollEverywhere Solution

```c
int main(int argc, char **argv) {
    int32_t arr[3] = {2, 3, 4};
    int32_t *p = &arr[1];
    int32_t **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

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff...70</td>
<td>arr[0]</td>
<td>2</td>
</tr>
<tr>
<td>0x7fff...74</td>
<td>arr[1]</td>
<td>3</td>
</tr>
<tr>
<td>0x7fff...78</td>
<td>arr[2]</td>
<td>4</td>
</tr>
<tr>
<td>0x7fff...68</td>
<td>p</td>
<td>0x7fff...74</td>
</tr>
<tr>
<td>0x7fff...60</td>
<td>dp</td>
<td>0x7fff...68</td>
</tr>
</tbody>
</table>

address | name | value |
---------|------|-------|
0x7fff...70 | arr[0] | 2     |
0x7fff...74 | arr[1] | 3     |
0x7fff...78 | arr[2] | 4     |
0x7fff...68 | p     | 0x7fff...74 |
0x7fff...60 | dp    | 0x7fff...68 |
Poll Everywhere Solution

```c
int main(int argc, char **argv) {
    int32_t arr[3] = {2, 3, 4};
    int32_t *p = &arr[1];
    int32_t **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

<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>4</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...60</td>
<td>dp</td>
<td>0x7fff...68</td>
</tr>
</tbody>
</table>
PollEverywhere Solution

```c
int main(int argc, char **argv) {
    int32_t arr[3] = {2, 3, 4};
    int32_t *p = &arr[1];
    int32_t **dp = &p; // pointer to a pointer

    *(*dp) += 1;
    p += 1;
    *(*dp) += 1;
    return 0;
}
```

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

```c
int main(int argc, char **argv) {
    int32_t arr[3] = {2, 3, 4};
    int32_t *p = &arr[1];
    int32_t **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

Big-Endian

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

Little-Endian

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<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; // uh oh
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

Note: Arrow points to next instruction.

```
Stack (assume x86-64)

+0 +1 +2 +3

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

`pointerarithmetic.c`
# 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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[0]</td>
<td>01 00 00 00 00</td>
</tr>
<tr>
<td>arr[1]</td>
<td>02 00 00 00 00</td>
</tr>
<tr>
<td>arr[2]</td>
<td>03 00 00 00 00</td>
</tr>
</tbody>
</table>

`pointerarithmetic.c`
**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>
</tr>
</thead>
<tbody>
<tr>
<td>arr[2]</td>
<td>03 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[1]</td>
<td>02 00 00 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[0]</td>
<td>01 00 00 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`pointerarithmetic.c`
### 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)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03 00 00 00 00</td>
<td>02 00 00 00 00</td>
<td>01 00 00 00 00</td>
</tr>
</tbody>
</table>

**pointerarithmetic.c**

<table>
<thead>
<tr>
<th>int_ptr:</th>
<th>0x0x7fffffffde010</th>
</tr>
</thead>
<tbody>
<tr>
<td>*int_ptr:</td>
<td>1</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)

```
+0 +1 +2 +3
```

`int_ptr`: `0x0x7fffffffde014`

`*int_ptr`: `2`
**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[2] 03 00 00 00 00
arr[1] 02 00 00 00 00
arr[0] 01 00 00 00 00
char_ptr
int_ptr +0 +1 +2 +3
```

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

**pointerarithmetic.c**

- `int_ptr`: 0x0x7fffffffde01C
- `*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`

<table>
<thead>
<tr>
<th>char_ptr:</th>
<th>0x0x7fffffffde010</th>
</tr>
</thead>
<tbody>
<tr>
<td>*char_ptr:</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

```
Pointer arithmetic.c

char_ptr: 0x0x7fffffffde01
*char_ptr: 0
```

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
```

Stack: +0 +1 +2 +3

`int_ptr` Stack: +0 +1 +2 +3

`char_ptr`: +0 +1 +2 +3
### 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)**

```
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
```

---

- **char_ptr**: 0x0x7fffffffde013
- ***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

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.

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
- Heap
- Read/Write Segments .data, .bss
- Read-Only Segments .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);
    ...
}
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 |
| swap | a 42 | b -7 |
|      | tmp 42 |

Heap

Read/Write Segments

.data, .bss

Read-Only Segments

.text, .rodata

brokenswap.c
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

<table>
<thead>
<tr>
<th>OS kernel [protected]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>main</td>
</tr>
<tr>
<td>a  42</td>
</tr>
<tr>
<td>b  -7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap</td>
</tr>
<tr>
<td>a  -7</td>
</tr>
<tr>
<td>b  42</td>
</tr>
<tr>
<td>tmp  42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write Segments</td>
</tr>
<tr>
<td>.data, .bss</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Only Segments</td>
</tr>
<tr>
<td>.text, .rodata</td>
</tr>
</tbody>
</table>
Broken Swap

brokenswap.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);
    ...

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.
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);
    ...
```

OS kernel [protected]

Stack

- main
  - a: 42
  - b: -7

Swap

- a
- b
- tmp: ??

Heap

Read/Write Segments

- .data
- .bss

Read-Only Segments

- .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

```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 -7 b -7

Swap
a b

tmp 42

Heap

Read/Write Segments .data, .bss

Read-Only Segments .text, .rodata
```
Fixed 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);
    ...

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);
    ...
}
```

Diagram:
- OS kernel [protected]
- Stack
  - main
    - a: -7
    - b: 42
- Heap
- Read/Write Segments
  - .data, .bss
- Read-Only Segments
  - .text, .rodata
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
    • `a[i]` is `*(a+i)` with pointer arithmetic – reference the data `i` elements forward from `a`
  ▪ 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 it

```
int a[] = {10, 20, 30, 40, 50};
int32_t *p1 = &a[3];  // refers to a's 4th element
int32_t *p2 = &a[0];  // refers to a's 1st element
int32_t *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
```
Pointers and Arrays

```c
int a[] = {10, 20, 30, 40, 50};
int32_t *p1 = &a[3]; // refers to a's 4th element
int32_t *p2 = &a[0]; // refers to a's 1st element
int32_t *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
```

Note: there is no variable "a" (no memory associated with the label "a") just its elements
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:

```
returnType (* name)(type1, ..., typeN)
```

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

❖ Using the function:

```
(*name)(arg1, ..., argN)
```

  ▪ 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);
    ...
}
```

*map.c*
Lecture Outline

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

“Pointers are merely variables that contain memory addresses”
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
", 
            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