Pointers, Pointers, Pointers
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Justin Tysdal
Sayuj Shahi
Nicholas Batchelder
Leanna Mi Nguyen
Administrivia (1)

- Exercise 2 out today!
  - Due Wednesday @ 10 am
- Homework 0 due tonight @ 11 pm
  - You should be pretty far along by now
  - Went over gitlab setup in sections on Thursday
Administrivia (2)

❖ HW1 was pushed to repos on Friday
  ▪ Linked list and hash table implementations in C
  ▪ Download starter code using `git pull` in your course repo
    • Fine to do this before submitting hw0, just will have to deal with some basic merging
    • Might have “merge conflict” if your local repo has unpushed changes
      – If your changes are all in commits, default git merge handling will almost certainly do the right thing
      – To avoid, always do a `git pull` before any git commit or push
      – If you have *uncommitted* changes, do some research on git “stashes”
  
  ▪ If you haven’t already, start reading the assignment and looking at the code now!
    • For large projects, you want to pace yourself so if something baffling happens, you can let it go for the day and come back to it tomorrow
Administrivia (3)

❖ Exercise grading

❖ Score is an overall evaluation: 3/2/1/0 = superior / good / marginal / not sufficient for credit
  - We expect lots of 2’s and 3’s at first, more 3’s on later exercises

❖ Scores are NOT proportional to grade: a 2 isn’t 70%, it’s pretty good.

❖ Then additional ±0 rubric items as needed
  - These are a quick way of communicating “why” – reasons for deductions or comments about your solution
  - Allows us to be more consistent in feedback
  - The ±0 “score” is just because that’s how we have to use Gradescope to handle feedback notes – it does not contribute to “the points”
Lecture Outline

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

Valid pointer arithmetic:

- Add/subtract an integer and a pointer
- Subtract two pointers (within same stack frame or malloc block)
- Compare pointers (`<`, `<=`, `==`, `!=`, `>`, `>=`), including `NULL`

Pointer arithmetic is scaled by `sizeof(*p)`

- Works nicely for arrays
- Does not work on `void*`, since `void` doesn’t have a size!
  - Acts like unscaled integer arithmetic
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 EXIT_SUCCESS;
}
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 EXIT_SUCCESS;
}
```

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 EXIT_SUCCESS;
}
```

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 EXIT_SUCCESS;
}
```

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...78</td>
</tr>
<tr>
<td>0x7fff...60</td>
<td>dp</td>
<td>0x7fff...68</td>
</tr>
</tbody>
</table>
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 EXIT_SUCCESS;
}
```

Note: arrow points to next instruction to be executed.
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 EXIT_SUCCESS;
}
How are multi-byte values stored in memory?

- Memory is byte-addressed, so we need to determine how consecutive bytes combine into a single value.

- Endianness determines what ordering that multi-byte data gets read and stored.
  - **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**:
  - 0x100: `a1`
  - 0x101: `b2`
  - 0x102: `c3`
  - 0x103: `d4`

- **Little-Endian**:
  - 0x100: `d4`
  - 0x101: `c3`
  - 0x102: `b2`
  - 0x103: `a1`
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 = 4;
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

`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 = 4;
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

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[2]</td>
<td>03 00 00 00 00</td>
</tr>
<tr>
<td>arr[1]</td>
<td>02 00 00 00 00</td>
</tr>
<tr>
<td>arr[0]</td>
<td>01 00 00 00 00</td>
</tr>
<tr>
<td>char_ptr</td>
<td></td>
</tr>
<tr>
<td>int_ptr</td>
<td></td>
</tr>
</tbody>
</table>
```

pointerarithmet.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 = 4;
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

**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</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>arr[1]</td>
<td>02</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>arr[0]</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</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 = 4;
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

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 = 4;
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

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: 0x0x7fffffffe010
*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 = 4;
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

Note: Arrow points to next instruction.

**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

**Pointer Values:**
- `int_ptr`: 0x7fffffffd014
- `*int_ptr`: 2
Pointer Arithmetic Example

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 = 4;
    int_ptr += 2;
    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}

Stack (assume x86-64)

Note: Arrow points to next instruction.

int_ptr: 0x0x7ffffffde014
*int_ptr: 4
**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 = 4;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

Stack (assume x86-64)

```
arr[2] 03 00 00 00 00
arr[1] 04 00 00 00 00
arr[0] 01 00 00 00 00
char_ptr
int_ptr
```

Note: Arrow points to next instruction.

**pointerarithmetic.c**

```c
int_ptr: 0x0x7ffffffde01C
*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 = 4;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

Stack (assume x86-64)

- `arr[2]`: 03 00 00 00 00
- `arr[1]`: 04 00 00 00 00
- `arr[0]`: 01 00 00 00 00

- `char_ptr: 0x0x7ffffffde010`
- `*char_ptr: 1`

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 = 4;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

**Stack** (assume x86-64)

Note: Arrow points to next instruction.

<table>
<thead>
<tr>
<th>char_ptr:</th>
<th>0x0x7fffffffde011</th>
</tr>
</thead>
<tbody>
<tr>
<td>*char_ptr:</td>
<td>0</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 = 4;
    int_ptr += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 3;

    return EXIT_SUCCESS;
}
```

**Stack** (assume x86-64)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[2]</td>
<td>03 00 00 00 00</td>
</tr>
<tr>
<td>arr[1]</td>
<td>04 00 00 00</td>
</tr>
<tr>
<td>arr[0]</td>
<td>04 00 00 00</td>
</tr>
<tr>
<td>char_ptr</td>
<td></td>
</tr>
<tr>
<td>int_ptr</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
char_ptr: 0x0x7fffffffe014
*char_ptr: 4
```
This code does not swap its parameter values – why?

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);
    ...
```
C parameters are 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);
    ...
}
```

![Diagram of OS kernel with stack and heap segments]
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]

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>tmp</td>
</tr>
</tbody>
</table>

| Heap |

| Read/Write Segment |
| .data, .bss |

| Read-Only Segment |
| .text, .rodata |
## 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);
    ...
}
```

![Memory Diagram]

- **OS kernel [protected]**
  - Stack
  - Stack Frame:
    - `main`: `a: 42, b: -7`
  - Heap
    - Pointer `tmp` pointing to `42`
  - Read/Write Segment
    - `.data, .bss`
  - Read-Only Segment
    - `.text, .rodata`
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

```plaintext
main
a 42 b -7
```

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata
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:

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata
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.
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);
    ...
}
```

The `swap` function takes two pointers to integers, swaps their values, and the `main` function calls `swap` with pointers to `a` and `b`. The values of `a` and `b` are then printed as output.
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

Heap

Swap

Heap | a 42 | b -7

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

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);
    ...

swapped.c
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);
    ...
}
```
Output Parameters

❖ Output parameter

- A pointer parameter used to store (via dereference) a function output value *outside* of the function’s stack frame
- Typically points to/modifies something in the **Caller**’s scope
- Useful if you want to return arrays, or have multiple return values

❖ Setup and usage:

1) **Caller** creates space for the data (*e.g.*, `type var;`)
2) **Caller** passes a pointer to that space to **Callee** (*e.g.*, `&var`)
3) **Callee** has an output parameter (*e.g.*, `type* outparam`)
4) **Callee** uses parameter to store data in space provided by caller (*e.g.*, `*outparam = value;`)
5) **Caller** accesses output via modified data (*e.g.*, `var`)

**Warning:** Misuse of output parameters is *the* largest cause of errors in this course!
Lecture Outline

❖ Pointers & Pointer Arithmetic
❖ Pointers as Parameters
❖ **Pointers and Arrays**
❖ Function Pointers
❖ Heap-allocated Memory
  ▪ `malloc()` and `free()`
  ▪ Memory leaks
❖ `structs` and `typedef`
Pointers and Arrays

- A pointer can point to an array element
  - You can use array indexing notation on all pointers
    - \( \text{ptr}[i] \) is the same as \( *(\text{ptr}+i) \) - reference the data \( i \) elements forward from \( \text{ptr} \)
  - An array name’s value is the beginning address of the array
    - Like a pointer to the first element of array

```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;
```
Pointers and Arrays: Trace

```
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;
```
Pointers and Arrays: Trace

```
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;
```

Note: Arrow points to next instruction.
Pointers and Arrays: Trace

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

\[
*p1 = 100;
\]
\[
*p2 = 200;
\]
\[
p1[1] = 300;
\]
\[
p2[1] = 400;
\]
\[
p3[2] = 500;
\]

\[\text{Note: Arrow points to next instruction.}\]
Pointers and Arrays: Trace

```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;
```

Note: Arrow points to next instruction.
Pointers and Arrays: Trace

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;

Note: Arrow points to next instruction.
Pointers and Arrays: Trace

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

*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;

Note: Arrow points to next instruction.
Pointers and Arrays: Trace

```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;
```

Note: Arrow points to next instruction.
Pointers and Arrays: Trace

```
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;
```

Note: Arrow points to next instruction.
Pointers and Arrays: Trace

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;

Note: Arrow points to next instruction.
Array Parameters

- Array parameters are *actually* passed (by value) 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
❖ Heap-allocated Memory
  ▪ malloc() and free()
  ▪ Memory leaks
❖ structs and typedef
Function Pointers

❖ Based on what you know about assembly, what is a function name, really?
  ▪ Just an address of code!
  ▪ 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 (but * is optional since all you can do is call it!)
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 (*map_op)(int n)) {
    for (int i = 0; i < len; i++) {
        a[i] = (*map_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`
Function Pointer Example

- C allows you to omit & on a function parameter and omit * when calling pointed-to function; both assumed implicitly.

```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};
    map(arr, LEN, square);
    ...
```

- implicit funcptr dereference (no * needed)
- no & needed for func ptr argument
That’s it! Don’t forget:

- Exercise 2 out today, due Wednesday @ 10 am
- Homework 0 due tonight @ 11 pm
- Homework 1 out now, due next week.
  - Get started now, so there’s lots of time to ask questions and get help!
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\n",
            arr[0], arr[1], arr[2], arr[3], *ptr);
    return EXIT_SUCCESS;
}
```
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