Pointers, Pointers, Pointers
CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:
Aaron Johnston    Andrew Hu    Daniel Snitkovskiy
Forrest Timour    Kevin Bi     Kory Watson
Pat Kosakanchit   Renshu Gu    Tarkan Al-Kazily
Travis McGaha
Administrivia

- Exercise 2 out today and due Monday 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)
  - Grades:
    - Correctness [0-3], Tools Check [0-2], Style [0-3]
Administrivia

- Homework 0 due Monday
  - Logistics and infrastructure for projects
    - `clint` and `valgrind` are useful for exercises, too
  - Should have set up an SSH key and cloned GitLab repo by now
    - Do this ASAP so we have time to fix things if necessary

- Homework 1 out later today, due in 2 weeks (Thu 4/18)
  - Linked list and hash table implementations in C
  - Get starter code using `git pull` in your course repo
    - Might have “merge conflict” if your local copy has unpushed changes
      - If git drops you into vi(m), `:q` to quit or `:wq` if you want to save changes
Administrivia

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

- **Folklore:**
  - Google-ing, Stack Overflow, 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("%p\n", &x, x);
    printf("%p\n", &arr[0], arr[0]);
    printf("%p\n", &arr[2], arr[2]);
    printf("%p\n", &p, p, *p);

    return 0;
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
</table>

The variables `x`, `arr[0]`, and `arr[2]` are stored in the stack, while the pointer `p` points to the elements of the array `arr` which are located on the stack.
Box-and-Arrow Diagrams

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;
}
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;x</td>
<td>x</td>
<td>1</td>
</tr>
<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>
</tbody>
</table>
Box-and-Arrow Diagrams

```
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...4c</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>0x7fff...48</td>
<td>arr[2]</td>
<td>4</td>
</tr>
<tr>
<td>0x7fff...44</td>
<td>arr[1]</td>
<td>3</td>
</tr>
<tr>
<td>0x7fff...40</td>
<td>arr[0]</td>
<td>2</td>
</tr>
<tr>
<td>0x7fff...38</td>
<td>p</td>
<td>0x7fff...74</td>
</tr>
</tbody>
</table>

- `p`: get addr (follow arrow)
- `*p`: get data at addr

- `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`
At this point in the code, what values are stored in arr[]?
```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;  // follow arrow, then follow that arrow
    p += 1;
    (*dp) += 1;
    return 0;
}
```

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

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.
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.
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
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
**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*
```
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`
### 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`
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` pointing to `arr[0]`
- `int_ptr` pointing to `arr[0]`

**Variables:***

- `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;
}
```

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
```

int_ptr: 0x0x7fffffffdde014
*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  |               |
```

**Notes:**
- Arrow points to next instruction.
- Pointer arithmetic example shown.

**Variables:**
- `int_ptr`: 0x007fffffffde01
- `*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;
}
```

*char_ptr*: 0x0x7fffdde010

*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[2]  03 00 00 00 00
arr[1]  02 00 00 00 00
arr[0]  01 00 00 00 00
```

- `char_ptr`: 0x07ffffffde01
- `*char_ptr`: 0

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)

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

`char_ptr`: 0x0x7fffffffdede013

`*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.
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

```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
- Heap
  - swap: a 42, b -7
  - tmp: ??

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

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

The code provided is a broken swap function and a main function. The swap function swaps the values of two integers, but it does not work correctly.

```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);
    ...
}
```

The OS kernel is depicted with various segments, including the stack and heap. The stack contains variables `a` and `b`, and the heap contains the variables `a` and `b`. The swap function swaps the values of these variables, but due to a logical error, it does not work as intended.
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

**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);
    ...
}
```
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

swap.c

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

```c
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

```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`
# 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
- `a` 42
- `b` -7

#### Heap
- `a`
- `b`
- `tmp` 42

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

#### Read-Only Segment
- `.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

```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` function:
  - `a` = -7
  - `b` = 42

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

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

Heap

- `tmp` = 42
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: 42

- Heap

- Read/Write Segment
  - .data, .bss

- Read-Only Segment
  - .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
    - `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

```
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];
    ...     // array
    f(a);
    return 0;
}
```

Equivalent to:
```c
void f(int* a);
int main( ... ) {
    int a[5];
    ...     // array
    f(&a[0]);
    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 (*) (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\n",
            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