More Pointers, The Heap
CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:
Andrew Hu        Austin Chan        Brennan Stein
Cheng Ni         Cosmo Wang         Diya Joy
Guramrit Singh   Mengqi Chen        Pat Kosakanchit
Rehaan Bhimar    Renshu Gu          Travis McGaha
Zachary Keyes
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:
    - Autograder [0 to 3], Style [-2 to +0.1] – Overall [0, 1, 2, 3, 3.1]
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 1/23)
  - 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?**
Lecture Outline

- **Pointer Arithmetic**
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
- Heap-allocated Memory
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!
    - Not allowed, though confusingly GCC allows it as an extension 😞

- Valid pointer arithmetic:
  - Add/subtract an integer to/from a pointer
  - Subtract two pointers (within stack frame or malloc block)
  - Compare pointers (`<`, `<=`, `==`, `!=`, `>`, `>=`), including `NULL`
  - ... but plenty of valid-but-inadvisable operations, too
Polling Question

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. We’re lost...
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

<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>
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.
### 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.

---

<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>
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

```plaintext
0x100 0x101 0x102 0x103
Big-Endian

0x100 0x101 0x102 0x103
Little-Endian
d4  c3  b2  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 += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    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 += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    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 += 2;  // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

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`:

---

`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 EXIT_SUCCESS;
}
```

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

Arrow points to next instruction.

`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 EXIT_SUCCESS;
}
```

Stack (assume x86-64)

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; // uh oh
    int_ptr += 2;

    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

The code above demonstrates pointer arithmetic. The `int_ptr` points to the first element of the `arr` array, and `char_ptr` is cast from `int_ptr` to a `char*` and points to the first element of the `arr` array. The code increments `int_ptr` by 1 and then by 2, and similarly for `char_ptr`. This example highlights the difference in accessing elements of an array using pointers versus array indexing.

**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 |

**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 EXIT_SUCCESS;
}
```

Note: Arrow points to *next* instruction.

Stack
(assume x86-64)

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

int_ptr: 0x07ffffffde01c
*int_ptr: ???
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 EXIT_SUCCESS;
}

Note: Arrow points to next instruction.

Stack (assume x86-64)

char_ptr: 0x0000000000000001
*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 EXIT_SUCCESS;
}
```

```
char_ptr:  0x0x7fffffffe011
*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

---

**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 EXIT_SUCCESS;
}
```

`pointerarithmetic.c`

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: 0x07ffffffde013
    *char_ptr: 0
```

```c
    int_ptr:
    char_ptr:
```
Lecture Outline

- Pointer Arithmetic
- **Pointers as Parameters**
- Pointers and Arrays
- Function Pointers
- Heap-allocated Memory
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);
    ...
}
```

OS kernel [protected]

Stack

main a 42 b -7

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);
    ...
```

brokenswap.c

```
OS kernel [protected]

Stack

main

\textbf{a} 42 \textbf{b} -7

swap

\textbf{a} 42 \textbf{b} -7

\textbf{tmp} ??

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);
    ...
```
### 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 -7, b -7, tmp 42

- **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);
    ...
}
Broken Swap

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

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata
Lecture Outline

- Pointer Arithmetic
- Pointers as Parameters
- **Pointers and Arrays**
- Function Pointers
- Heap-allocated Memory
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 EXIT_SUCCESS;
}
```

Equivalent to:
```c
void f(int* a);

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

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- **Function Pointers**
- Heap-allocated Memory
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);
    ...
}
```
Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
- Heap-allocated Memory
Memory Allocation So Far

- So far, we have seen two kinds of memory allocation:

  - `counter` is *statically*-allocated
    - Allocated when program is loaded
    - Deallocated when process gets reaped

  - `a`, `x`, `y` are *automatically*-allocated
    - Allocated when function is called
    - Deallocated when function returns

```c
int counter = 0;  // global var

int main(int argc, char** argv) {
    counter++;
    printf("count = %d\n", counter);
    return EXIT_SUCCESS;
}

int foo(int a) {
    int x = a + 1;     // local var
    return x;
}

int main(int argc, char** argv) {
    int y = foo(10);   // local var
    printf("y = %d\n", y);
    return EXIT_SUCCESS;
}
```
Dynamic Allocation

- Situations where static and automatic allocation aren’t sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than can fit on the Stack
  - We need memory whose size is not known in advance to the caller

```c
// this is pseudo-C code
char* ReadFile(char* filename) {
    int size = GetFileSize(filename);
    char* buffer = AllocateMem(size);

    ReadFileIntoBuffer(filename, buffer);
    return buffer;
}
```
Dynamic Allocation

- What we want is *dynamically*-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)

- C requires you to manually manage memory
  - Gives you more control, but causes headaches
Aside: NULL

- **NULL** is a memory location that is guaranteed to be invalid
  - In C on Linux, **NULL** is `0x0` and an attempt to dereference **NULL** causes a segmentation fault
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It’s better to cause a segfault than to allow the corruption of memory!

```c
int main(int argc, char** argv) {
    int* p = NULL;
    *p = 1;  // causes a segmentation fault
    return EXIT_SUCCESS;
}
```
**malloc()**

- **General usage:**
  
  ```c
  var = (type*) malloc(size in bytes)
  ```

- **malloc** allocates a block of memory of the requested size
  
  - Returns a pointer to the first byte of that memory
  - And *returns NULL* if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You’ll typically use `sizeof` to calculate the size you need

```c
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
    return errcode;
}
...
 // do stuff with arr
```
calloc()

- General usage:

```c
var = (type*) calloc(num, bytes per element)
```

- Like `malloc`, but also zeros out the block of memory
  - Helpful when zero-initialization wanted (but don’t use it to mask bugs – fix those)
  - Slightly slower; but useful for non-performance-critical code
  - `malloc` and `calloc` are found in `stdlib.h`

```c
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
    return errcode;
}
...  // do stuff with arr
```
free ()

- **Usage:** `free(pointer);`

- **Deallocates the memory pointed-to by the pointer**
  - Pointer *must* point to the first byte of heap-allocated memory *(i.e. something previously returned by `malloc` or `calloc`)*
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call to `free`
    - Defensive programming: can set pointer to `NULL` after freeing it

```c
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
    return errcode;
...
    // do stuff with arr
free(arr);
arr = NULL;  // OPTIONAL
```
The Heap

- The Heap is a large pool of available memory used to hold dynamically-allocated data
  - `malloc` allocates chunks of data in the Heap; `free` deallocates those chunks
  - `malloc` maintains bookkeeping data in the Heap to track allocated blocks
    - Lab 5 from 351!
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```

Note: Arrow points to next instruction.

OS kernel [protected]

Stack

main

nums

ncopy

Heap (malloc/free)

Read/Write Segment

Read-Only Segment
(main, copy)
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```

OS kernel [protected]

Stack

main

nums 1 2 3 4
ncopy

copy

a 1 2 3 4
size 4
i 0

Heap (malloc/free)

Read/Write Segment

Read-Only Segment

(main, copy)
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```

OS kernel [protected]

Stack

main

nums 1 2 3 4
ncopy

copy

a
i 0
size 4
a2

Heap (malloc/free)

Read/Write Segment

Read-Only Segment (main, copy)
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```

# Heap and Stack Example

## arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // ... do stuff with the array ...
    free(ncopy);
    return EXIT_SUCCESS;
}
```

![Diagram showing heap and stack example]
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
# Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```

OS kernel [protected]

- Stack
- Heap (malloc/free)
- Read/Write Segment
- Read-Only Segment

- main
- ncopy
- nums
- free
Heap and Stack Example

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
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