



pollev.com/naomila



About how long did Exercise 1 take you?

- A. [0, 2) hours
- B. [2, 4) hours
- C. [4, 6) hours
- D. [6, 8) hours
- E. 8+ Hours
- F. I didn't submit / I prefer not to say

CSE333 Systems Programming

Pointers

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski

Barbora Buzkova

Mendel Carroll

Camden Harris

Hannah Hempstead

Rishabh Jain

Irene Lau

Jonathan Nister

Advay Patil

Aviel Wood

Angela Wu

Jennifer Xu

Administrivia (1/3)

- ❖ Exercise grading
 - Sample solutions released same day as deadline
 - Autograder scores should be visible around Sunday
 - Full grades follow shortly after
 - Style things to watch for:
 - FOLLOW THE SPEC (especially the Style Guide section)
 - Check the Google C++ Style Guide
 - Make a judgment call and document
 - Keep style tips in mind, as you will need to use them in HW

Administrivia (2/3)

- ❖ Pre-quarter survey (Google form) due tonight
- ❖ Exercise 2 out today and due Friday (4/10) morning
- ❖ Homework 1 out tonight, due in 2 weeks (Thu 4/16)
 - Linked list and hash table implementations in C
 - Get starter code by cloning your new *homework* repo

Administrivia (3/3)

❖ Documentation:

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

❖ Folklore:

- Google-ing, Stack Overflow, generative AI, that rando in Discord

Lecture Outline

- ❖ **Arrays**
- ❖ Pointer Basics
- ❖ Parameters
- ❖ Pointer Arithmetic

Arrays (review from last lec)

- ❖ Definition: `type name[size]` allocates `size*sizeof(type)` bytes of *contiguous* memory
 - By default, array values are “mystery” data (i.e., uninitialized)
 - Normal usage is a compile-time constant for `size`
e.g., `int scores[175];`
- ❖ Size of an array
 - Not stored anywhere – array does not know its own size!
 - `sizeof(array)` only works in the variable scope of array definition

Using Arrays

- ❖ Initialization: `type name[size] = {val0, ..., valN};`
 - `{}` initialization can *only* be used at time of definition
 - If no `size` supplied, infers from length of array initializer
- ❖ Array name used as identifier for “collection of data”
 - **Array name, in code, produces the address of the start of the array**
 - Cannot be assigned to / changed
 - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
 - **Is actually `*(name+index)`** with pointer arithmetic

```
int primes[6] = {2, 3, 5, 6, 11, 13};  
primes[3] = 7;  
primes[100] = 0; // memory smash!
```

Multi-dimensional Arrays

❖ Generic 2D format:

```
type name[rows][cols] = {{values},..., {values}};
```

- Still allocates a single, contiguous chunk of memory
- C is *row-major*

```
// a 2-row, 3-column array of doubles
double grid[2][3];

// a 3-row, 5-column array of ints
int matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
};
```

- 2-D arrays normally only useful if size known in advance; otherwise, use dynamically-allocated data

Structs

- ❖ The *size* and *layout* of a struct instance is completely determined by (1) the field ordering and (2) alignment requirements
 - Can review 351 if curious
- ❖ Compiler figures these things out at compile-time and will replace these “function calls” with their results before machine code generation
 - `sizeof(type)` returns the size in bytes
 - `offsetof(type, field)` returns offset value in bytes
 - Defined in `stddef.h`
- ❖ We’ll talk more about struct usage in Lecture 4

Lecture Outline

- ❖ Arrays
- ❖ **Pointer Basics**
- ❖ Parameters
- ❖ Pointer Arithmetic

Pointers

- ❖ Variables that store addresses
 - It points to somewhere in the process' virtual address space
 - `&foo` produces the virtual address of `foo`
- ❖ Generic definition: `type* name;` or `type *name;`
 - Recommended: do not define multiple pointers on same line:
`int *p1, p2;` not the same as `int *p1, *p2;`
 - Instead, use:
`int *p1;`
`int *p2;`
- ❖ *Dereference* a pointer using the unary `*` operator
 - Access the memory referred to by a pointer

Pointers

- ❖ Variable that store
▪ if points to some wh
▪ &foo produces the

Generic definition:

- Recommended: do
`int *p1, p2;` as `int *p1, *p2;`
- Instead, use:

Dereference a

- Access the m

The choice between `int* p;` and `int *p;` is not about right and wrong, but about style and emphasis. C emphasized expressions; declarations were often considered little more than a necessary evil

https://www.stoustrup.com/bs_faq2.html#whitespace

**Bjarne
don't
care**



Box-and-Arrow Diagrams (1/4)

boxarrow.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 EXIT_SUCCESS;
}
```

address

name	value
------	-------

Box-and-Arrow Diagrams (2/4)

boxarrow.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 EXIT_SUCCESS;
}
```

address	name	value
---------	------	-------

&x	x	value
&arr[2]	arr[2]	value
&arr[1]	arr[1]	value
&arr[0]	arr[0]	value
&p	p	value

stack frame for main()

Box-and-Arrow Diagrams (3/4)

boxarrow.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 EXIT_SUCCESS;
}
```

address

name	value
------	-------

&x	x	1
&arr[2]	arr[2]	4
&arr[1]	arr[1]	3
&arr[0]	arr[0]	2
&p	p	&arr[1]

Box-and-Arrow Diagrams (4/4)

boxarrow.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 EXIT_SUCCESS;
}
```

address

name	value
------	-------

0x7fff...4c	x	1
0x7fff...48	arr[2]	4
0x7fff...44	arr[1]	3
0x7fff...40	arr[0]	2
0x7fff...38	p	0x7fff...44

Lecture Outline

- ❖ Arrays
- ❖ Pointer Basics
- ❖ **Parameters**
- ❖ Pointer Arithmetic

Parameters: Reference vs. Value

- ❖ There are two fundamental parameter-passing schemes in programming languages
- ❖ **Pass-by-value**
 - Parameter is a local variable initialized with a copy of the calling argument when the function is called; manipulating the parameter only changes the copy, *not* the calling argument
 - **C, Java, C++** (most things)
- ❖ **Pass-by-reference**
 - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
 - C++ references (we'll see these later)

Arrays as Parameters

- ❖ It's tricky to use arrays as parameters
 - What happens when you use an array name as an argument?
 - Arrays do not know their own size

```
int SumAll(int a[]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = SumAll(numbers);
    return 0;
}

int SumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ...???)
}
```

Standard Idiom: Pass Size as Parameter

```
int SumAll(int a[], int size); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = SumAll(numbers, 5);
    printf("sum is: %d\n", sum);
    return 0;
}

int SumAll(int a[], int size) {
    int i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += a[i];
    }
    return sum;
}
```

arraysum.c

Arrays: Call-by-what?

- ❖ Technical answer: a `T []` array parameter is “promoted” to a pointer of type `T*`, and the *pointer* is passed by value
 - So it acts like a *call-by-reference array* – caller’s array can be changed if callee modifies the array parameter elements
 - But it’s really a *call-by-value pointer* – the callee’s pointer parameter can be changed without affecting the caller’s array
 - This is because `T [i]` is really `*(T+i)`. We aren’t changing `T`!

```
void CopyArray(int src[], int dst[], int size) {
    int i;
    dst = src; // doesn't copy the array, copies the address
    for (i = 0; i < size; i++) {
        dst[i] = src[i]; // copies source array to itself
    }
}
```



Array Parameters

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

This code:

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

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

void f(int a[]) {
```

Equivalent to:

```
void f(int* a);

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

void f(int* a) {
```

Returning an Array

- ❖ Local variables, including arrays, are allocated on the Stack
 - They “disappear” when a function returns!
 - Can't safely return local arrays from functions
 - Can't return an array as a return value – why not?

```
int* CopyArray(int src[], int size) {  
    int i, dst[size];    // OK in C99  
  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
  
    return dst;    // no compiler error, but wrong!  
}
```

buggy_copyarray.c

Solution: Output Parameter

- ❖ Create the “returned” array in the caller
 - Pass it as an **output parameter** to `CopyArray()`
 - A pointer parameter that allows the called function to store values that the caller can use
 - Works because arrays are “passed” as pointers

```
void CopyArray(int src[], int dst[], int size) {  
    int i;  
  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
}
```

copyarray.c

Array Memory Diagram (1/3)

```
int main(){
    int original[] = {123, 351, 333};
    int copy[3];
    → CopyArray(original, copy, 3);
}

void CopyArray(int src[], int dst[],
               int size)
{
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

original

original[2]	333
original[1]	351
original[0]	123
copy[2]	<???
copy[1]	<???
copy[0]	<???

copy

Array Memory Diagram (2/3)

```

int main(){
    int original[] = {123, 351, 333};
    int copy[3];
    CopyArray(original, copy, 3);
}

void CopyArray(int src[], int dst[],
               int size)
{
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
    
```

original

copy

original[2]	333
original[1]	351
original[0]	123
copy[2]	<???
copy[1]	<???
copy[0]	<???
src	●
dst	●
size	3
i	<???



Array Memory Diagram (3/3)

```
int main(){
    int original[] = {123, 351, 333};
    int copy[3];
    CopyArray(original, copy, 3);
}

void CopyArray(int src[], int dst[],
               int size)
{
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

dst[i] is really *(dst+i). We aren't changing dst!

original

copy

original[2]	333
original[1]	351
original[0]	123
copy[2]	333
copy[1]	351
copy[0]	123
src	●
dst	●
size	3
i	3



Array Memory Diagram (4/4)

```
int main(){
  int original[] = {123, 351, 333};
  int copy[3];
  CopyArray(original, copy, 3);
}

void CopyArray(int src[], int dst[],
               int size)
{
  for (int i = 0; i < size; i++) {
    dst[i] = src[i];
  }
}
```

original

original[2]	333
original[1]	351
original[0]	123
copy[2]	333
copy[1]	351
copy[0]	123

copy

Output Parameters

- ❖ Output parameters are common in library functions
 - `long int strtol(char* str, char** endptr, int base);`
 - `int sscanf(char* str, char* format, ...);`

```
int    num, i;
char*  p_end, str1 = "333 rocks";
char   str2[10];

// converts "333 rocks" into long - p_end is conversion end
num = (int) strtol(str1, &p_end, 10);

// reads string into arguments based on format string
num = sscanf("3 blind mice", "%d %s", &i, str2);
```

outparam.c

Lecture Outline

- ❖ Arrays
- ❖ Pointer Basics
- ❖ Parameters
- ❖ **Pointer Arithmetic**

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

Pointers and Arrays

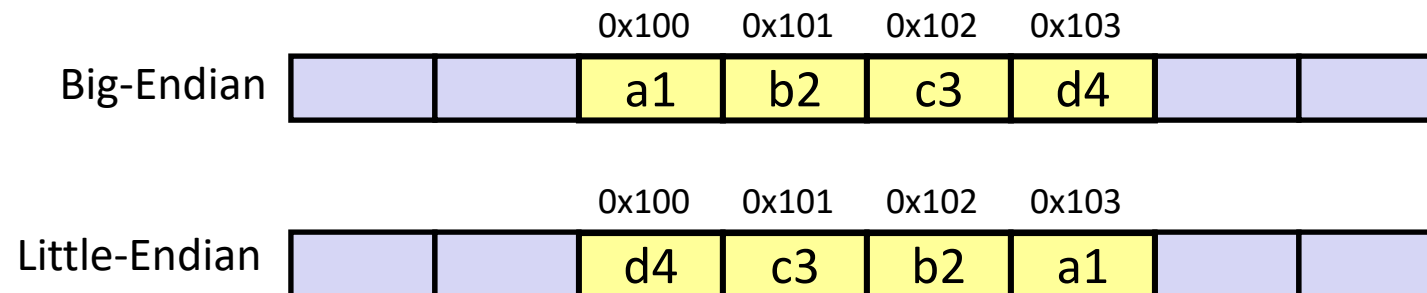
- ❖ A pointer can point to an array element
 - You can use array indexing notation on pointers
 - `ptr[i]` is `*(ptr+i)` with pointer arithmetic – reference the data `i` elements forward from `ptr`
 - An array name's value is the beginning address of the array
 - *Like a pointer to the first element 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
```

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/biggest* address
 - **Little-endian:** Least significant byte has *lowest/littlest* address
- ❖ **Example:** 4-byte data 0xa1b2c3d4 at address 0x100

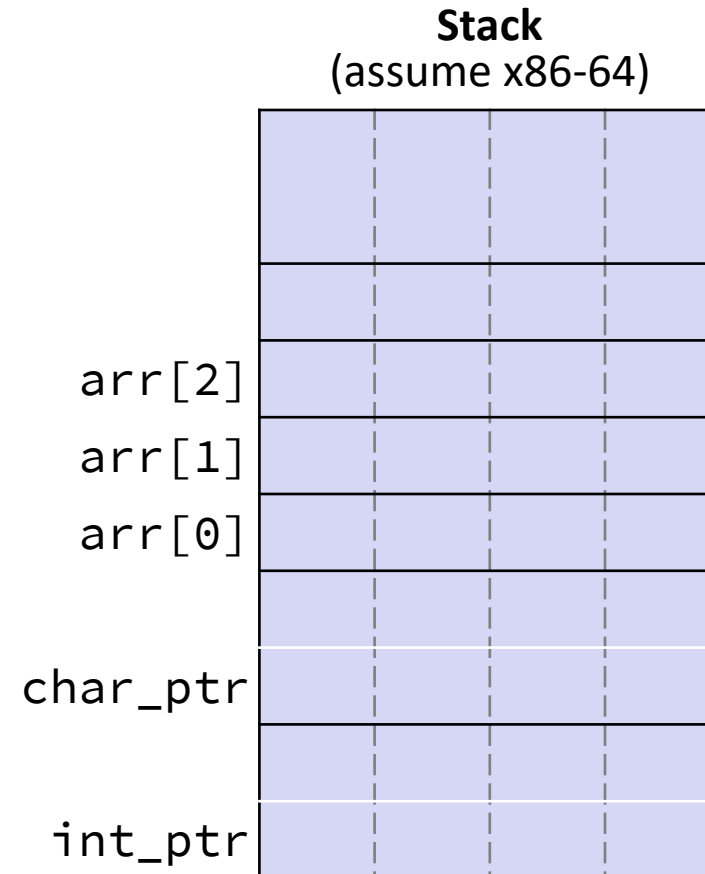


Pointer Arithmetic Example (1/10)

Note: Arrow points to *next* instruction.

```
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



Pointer Arithmetic Example (2/10)

```
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

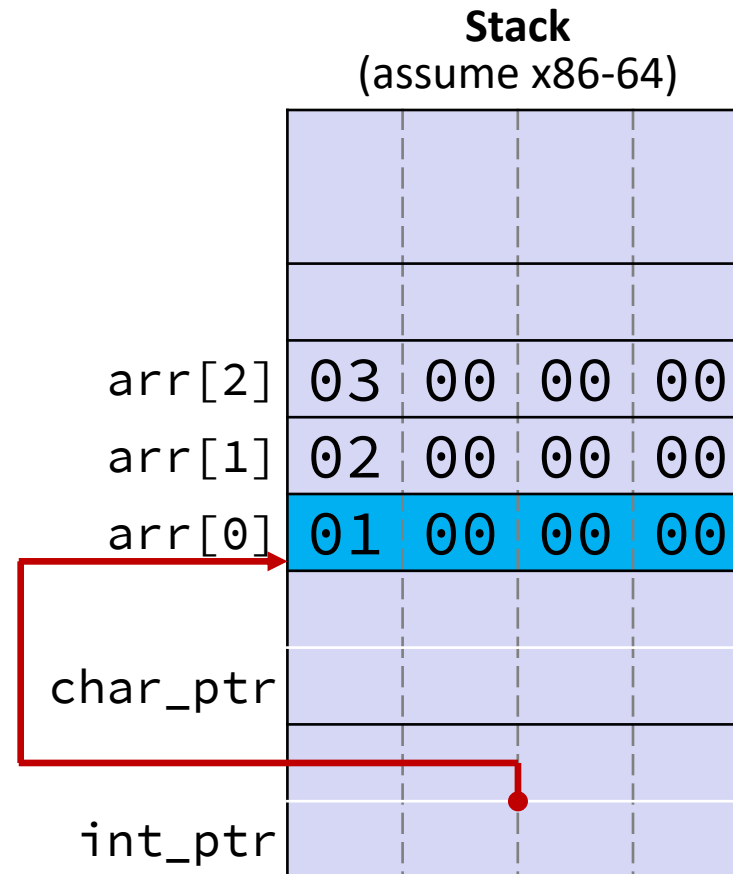
Stack
(assume x86-64)

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

Pointer Arithmetic Example (3/10)

```
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;  
}
```

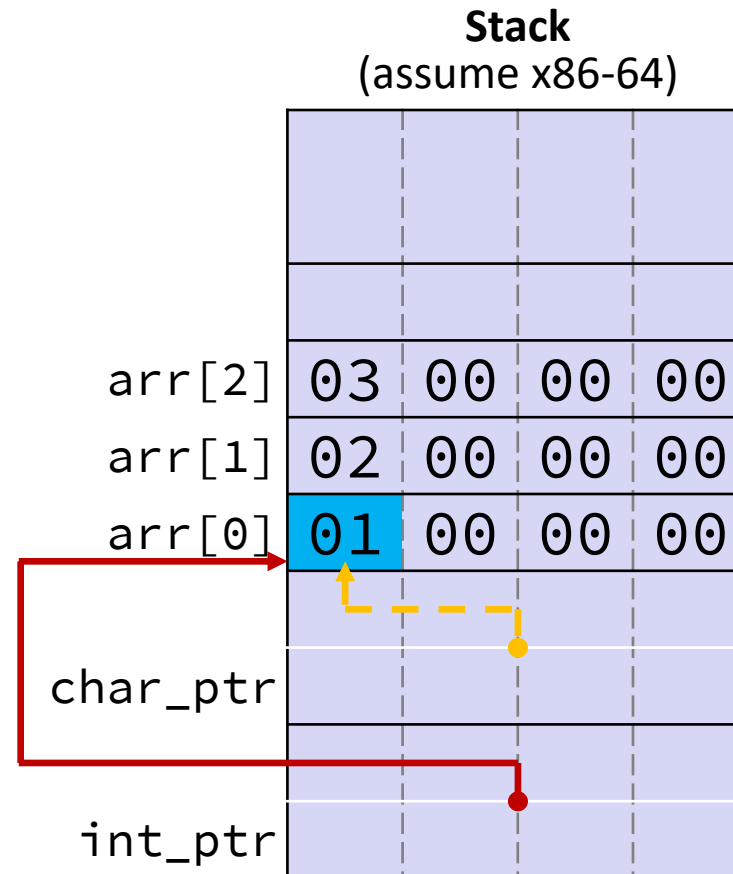
pointerarithmetic.c



Pointer Arithmetic Example (4/10)

```
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



Pointer Arithmetic Example (5/10)

```

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;
}

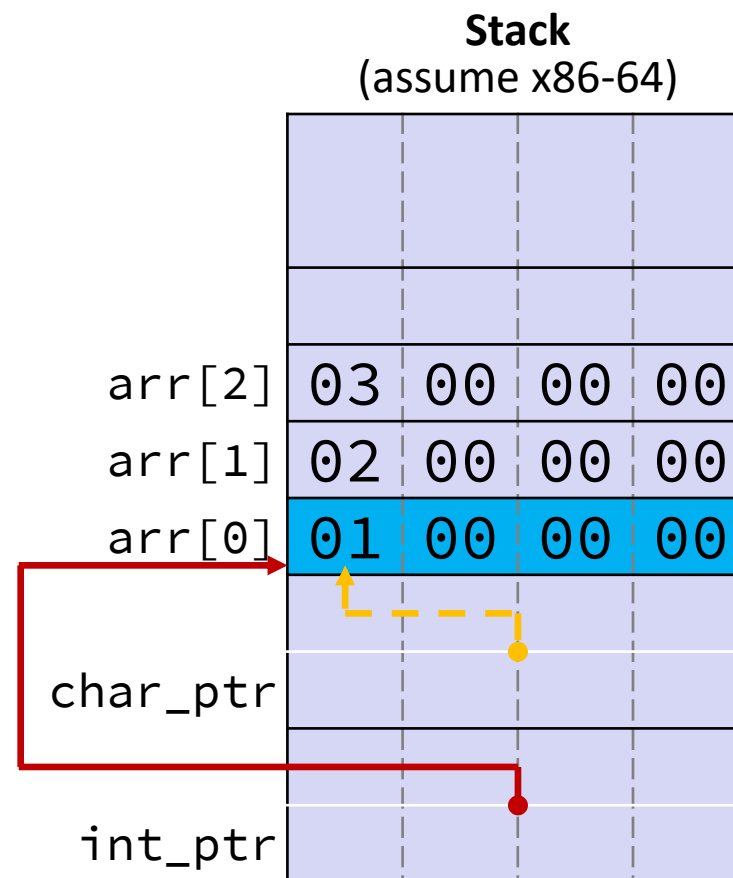
```

pointerarithmetic.c

```

int_ptr: 0x7fffffffde010
*int_ptr: 1

```



Pointer Arithmetic Example (6/10)

```

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;

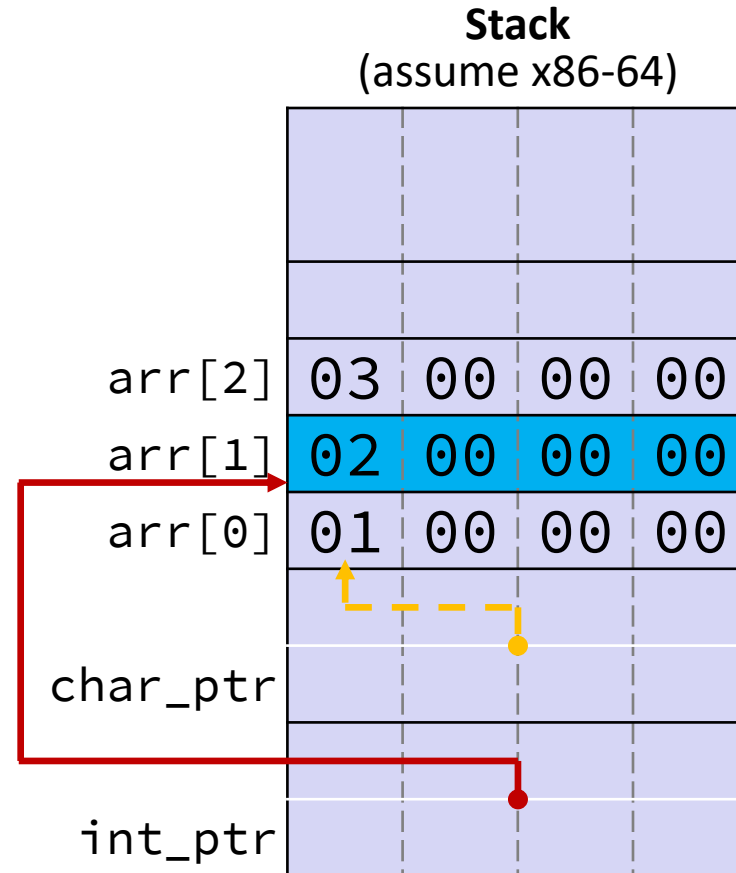
    return EXIT_SUCCESS;
}

```



pointerarithmetic.c

int_ptr: 0x7fffffffde014
***int_ptr:** 2



Pointer Arithmetic Example (7/10)

```

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;

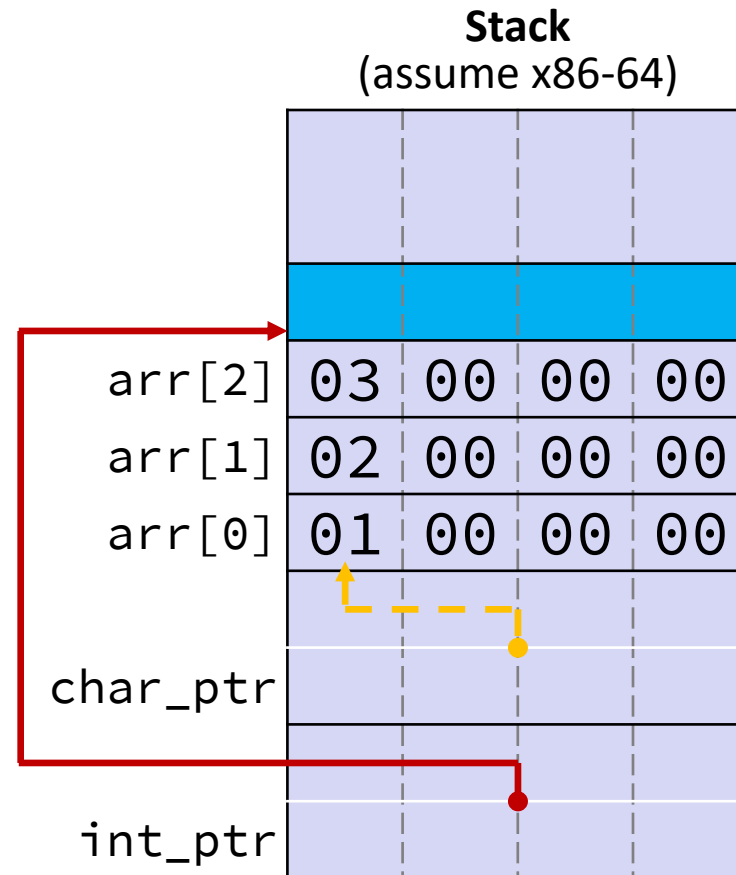
    return EXIT_SUCCESS;
}

```



pointerarithmetic.c

int_ptr: 0x7fffffffde01c
 *int_ptr: ???



Pointer Arithmetic Example (8/10)

```

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;

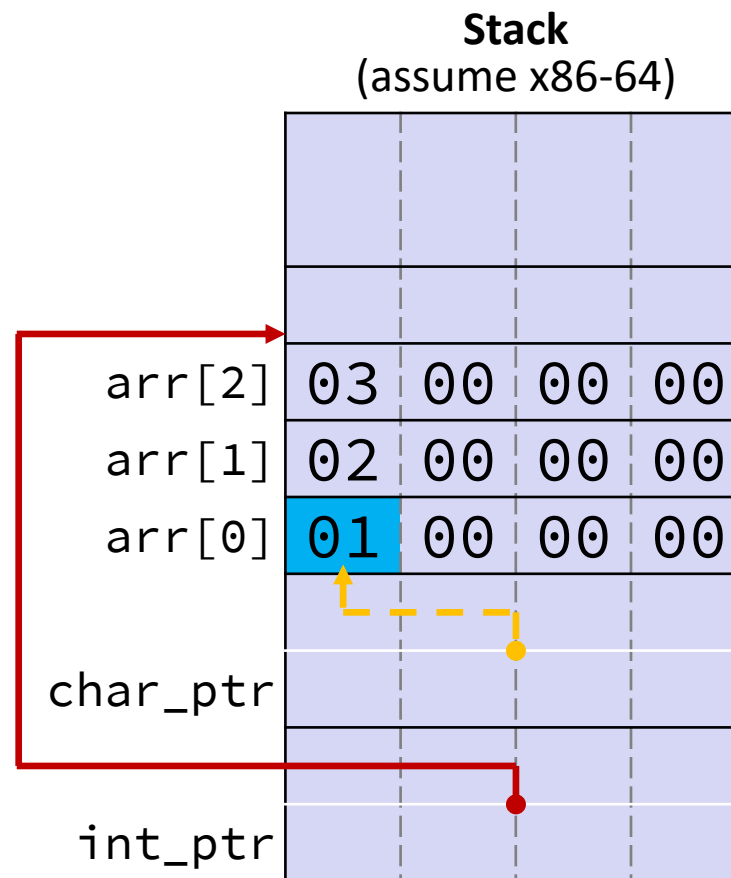
    return EXIT_SUCCESS;
}

```



pointerarithmetic.c

char_ptr: 0x7fffffffde010
***char_ptr:** 1



Pointer Arithmetic Example (9/10)

```

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

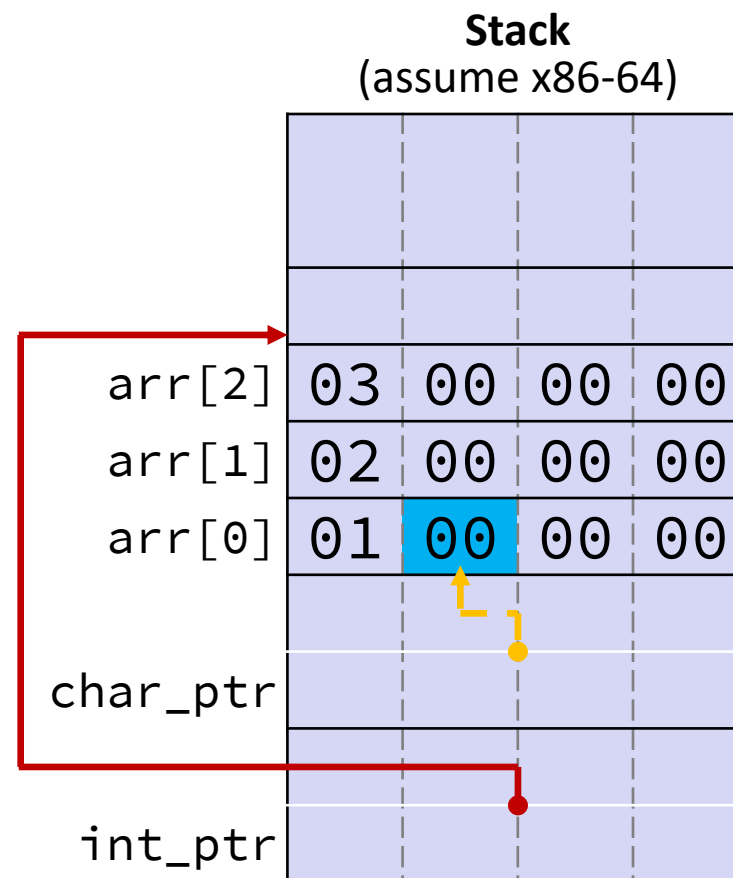
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}

```

pointerarithmetic.c

char_ptr: 0x7fffffffde011
 *char_ptr: 0



Pointer Arithmetic Example (10/10)

```

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

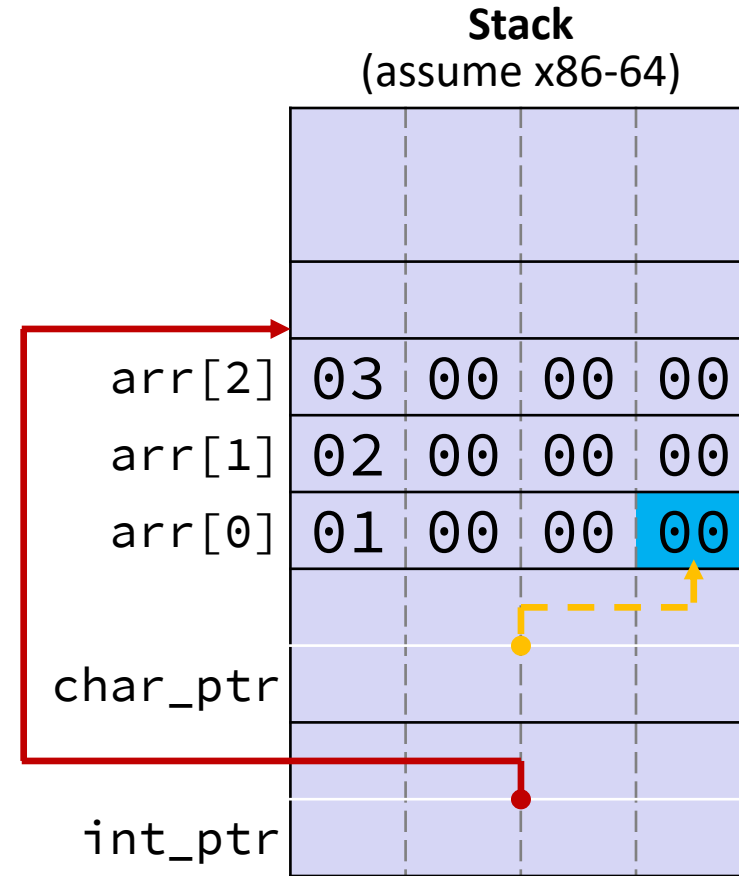
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}

```

pointerarithmetic.c

char_ptr: 0x7fffffffde013
***char_ptr:** 0



Extra Exercise #1

- ❖ Write a function that:
 - Accepts an array of 32-bit unsigned integers and a length
 - Reverses the elements of the array in place
 - Returns nothing (`void`)

Extra Exercise #2

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

```
#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 #3

- ❖ 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