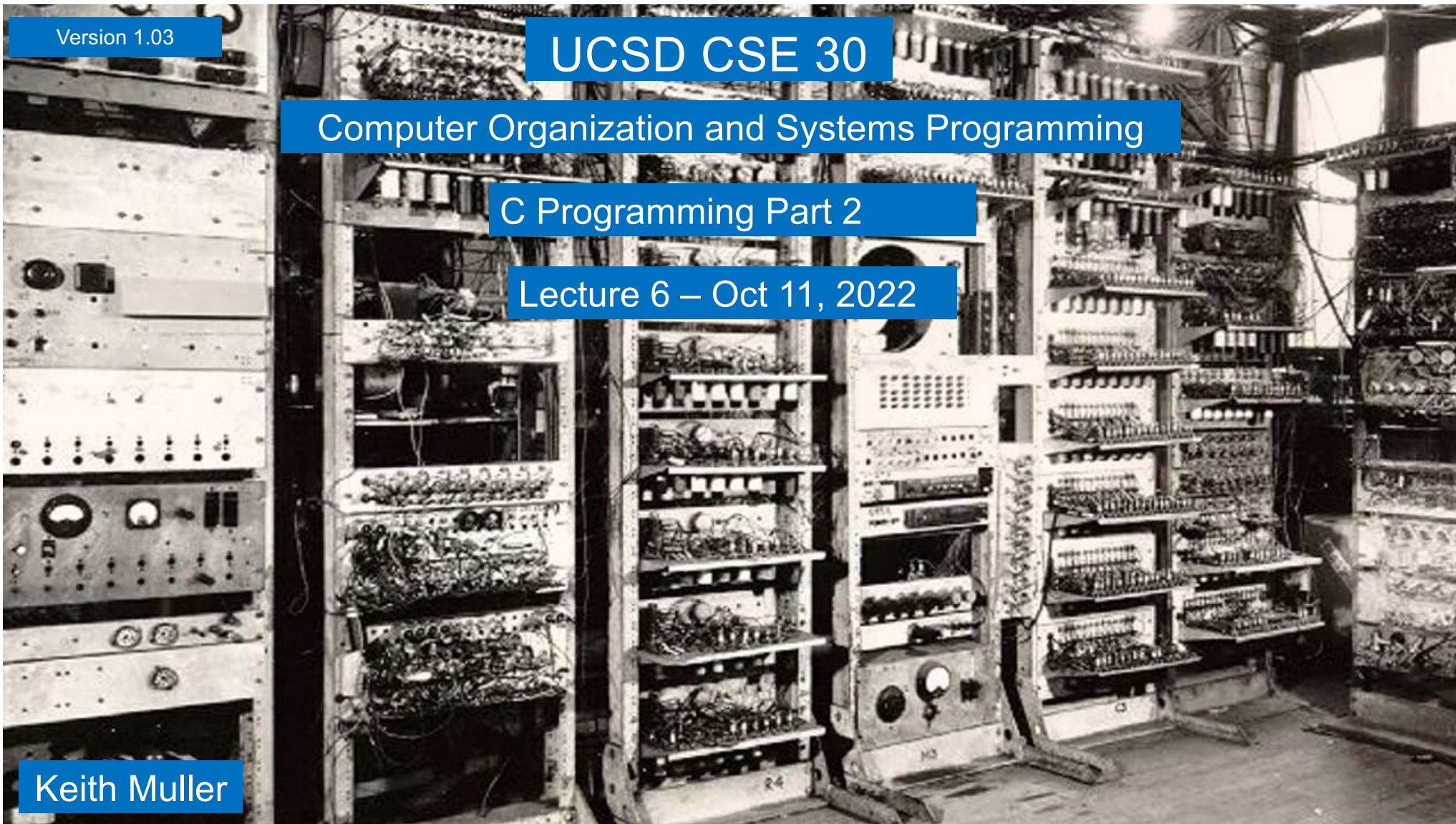Version 1.03

UCSD CSE 30

Computer Organization and Systems Programming
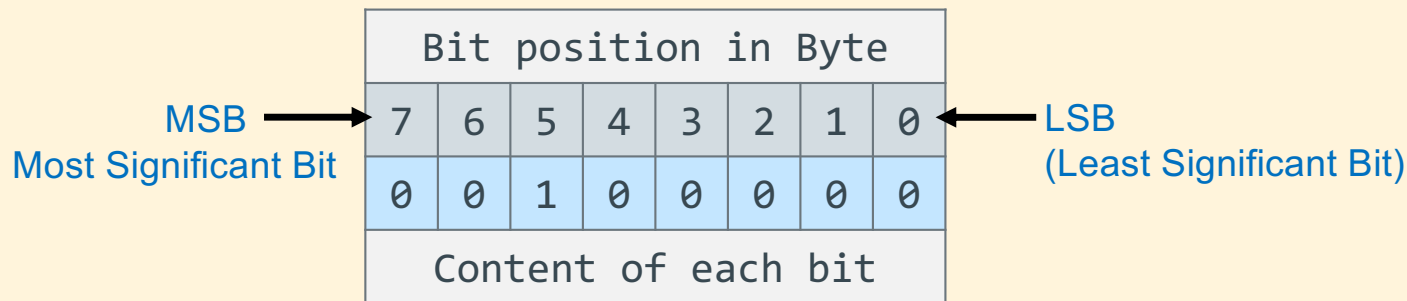
C Programming Part 2

Lecture 6 – Oct 11, 2022
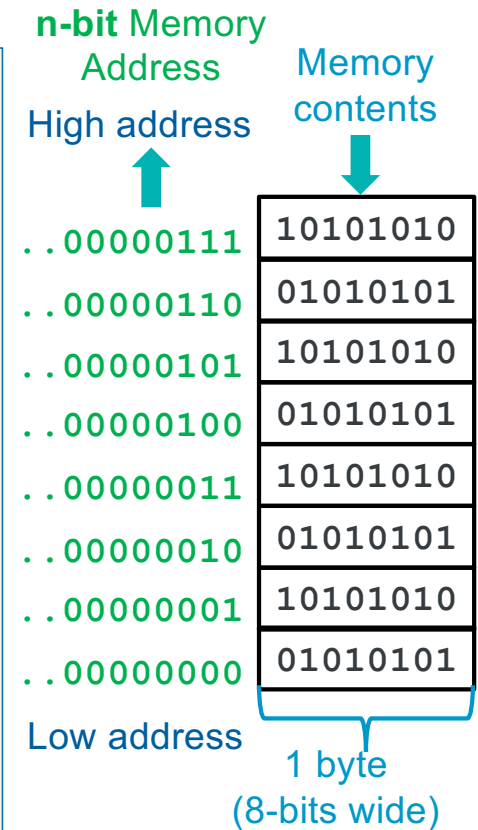
Keith Muller

Memory Review: Organized in Units of Bytes

- One bit (digit) of storage (in memory) has two possible **states**: 0 or 1

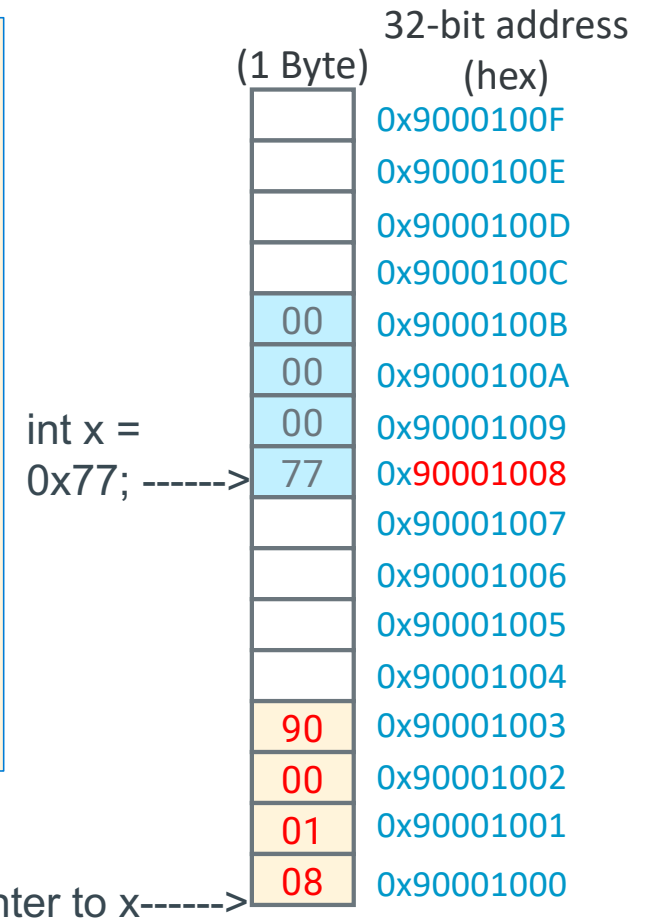- Memory is organized into a **fixed unit** of 8 bits, called a **byte**

	Bit position in Byte						
7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
	Content of each bit						

MSB →
Most Significant Bit

← LSB
(Least Significant Bit)

- Conceptually, memory is a single, **large array** of **bytes**, **where each byte** has a unique *address (byte addressable memory)*

- An address is an **unsigned** (positive #) *fixed-length* n-bit binary value
  - Range (domain) of possible addresses = *address space*

- Each byte in memory can be **individually accessed** and operated on given its **unique address**

n-bit Memory Address

Memory contents

High address

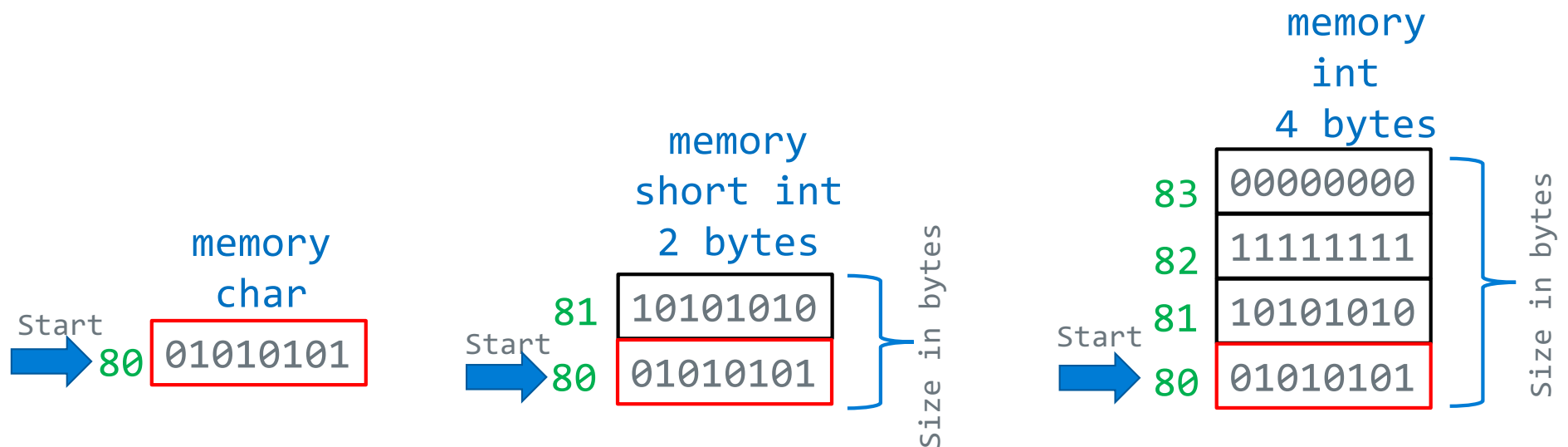| Address	Contents
..00000111	10101010
..00000110	01010101
..00000101	10101010
..00000100	01010101
..00000011	10101010
..00000010	01010101
..00000001	10101010
..00000000	01010101

Low address

1 byte
(8-bits wide)

2

x

Address and Pointers

- An address refers to a location in memory, the lowest or first byte in a contiguous sequence of bytes

- A pointer is a variable whose contents (or value) can be properly used as an address
  - The value in a pointer *should* be a valid address allocated to the process by the operating system

- The variable x is at memory address 0x90001008

- The variable pt is at memory location 0x90001000

- The contents of pt is the address of x 0x90001008

32-bit address (hex)

(1 Byte)

	0x9000100F
	0x9000100E
	0x9000100D
	0x9000100C
00	0x9000100B
00	0x9000100A
00	0x90001009
77	0x90001008
	0x90001007
	0x90001006
	0x90001005
	0x90001004
90	0x90001003
00	0x90001002
01	0x90001001
08	0x90001000

int x = 0x77; ------>

pt is a pointer to x------>

3

X

Variables in Memory: Size and Address

- The number of **contiguous bytes** a variable uses is based on the *type* of the variable
  - Different variable types require different numbers of contiguous bytes
- *Variable names* map to a *starting address in memory*
- Example Below: Variables all starting at address 0x80, each box is a byte

memory
char

Start
80 | 01010101

memory
short int
2 bytes

81 | 10101010
Start
80 | 01010101

Size in bytes

memory
int
4 bytes

83 | 00000000
82 | 11111111
81 | 10101010
Start
80 | 01010101

Size in bytes

sizeof(): Variable Size (number of bytes) *Operator*

```
#include <stddef.h>
/* size_t type may vary by system but is always unsigned */
```

sizeof() operator returns a value of type size_t:

the number of bytes used to store a variable or variable type

```
size_t size = sizeof(variable_type);
```
or
```
size_t size = sizeof(variable_name); // preferred!
```

- The argument to `sizeof()` is often an expression:

```
size = sizeof(int * 10);
```

- reads as:
  - number of bytes required to store **10 integers (an array of [10])**

X

Memory Addresses & Memory Content

```
x = x;      // Lvalue = Rvalue
```
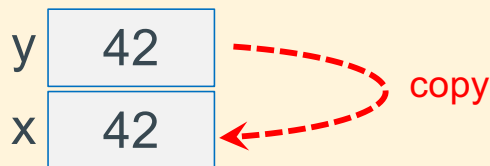
Variable name in a C statement evaluates to either:

- **Lvalue:** when on the left side (Lside or Left value) of the = sign is the
  - address where it is stored in memory – a constant
  - Address assigned to a variable cannot be changed at runtime
- **Rvalue:** when on the right side (Rside or Right value) of an = sign is the
  - contents or value stored in the variable (at its memory address)
  - requires a memory read to obtain

32-bit address
(1 Byte)   (hex)

	0x9000100F
	0x9000100E
	0x9000100D
	0x9000100C
00	0x9000100B
00	0x9000100A
00	0x90001009
77	0x90001008
	0x90001007
	0x90001006
	0x90001005
	0x90001004
00	0x90001003
00	0x90001002
01	0x90001001
08	0x90001000

x's contents (rvalue)➡ (points to 77 at 0x90001008)

x's address (lvalue) ⬅ (points to 0x90001008)

X

Memory Addresses & Memory Content

```
y = 42;

x = y;       // Lvalue = Rvalue
```

y | 42 |
x | 42 |

copy

- **x** on left side (**Lside**) of the assignment operator = evaluates to:
  - The address of the memory assigned to the x – this is x's **Lvalue**
- **y** on right side (**Rside**) of the assignment operator = evaluates to:
  - READ the contents of the memory assigned to the variable y (type determines length – number of bytes) - this is y's **Rvalue**
- So x = y; is:

    Read memory at y (**Rvalue**); write it to memory at x's address (**Lvalue**)

x

Introduction: Address Operator: &

- Unary *address operator* (&) produces the **address** of where an identifier is in memory

- Requirement: **identifier must have a Lvalue**
  - Cannot be used with constants (e.g., 12) or expressions (e.g., x + y)
  - `&12` does not have an *Lvalue*, so &12 is **not** a legal expression

- How can I get an address for use on the **Rside**? Three ways:
  - **&var** (any variable identifier or name)
  - **function_name** (name of a function, not func()); **&funct_name** is equivalent
  - **array_name** (name of the array like array_name[5]); &array_name is equivalent

x

Introduction: Address Operator: &

- Unary ***address operator*** (&) produces the **address** of where an identifier is in memory

- Example: this might print:

  value *of g is: 42*

  address *of g is: 0x71a0a0*

  (the address will vary)

```c
int g = 42;
int
main(void)
{
    printf("value of g is: %d\n", g);
    printf("address of g is: %p\n", &g);
    return EXIT_SUCCESS;

}
```

- *Tip*: `printf()` format specifier to display an address/pointer (in hex) is "%p"

X

Introduction: Pointer Variables - 1

- In C, there is a *variable type* for **storing an address**: a *pointer*
  - **Contents** of a pointer is an **unsigned** (0+, positive numbers) **memory address**

- When the **Rside of a variable** contains a **memory address**, (it **evaluates** to an **address**) the variable is called a **pointer variable**

- A pointer is defined by placing a *star (*or *asterisk) (*)* *before* the identifier (name)

```
type *name;  // defines a pointer; name contains address of a variable of type
```

X

Introduction: Pointer Variables - 1

```
type *name;   // defines a pointer; name contains address of a variable of type
```

- You also must specify the type of variable to which the pointer points

```
int i = 42;
int *p = &i;   /* p "points at" i (assign address of i to p) */
```

i [42]

p [ ]
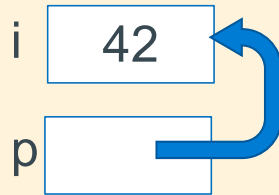
- Recommended: be careful when defining multiple pointers on the same line:

```
int *p1, p2;
```
is not the same as:
```
int *p1, *p2;
```

Use instead:
```
int *p1;
int *p2;
```

x

Introduction: Pointer Variables - 2

- **Pointers are <u>typed</u>**! Why?
  - The compiler needs the size (sizeof()) of the data **you are pointing at** (number of bytes to access)

- A pointer definition:

      int *p = &i;   /* p points at i (assign address i to p) */

- Is the same as writing the following definition and assignment statements

      int *p;   /* p is defined (not initialized) */
      p = &i;   /* p points at i (assign address i to p */

- The * is part of the definition of p and is not part of the variable name
  - The name of the variable is simply p, not *p

- C mostly ignores whitespace, so these three definitions are equivalent

      int *p = &i;   /* Style A */
      int * p = &i;   /* Style B */
      int* p = &i;   /* Style C */

X

Introduction: Pointer Variables - 3

• As with any variable, its value can be changed

```
p = &j;        /* p now points at j */
```

i `42`
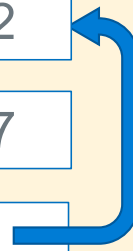
j `77`

p

```
p = &i;        /* p now points at i */
```

i `42`

j `77`

p

X

Introduction: Pointer Variables - 4

- Pointer variables all use the **same amount of memory** no matter what they point at

```
    int *iptr;
    char *cptr;

    printf("iptr(%u) cptr(%u)\n", sizeof(iptr), sizeof(cptr));
```

- Above prints on a 32-raspberry pi `iptr(4) cptr(4)`

X

Introduction: Indirection (or dereference) Operator: *

- The **indirection operator** (*) or the *dereference operator to a variable* is the **inverse** of the *address operator* (&)

- **address operator (&)** can be thought of as:

    "get the address of this box"

    i or (*p) | 42 |

    p = &i | |
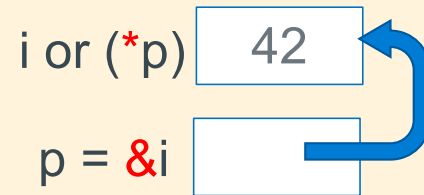
- **indirection operator (*)** can be thought of as:

    "follow the arrow to the next box and get its contents"

X

Introduction: Indirection (or dereference) Operator: *

*Contents of **p** is the address of **i** (p points at i)*

```
int i = 42;
int *p = &i;

printf("*p is %d\n", *p);
```

```
% ./a.out
*p is 42
```

X

Introduction: Indirection Operator Rside

- Performs the following steps when the * is on the Rside:

1. read the contents of the variable to get an address

2. **read** and return the contents at that address
   - (requires two reads of memory on the Rside)

```
z = *x; // copy the contents of memory pointed at by x to z
```

read (address)   read

Rside x [   ] ——————▶ [0x0c]

z [0x0c] ◀- - - - copy

x
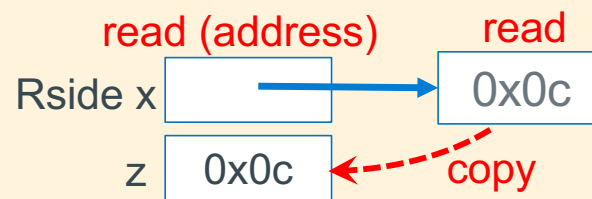
Introduction: Indirection Operator Lside

Performs the following steps when the * is on the Lside:

1. read the contents of the variable to get an address

2. **write** the evaluation of the Rside expression to that address

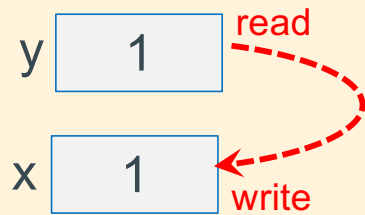   - (requires one read of memory and one write of memory on the Lside)

```
*z = x; // copy the value of x to the memory pointed at by z
```
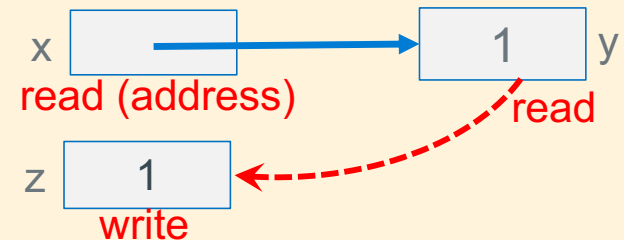
X
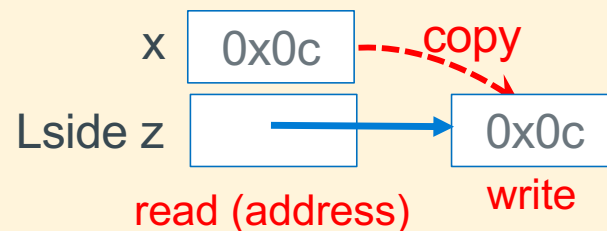
Each use of a * operator results in one additional read -1

Each * when used as a dereference operator in a statement (Lside and Rside) generates an additional read
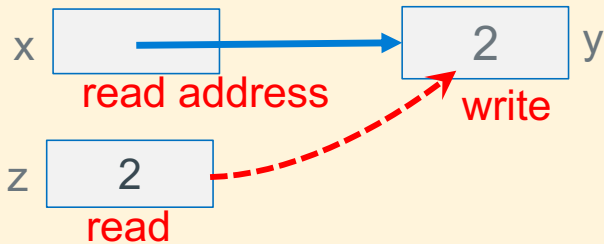
```
int x = 2, y = 1;
x = y; // one read
```

```
int z = 2, y = 1;
int *x = &y;
z = *x; // two reads
```

X

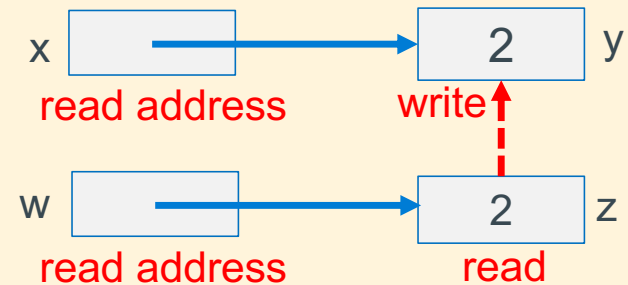# Each use of a * operator results in one additional read -2

- Each * when used as a dereference operator in a statement (Lside and Rside) generates an additional read

```
int z = 2, y = 1;
int *x = &y;
*x = z;
```



```
int z = 2, y = 1;
int *x = &y;
int *w = &z;
*x = *w;
```

x

Recap: Lside, Rside, Lvalue, Rvalue

```
int x = 2, y = 1;
x = y;
```

Constant Var Name	Lvalue address	Rvalue Contents	
y	0x108	0x1	read
x	0x104	0x1	write

```
int z = 2, y = 1;
int *x = &y;
int *w = &z;
*x = *w;
```

```
*x on Lside is  0x108
 w on Rside is  0x100
*w on Rside is  2
```

Constant Var Name	Lvalue address	Rvalue Contents	
x	0x10c	0x108	read (address)
y	0x108	0x2	write
z	0x104	0x2	read
w	0x100	0x104	read (address)

x

Pointer Practice

```
int *ptr;
```
Declares a variable, `ptr`, which is a pointer to (*it* contains the address of) an `int` in memory

ptr [ ]

```
int x = 5;

int y = 2;
```
Declares two variables, `x` and `y`, that contain `ints`, and *initializes* them to 5 and 2, respectively

x [5] write
y [2] write

```
ptr = &x;
```
Sets `ptr` to contain the address of `x` ("`ptr` points to `x`")

write
ptr [ ] ──→ x [5]
y [2]

```
y = 1 + *ptr;
```
"Dereference `ptr`"

Sets `y` to "1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y = 1 + x;`

read
ptr [ ] ──→ x [5] read
y [6] write
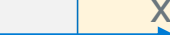
```
x = *(&y);
```
Sets x = y; The * and & cancel each other. get the address of y and then get the contents pointed by that address

ptr [ ] ──→ x [6] write
y [6] read

22

x

The NULL Constant and Pointers

- **NULL is a constant** that **evaluates to zero (0)**

- You assign a pointer variable to contain NULL to indicate that the pointer does not point at anything

- A pointer variable with a value of NULL is called a "NULL pointer" (invalid address!)

- Memory location 0 (address is 0) is not a valid memory address in any C program

- Dereferencing NULL at runtime will cause a program fault (segmentation fault)!

```
p = NULL;
i = *p;                   /* segmentation fault! */
*(int *)900000 = 25;  /* cast 900000 to a pointer */
                      /* if writeable address space, it works */
                      /* that memory location just changed */
```

x

Using the NULL Pointer

- Many functions return NULL to indicate an error has occurred

```
/* these are all equivalent */
int *p = NULL;
int *p = (int *)0;    // cast 0 to a pointer type
int *p = (void *)0;   // automatically gets converted to the correct type
```

- NULL is considered "false" when used in a Boolean context
  - **Remember: false expressions** in C are defined to be zero *or* NULL

- The following two are equivalent (the second one is preferred for readability):

```
if (p) ...
if (p != NULL) ...
```

X

What is Aliasing?

- Two or more variables are aliases of each other when they all reference the same memory (so different names, same memory location)

- When one pointer is copied to another pointer it *creates an **alias***

- ***Side effect***: Changing one variables value (content) changes the value for other variables

  - Multiple variables all read and write the **same** memory location
  - Aliases occur either by accident (coding errors) or deliberate (careful: readability)

```
int i = 5;
int *p = &i;
int *q;

q = p;    // *p & *q are aliases
*q = 4;   // changes i
```

*p and *q are aliases

p [ ] → [4] i

q [ ]
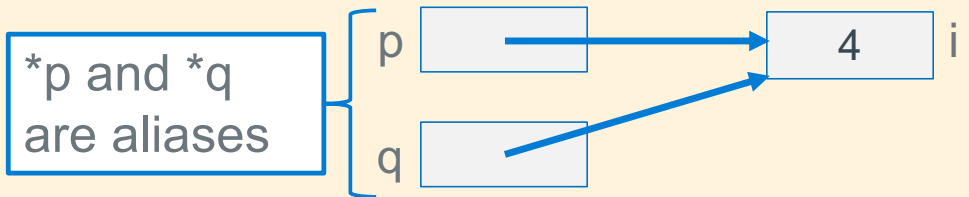
Result *p, *q and i all have the value of 4

X

Version 1.03

UCSD CSE 30

Computer Organization and Systems Programming

C Programming Part 2

Lecture 7 – Oct 13, 2022

Keith Muller
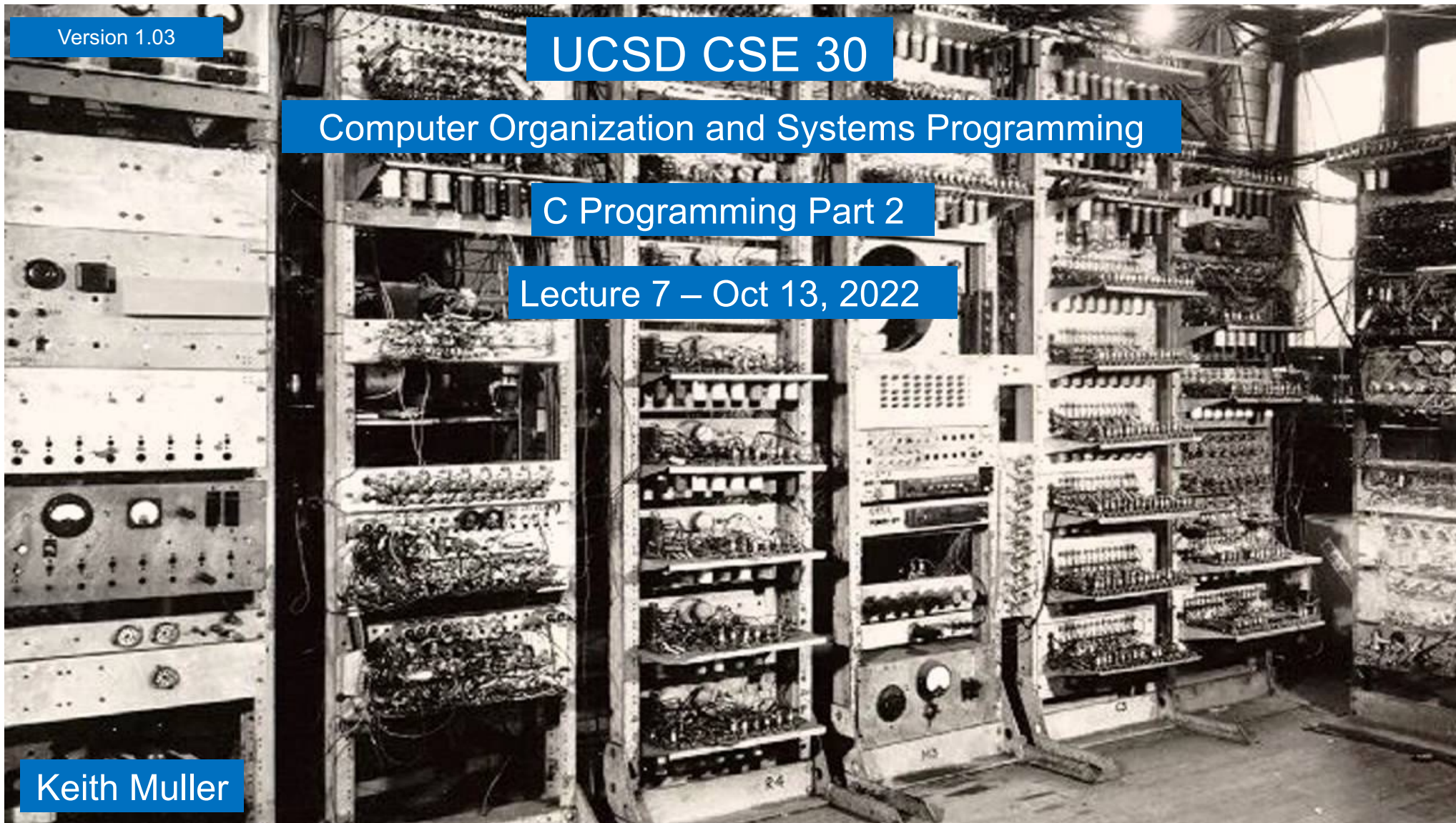
Defining Arrays

Definition: `type` `name`[`count`]

- *"Compound"* data type where each value in an array is an element of type
- Allocates **name** with a *fixed* count array elements of type **type**
- Allocates (`count` * `sizeof`(`type`)) bytes of ***contiguous memory***
- Common usage is to specify a compile-time constant for `count`

  ```
  #define BSZ     6
  int b[BSZ];
  ```
  BSZ is a macro replaced by the C preprocessor at compile time

- Array **names are constants (like all variable names)** and cannot be assigned (the name cannot appear on the Lside by itself)

  ```
  a = b;        // invalid does not copy the array
                // copy arrays element by element
  ```

**1 word
(int = 4 bytes)**

??	high memory address
??	
??	
??	
??	
??	
??	
??	
b[5] ??	9020
b[4] ??	9016
b[3] ??	9012
b[2] ??	9008
b[1] ??	9004
b[0] ??	9000

```
int b[6];
```

27

X
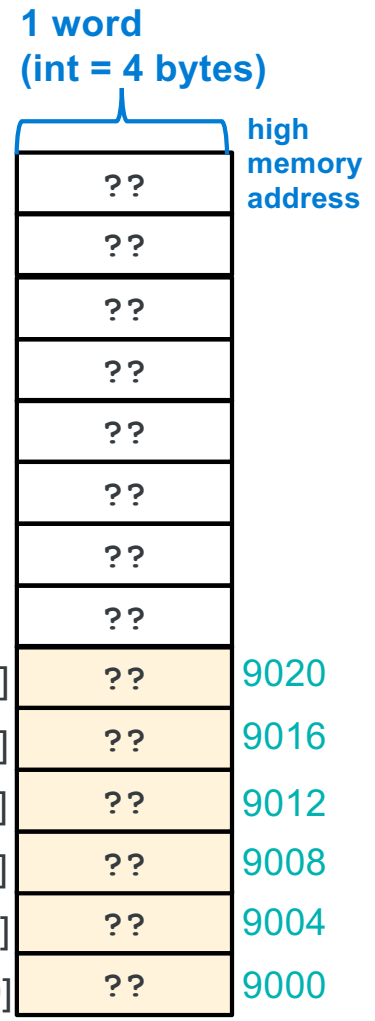
Array Initialization

- Initialization: `type name[count] = {val0,…,valN};`

  - `{ }` *(optional)* initialization list can <u>*only*</u> be used at **time** of **definition**

  - If no `count` supplied, `count` is determined by compiler using the number of array initializers | no initialization values given; then elements are initialized to 0 |

  - `int block[20] = {};` `//only works with constant size arrays`
    - defines an **array of 20 integers** each element filled with zeros
    - Performance comment: do not zero automatic arrays unless really needed!

  - When a **count** is given:
    - **extra** *initialization values* are **ignored**
    - **missing** *initialization values* are set to **zero**

1 word (int = 4 bytes)

	address
??	high address
??	
??	
??	
??	
??	
??	
??	
b[5] ??	0020
b[4] 11	0016
b[3] 6	0012
b[2] 5	0008
b[1] 3	0004
b[0] 2	0000

low address

```
int block[5] = {2, 3, 5, 6, 11, 13};
```

not needed and if used **may** truncate initialization list

6 initialization values given, **only 5 are used**

X
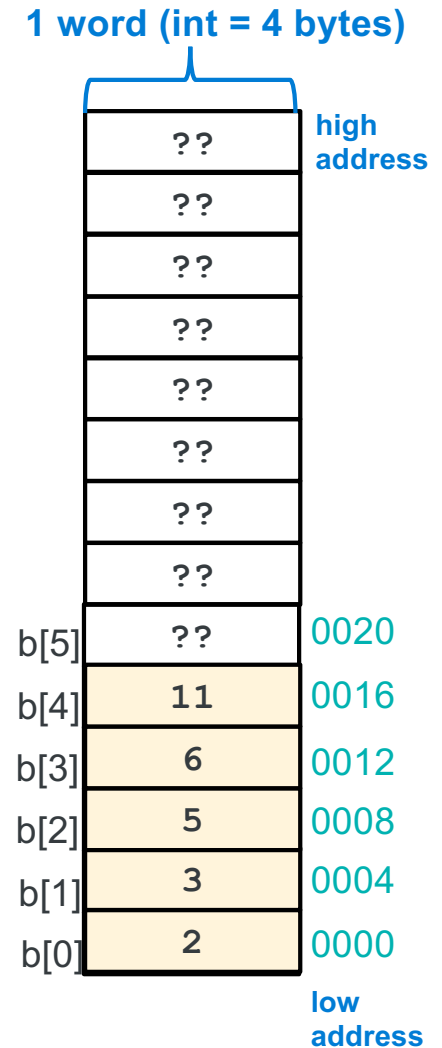
Accessing Arrays Using Indexing

- **name**[**index**] selects the **index** element of the array
  - index **should be** unsigned
  - Elements range from: 0 to count – 1 (int x[count];)

- **name**[**index**] can be used as an assignment target or as a value in an expression

  ```
  int a[5];
  int b[5];
  ```

- Array name (by itself with no [ ]) on the Rside evaluates to the address of the first element of the array

  ```
  int b[5];
  int *p = b;
  ```

  p | 9000 |

**1 word
(int = 4 bytes)**

??	**high address**
??	
??	9020
b[4] ??	9016
b[3] ??	9012
b[2] ??	9008
b[1] ??	9004
b[0] ??	9000

low address

29

X
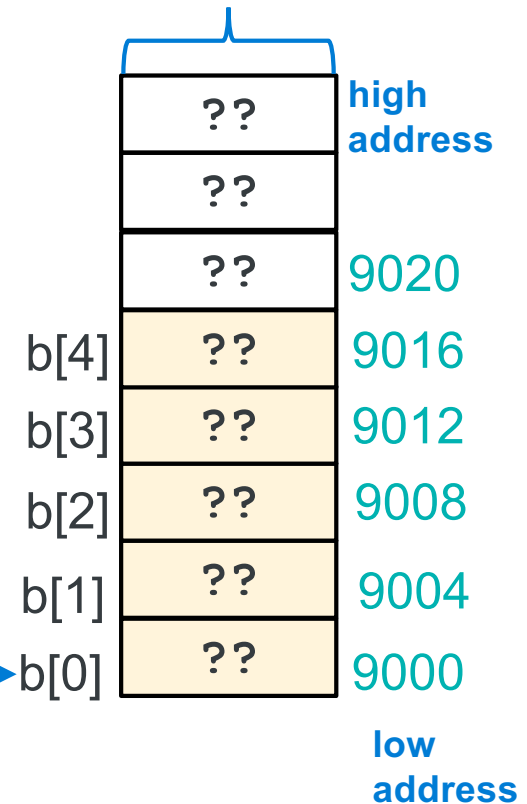
How many elements are in an array?

- **The number of elements of space allocated to an array (called element count) and indirectly the total size in bytes of an array** is not stored anywhere!!!!!!

- An **array name** is just the **address of the first element in a block of contiguous memory**
  - So an array does not know its own size!

```
#define SZ 6
int block[SZ];      // you specify the array has SZ elements
int indx;           // use when SZ is defined

for (indx = 0; indx < SZ; indx++)
    block[indx] = 0;
```

`int b[6];`

**1 word
(int = 4 bytes)**

	??	high memory address
	??	
	??	
	??	
	??	
	??	
	??	
	??	
b[5]	??	0020
b[4]	??	0016
b[3]	??	0012
b[2]	??	0008
b[1]	??	0004
b[0]	??	0000

X
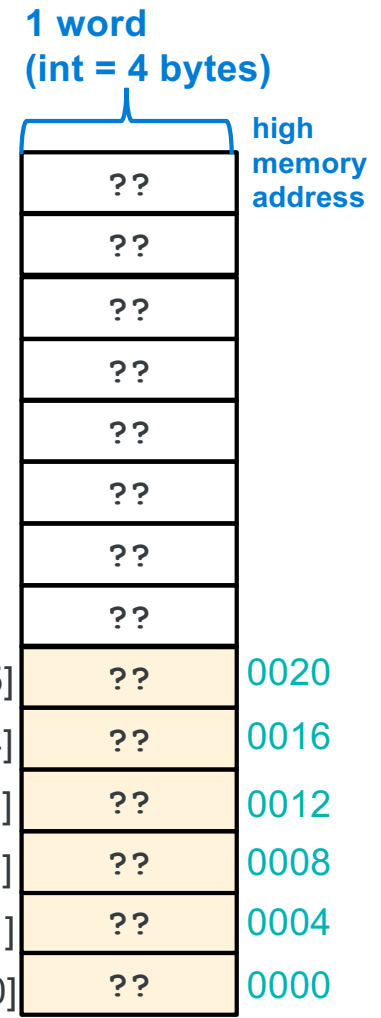
Determining Element Count for a compiler calculated array

- Programmatically determining the element count in a compiler calculated array

    sizeof(array) / sizeof(of just one element in the array)

- sizeof(array) **only works** when used in the SAME **scope** as where the array variable was defined

```c
#include <stddef.h>

int block[] = {2, 3, 5, 6, 11, 13};     // automatic: compiler calculates array size

int cnt = (int)(sizeof(block) / sizeof(block[0])); // in this case cnt = 6

for (int indx = 0; indx < cnt; indx++)
      block[indx] = 0;
```

X

Pointer and Arrays - 1

**1 byte Memory Content
One byte per row**

- A few slides back we stated: Array name (by itself) on the Rside evaluates to the address of the first element of the array

```
int buf[] = {2, 3, 5, 6, 11};
```

- Array indexing syntax ([ ]) an operator that performs *pointer arithmetic*

- **buf and &buf[0]** on the **Rside are equivalent**, *both evaluate* to the address of the first array element

```
int *p = buf;          // or int *p = &buf[0];
int *p1 = &buf[1];
int *p2 = &buf[2];
int *p3 = &buf[3];

*p = *p + 10;
*p1 = *p1 + 10;        // {12, 13, 5, 6, 11}
```

Byte Memory Address

p2

	Byte Memory Address
0x00	0x12345687
0x00	0x12345686
0x00	0x12345685
0x03	0x12345684
0x00	0x12345683
0x00	0x12345682
0x00	0x12345681
0x02	0x12345680

p1

p

X

Pointer and Arrays - 2

When p is a pointer, the actual value of (p+1) **depends on the type** that pointer p points at

- **(p+1)** adds `1 x sizeof(what p points at)` bytes to p
  - **++p** is equivalent to `p = p + 1`

- Using pointer arithmetic to find array elements:
  - Address of the second element **&buf[1]** is **(buf + 1)**
  - It can be referenced as ***(buf + 1) or buf[1]**

```
int buf[] = {2, 3, 5, 6, 11};
int *p = buf;

*p = *p + 10;
*(p + 1) = *(p + 1) + 10; // {12, 13, 5, 6, 11}
```

	index	pointer	pointer
	buf[2]	*(buf+2)	*(p+2)
0x00			
0x00			
0x00			
0x03	buf[1]	*(buf+1)	*(p+1)
0x00			
0x00			
0x00			
0x02	buf[0]	*buf	*p

X

Pointer Arithmetic In Use – C's Performance Focus

```
char a[] = {'A', 'B', 'C'};
```

address

a+3	→	&a[3]
a+2	→	&a[2]
a+1	→	&a[1]
a	→	&a[0]
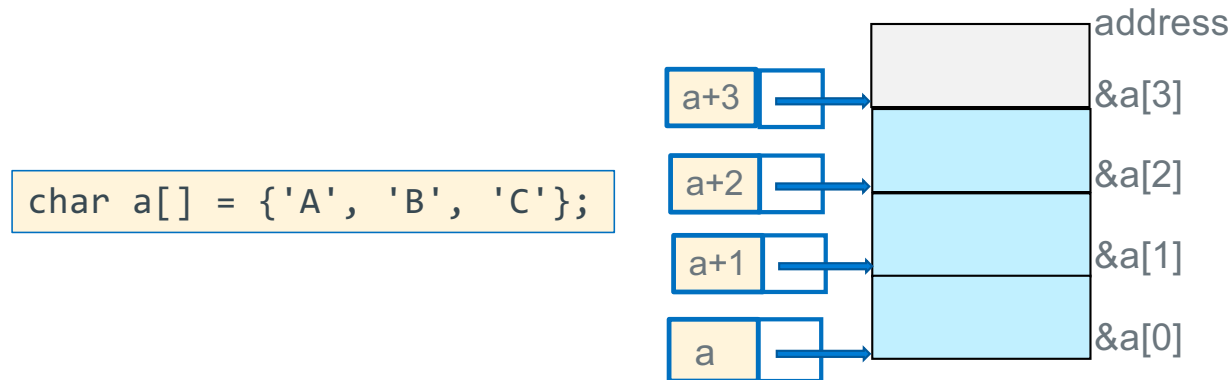
- **Alert!:** C performance focus **does not** perform any array "bounds checking"

- **Performance by Design**: *bound checking **slows down execution** of a properly written program*

- Example: array **a** of length i, C **does not verify** that **a[j] or *(a + j)** is valid (does not check: 0 ≤ j < i)
  - C simply *"translates"* and accesses the memory specified from: `a[j]` to be `*(a + j)` which may be *outside the bounds* of the array
  - OS only ***"faults"*** for an incorrect <u>access</u> to memory (read-only or <u>not</u> assigned to your process)
    - It does not fault for out of bound indexes or out of scope

- **lack of bound checking** is a common source of **errors** and **bugs** and is a common criticism of C

X

Pointer Arithmetic

- **You cannot add two pointers** *(what is the reason?)*

- A pointer q can be subtracted from another pointer p when the pointers are the same type – best done only within arrays!

- The value of `(p-q)` is the number of **elements between** the two pointers
  - Using memory address arithmetic (p and q Rside are both byte addresses):

  ```
  distance in elements = (p – q) / sizeof(*p)

  (p + 3) – p = 3 = (0x08c – 0x080)/4 = 3
  ```

p+3   0x08c

4-byte integer

int *q = p+2;
p+2   0x088

4-byte integer

p+1   0x084

4-byte integer

int *p;
p   0x080
           X

35
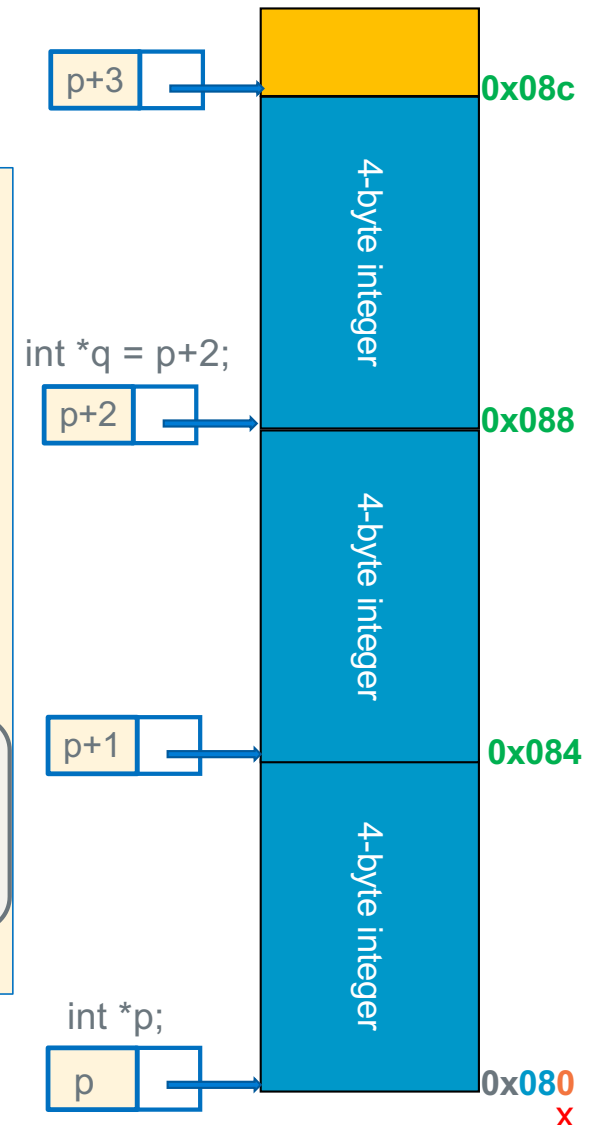
Pointer Comparisons

- Pointers (**same type**) can be compared with the comparison operators:

  `<, <=, ==, !=, >=, >`

  ```
  int numb[] = {9, 8, 1, 9, 5};
  int *end = numb + (int) (sizeof(numb)/sizeof(*numb));
  int *a = numb;

  while (a < end)  // compares two pointers (address)
        /* rest of code */
  ```

- Invalid, Undefined, or **risky** pointer arithmetic (some examples)
  - Add, multiply, divide on two pointers
  - Subtract two pointers of different types or pointing at different arrays
  - Compare two pointers of different types
  - Subtract a pointer from an integer

Fast Ways to "Walk" an Array: Use a Limit Pointer

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));


int *ptr = x; //or &x[0]
```

xpt is a loop **limit pointer** it points 1 element past the end of the array

```
int *xpt = ptr + cnt;

while (ptr < xpt) {
    printf("%#x\n", *ptr);
    ptr++;
}
```

```
cnt   = 3;
bytes = cnt * sizeof(*x);
      = 12
```

```
% ./a.out
0xd4c3b2a1
0xd4c3b200
0x12345684
```

xpt		0x??	0x1234568c
		0x12	0x1234568b
		0x34	0x1234568a
		0x56	0x12345689
		0x84	0x12345688
		0xd4	0x12345687
		0xc3	0x12345686
		0xb2	0x12345685
		0x00	0x12345684
		0xd4	0x12345683
		0xc3	0x12345682
		0xb2	0x12345681
ptr		0xa1	0x12345680
		0x??	0x1234567f

1 byte

X
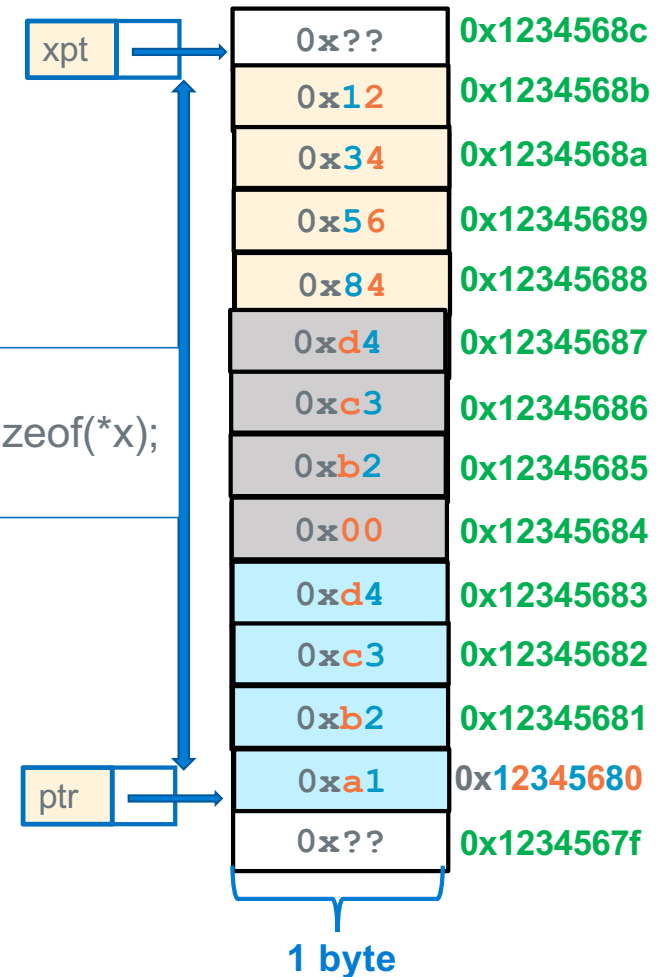
C Strings - 1

- **C does not** have a **dedicated type** for strings

- **Strings are** an **array of characters terminated by** a sentinel termination **character**

- **'\0'** is the **Null termination character;** has the **value of zero (do not confuse with '0')**

- An **array of chars** contains **a string only when** it is terminated by a '\0'

- **Length of a string** is the number of characters in it, not including the '\0'

- Strings in C are **not** objects

  - No embedded information about them, you just have a name and a memory location
  - You cannot use **+** or **+=** to concatenate strings in C
  - For example, you must **calculate string length** using code at runtime looking for the end

length of the string: number of char (= 5)

index	0	1	2	3	4	5
char	'H'	'e'	'l'	'l'	'o'	'\0'

length (size) of the array in number of char (= 6)

x
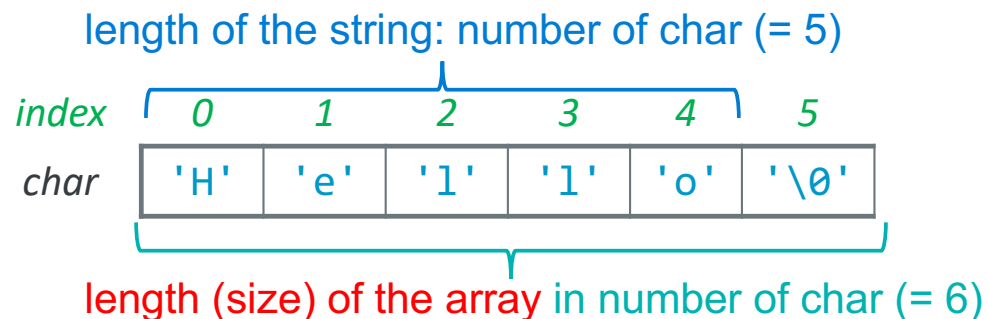
C Strings - 2

- **First `'\0'` encountered from the start of the string** always indicates the end of a string

- The **`'\0'` does not have to be** in the **last element in the space allocated to the array**
  - But, String length is always less than the size of the array it is contained in

- In the example below, the array buf contains two strings
  - One string starts at &(buf[0]) is "cat" with a string length of 3
  - The other string starts at &(b[4]) is "o" with a string length of 1
  - "o" has two bytes: 'o' and '\0'

string length: number of char (= 3)   string length: number of char (= 1)

index	0	1	2	3	4	5
buf	'c'	'a'	't'	'\0'	'o'	'\0'
	0x63	0x61	0x74	0x00	0x6f	0x00

length (size) of the array in number of char (= 6)

X
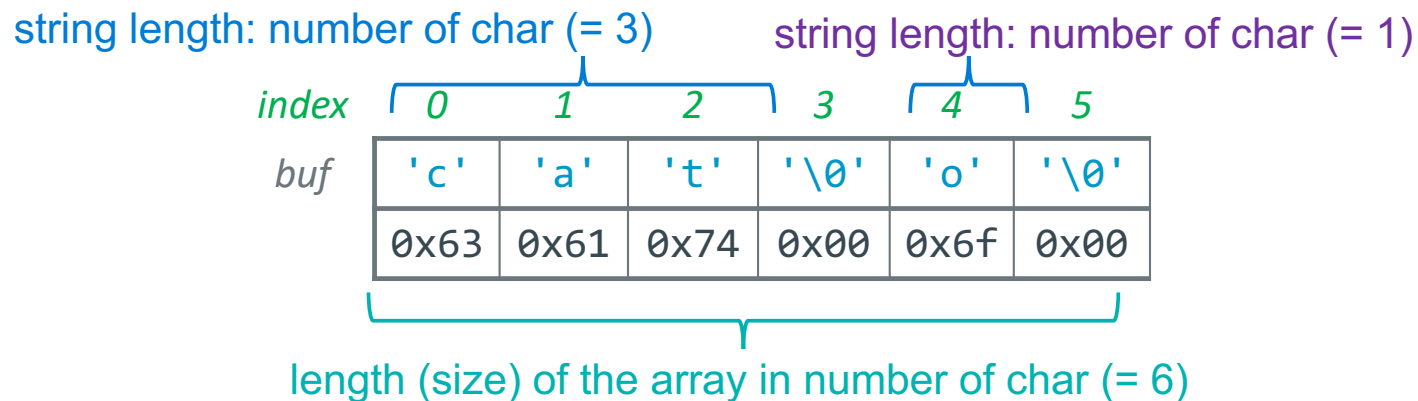
Defining Strings: Initialization

- When you combine the automatic length definition for arrays with double quote(")
  initialization
  - Compiler automatically adds the null terminator '\0' for you

```
char a[4] = {'c', 'a', 't', '\0'};
char b[] = "cat";                         // compiler calculates size, adds '\0'
char c[] = {'c', 'a', 't', '\0', 'a, 'b'}; // array size 6, string length 3
char empty[] = "";                        // empty string – contains '\0'
                                          // string length = 0
```

X

Defining Strings: Initialization Equivalents

- Following definitions create **equivalent** 4-character arrays
  - These are all strings as they all include a null ('\0') terminator

```
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};        // missing initial value defaults to 0
char d[4] = { 99, 97, 116, 0};      // 99 = 'c', 97 = 'a', 116 = 't'
char e[4] = "cat";
char f[4] = "cat\0";                // literal has 5 chars; array f string
                                    // length is 3
```

X

Background: Different Ways to Pass Parameters

- **Call-by-reference (or pass by reference)**
  - Parameter in the called function is an **_alias_** (references the same memory location) for the supplied argument
  - Modifying the parameter modifies the calling argument

**Call-by-value** (or pass by value) (C)
  - What **Called** Function Does
    - Passed Parameters are used like local variables
    - Modifying the passed parameter in the function is allowed just like a local variable
    - So, writing to the parameter, **_only_** changes the **_copy_**
- The return value from a function in C is **by value**

X

Passing Parameters – Call by Value Example

```c
int main(void)
{
    int x = 5;
    inc(x);              // makes a copy of x
    printf("%d\n", x); // 5 or 6 ?

}

void inc(int i)        // i is local to inc
{
    ++i;
}
```

if this was an expression like inc(x+1) it evaluates and stores the result in the memory allocated for the copy

different memory locations

x   scope main()

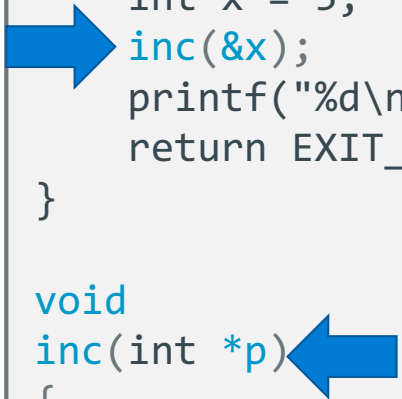copy of contents

i   scope inc()
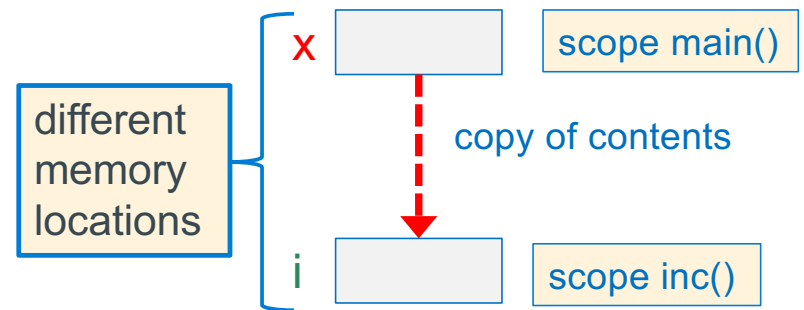
- when `inc(x)` is called, a copy of x is made to another memory location
  - `inc()` cannot change the variable x since `inc()` does not have the address of x, it is local to `main()` so, 5 is printed
- The `inc()` function is free to change it's copy of the argument (just like any local variable) remember it does <u>NOT</u> change the parameter in `main()`

x

Output Parameters

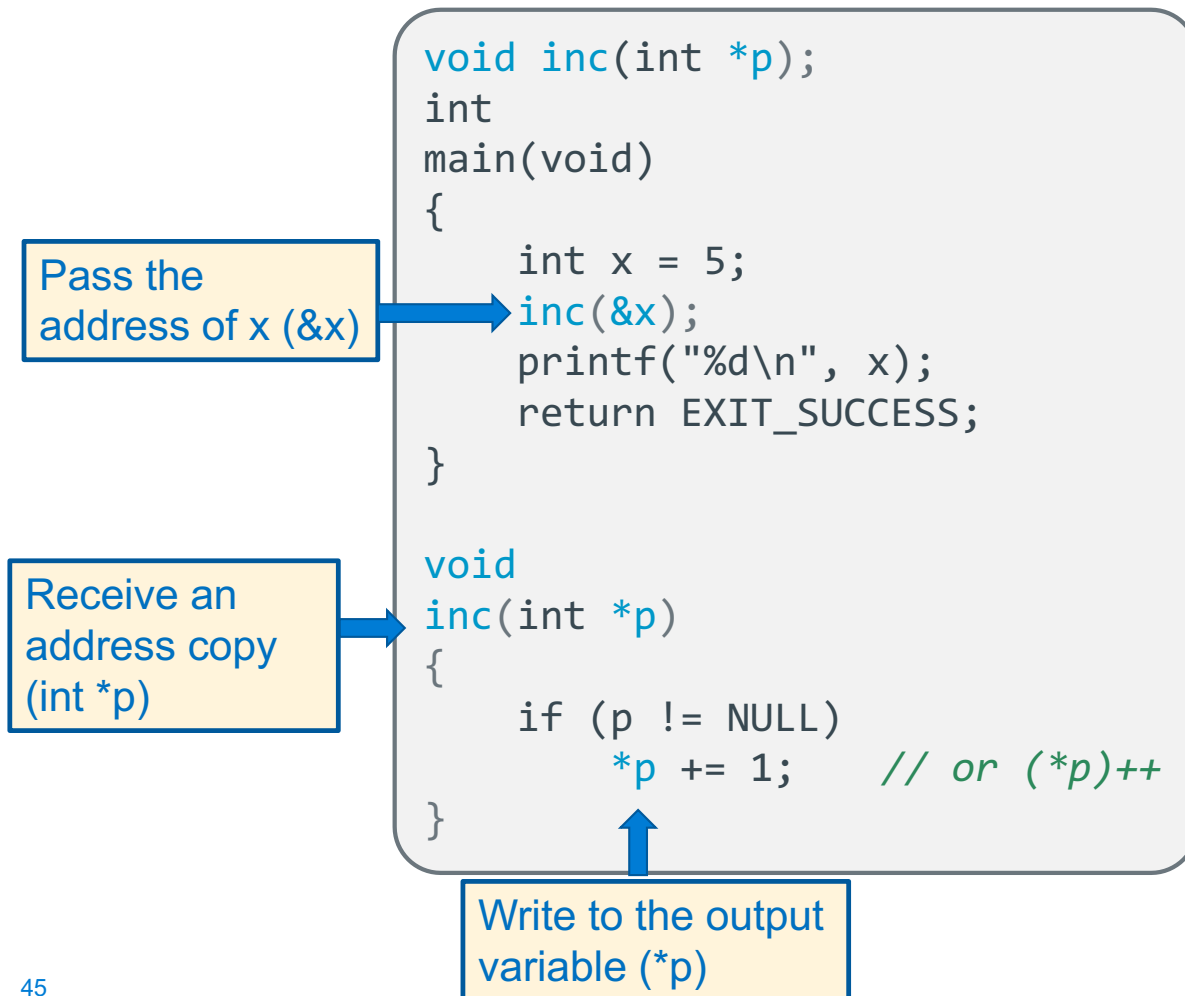- Passing a pointer parameter with the **<u>intent</u>** that the called function will use the address it to store values for use by the calling function, then pointer parameter is called an **output parameter**

- To pass the address of a variable x use the **address operator** (&x) **or** the contents of a pointer variable that points at x

- To be receive an address in the called function, define the corresponding parameter type to be a pointer
  - It is common to describe this method as: "pass a pointer to x

- C is still using "*pass by value*"
  - we pass the **value** of the address/pointer in a **parameter copy**
  - **The called routine** uses the address to change a variable in the caller's scope

```c
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);
    printf("%d\n", x);
    return EXIT_SUCCESS;
}

void
inc(int *p)
{
    if (p != NULL)
        *p += 1; // or (*p)++
}
```

X

Example Using Output Parameters

```
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);
    printf("%d\n", x);
    return EXIT_SUCCESS;

}

void
inc(int *p)
{
    if (p != NULL)
        *p += 1;    // or (*p)++
}
```

Pass the address of x (&x)

Receive an address copy (int *p)

Write to the output variable (*p)

At the Call to inc() in main()

1. Allocate space for p

2. Copy x's address into p

x	6	0x1014
		0x1010
		0x100c
		0x1008
		0x1004
p	0x1014	0x1000

Word address
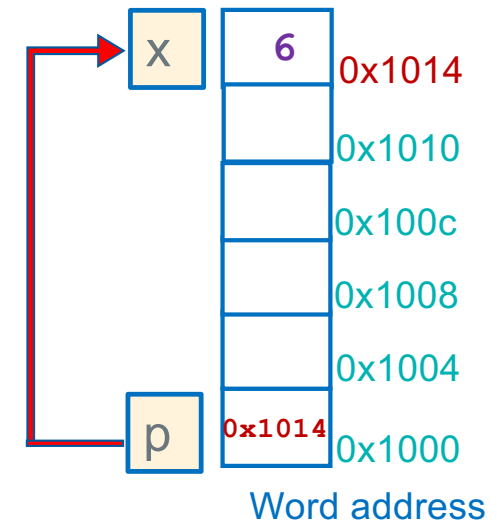
With a pointer to X,

inc() can change x in main()

this is called a side effect

p just like any other local variable

X

Array Parameters: Call-By-Value or Call-By-Reference?

- `Type[]` array parameter is automatically "**promoted**" to a pointer of type `Type *`, and a <u>**copy**</u> of the *pointer* is *passed by value*

```c
int main(void)
{
    int numbers[] = {9, 8, 1, 9, 5};

    passa(numbers);
    printf("numbers size:%lu\n", sizeof(numbers)); // 20
    return EXIT_SUCCESS;
}
```

```c
void passa(int a[])
{
    printf("a size:%lu\n", sizeof(a)); // 4
    return;
}
```

IMPORTANT:
See the size difference 20 in main() in passa() is 4 bytes (size of a pointer)

- Call-by-value pointer (callee can change the pointer parameter to point to something else!)

- Acts like call-by-reference (called function can change the contents caller's array)

46

X

Arrays As Parameters: What is the size of the array?

- It's tricky to use arrays as parameters, as **they are passed as pointers to the start of the array**
  - In C, **Arrays do not know their own size** and at runtime there is no "bounds" checking on indexes

```
int sumAll(int a[]);

int main(void)
{
  int numb[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numb);

  return EXIT_SUCCESS;
}

int sumAll(int a[])
{
  int i, sum = 0;
  int sz = (int) (sizeof(a)/sizeof(*a));
  for (i = 0; i < sz; i++) // this does not work
      sum += a[i];
  }
}
```

the name is the address, so this is passing a pointer to the start of the array

"inside" the body of sumAll(), the question is: how big is that array? all I have is a POINTER to the first element.....
sz is a 1 on 32 bit arm
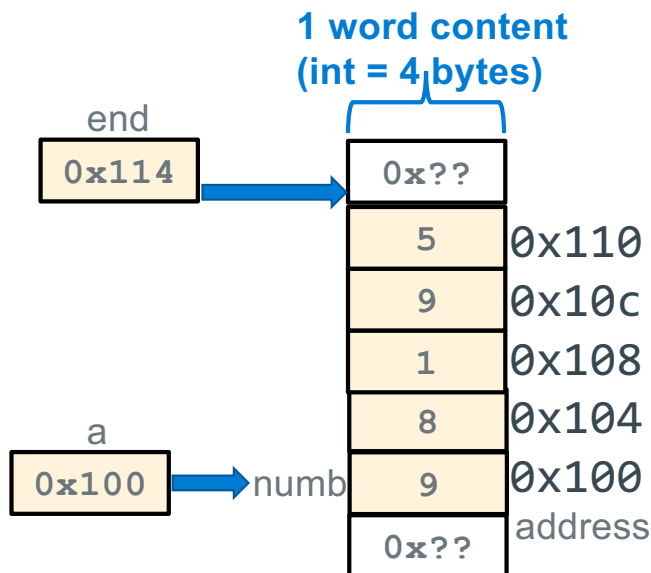
47

X

Arrays As Parameters, Approach 1: Pass the size

Two ways to pass array size

1. pass the count as an additional argument

2. add a sentinel element as the last element

remember you can only use sizeof() to calculate element count where the array is <u>defined</u>

```c
int sumAll(int *a, int size);
int main(void)
{
  int numb[] = {9, 8, 1, 9, 5};
  int cnt = sizeof(numb)/sizeof(numb[0]);

  printf("sum is: %d\n", sumAll(numb, cnt););
  return EXIT_SUCCESS;
}
```

1 word content
(int = 4 bytes)

end
0x114

0x??	
5	0x110
9	0x10c
1	0x108
8	0x104
9	0x100
0x??	address

a
0x100 → numb

```c
int sumAll(int *a, int size)
{
  int *end = a + size;
  int sum = 0;

  while (a < end)
    sum += *a++;
  return sum;
}
```

```c
int sumAll(int *a, int size)
{
  int sum = 0;

  for (int i= 0; i < size; i++)
    sum += a[i]; // *(a + i)
  return sum;
}
```
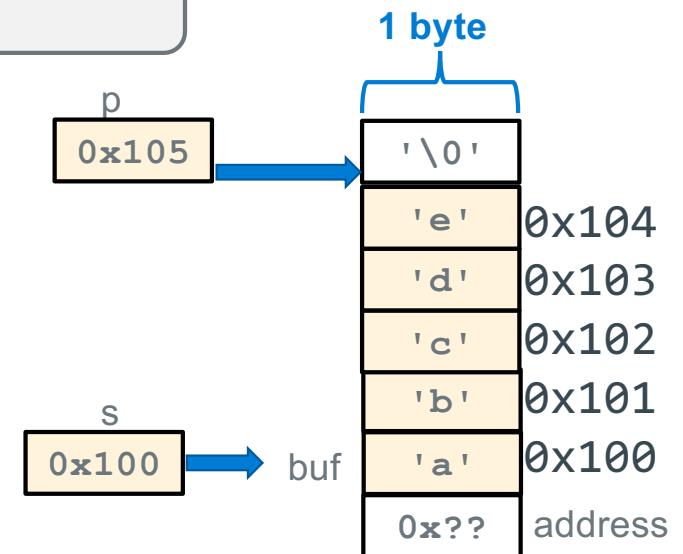
48

X

Arrays As Parameters, Approach 2: Use a sentinel element

- A sentinel is an element that contains a value that is not part of the normal data range
  - Forms of 0 are often used (like with strings). Examples: '\0', NULL

```c
int strlen(char *a);
int main(void)
{
  char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // string

  printf("Number of chars is: %d\n", my_strlen(buf));
  return EXIT_SUCCESS;
}
```

```c
/* Assumes parameter is a terminated string */
int strlen(char *s)
{
  char *p = s;
  if (p == NULL)
      return 0;
  while (*p)
      p++;
   return (p – s);
}
```

1 byte

p
0x105

'\0'	
'e'	0x104
'd'	0x103
'c'	0x102
'b'	0x101

s
0x100

buf

'a'	0x100
0x??	address

X

Do not overuse strlen()

- C string library function `strlen()` calculates string length **at runtime**

- **Do not overuse strlen(), as it walks the array each time called**

```c
int count_e(char *s)  // o(n²) !!!
{
    int count = 0;
    if (s == NULL)
        return 0;
    for (int j = 0; j < strlen(s); j++) {
        if (s[j] == 'e')
            count++
    }
    return count ;
}
```

```c
int count_e(char *s)  // o(n) !!!
{
    int count = 0;
    if (s == NULL)
        return 0;
    while (*s) {
        if (*s++ == 'e')
            count++
    }
    return count ;
}
```

x

Comparing stings

- Characters can be easily compared (c1 < c2) as they are numbers, so the **character order** is determined by the ASCII values assigned to each character
  - 65 = A   66 = B   67 = C   68 = D   69 = E   70 = F   71 = G, and so on.

- Example: the following strings are in lexicographical (alphabetical) order:

  "" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"

- Compare two strings lexicographically (i.e., comparing ASCII values), subtract one from the other

Return Value	Comparison
< 0	s1 < s2
> 0	s1 > s2
= 0	s1 == s2

```c
int strcmp(char *s1, char *s2)
{
    while (*s1 == *s2) {
        if ((*s1 == '\0') || (*s2 == '\0'))
            break;
        s1++;
        s2++;
    }
    return *s1 - *s2;  // character difference
}
```

X

Copying Strings: Use the Sentinel; libc: strcpy(), strncpy()

- To copy an array, you must copy each character from source to destination array
- Watch overwrites: strcpy assumes the target array size is equal or larger than source array

index	0	1	2	3	4	5
char	'H'	'e'	'l'	'l'	'o'	'\0'

```
char str1[80];
strcpy(str1, "hello");
```

```
// strncpy adds a length limit on copy
char str1[6];
strncpy(str1, "hello", 5); // \0 not copied
str1[5] = '\0'; // make sure \0 terminated
```

```
char *strcpy(char *s0, char *s1)
{
    char *str = s0;

    if ((s0 == NULL) || (s1 == NULL))
        return NULL;
    while (*s0++ = *s1++)
        ;
    return str;

}
```

```
char *strncpy(char *s0, char *s1, int len)
{
    char *str = s0;
    if ((s0 == NULL) || (s1 == NULL))
        return NULL;

    while ((*s0++ = *s1++) && --len)
        ;
    return str;

}
```

X

Reference: Some String Routines in libc (#include <string.h>)

Function	Description
strlen(*str*)	returns the # of chars in a C string (before null-terminating character).
strcmp(*str1, str2*), strncmp(*str1, str2, n*)	compares two strings; returns 0 if identical, <0 if **str1** comes before **str2** in alphabet, >0 if **str1** comes after **str2** in alphabet. **strncmp** stops comparing after at most **n** characters.
strchr(*str, ch*) strrchr(*str, ch*)	character search: returns a pointer to the first occurrence of **ch** in **str**, or **NULL** if **ch** was not found in **str**. strrchr find the last occurrence.
strstr(*haystack, needle*)	string search: returns a pointer to the start of the first occurrence of **needle** in **haystack**, or **NULL** if **needle** was not found in **haystack**.
strcpy(*dst, src*), strncpy(*dst, src, n*)	copies characters in **src** to **dst**, including null-terminating character. Assumes enough space in **dst**. Strings must not overlap. **strncpy** stops after at most **n** chars, and does not add null-terminating char.
strcat(*dst, src*), strncat(*dst, src, n*)	concatenate **src** onto the end of **dst**. **strncat** stops concatenating after at most **n** characters. Always adds a null-terminating character.
strspn(*str, accept*), strcspn(*str, reject*)	**strspn** returns the length of the initial part of **str** which contains only characters in **accept**. **strcspn** returns the length of the initial part of **str** which does not contain any characters in **reject**.

2D Array of Char (where elements may contain strings)

- 2D array of chars (where rows may include strings)

- Each row has the same fixed number of memory allocated

- All the rows are the same length regardless of the actual string length)

- The column size must be large enough for the longest string

high
memory

```
char aos2d[3][22] = {"my", "two dimensional", "char array"};
```

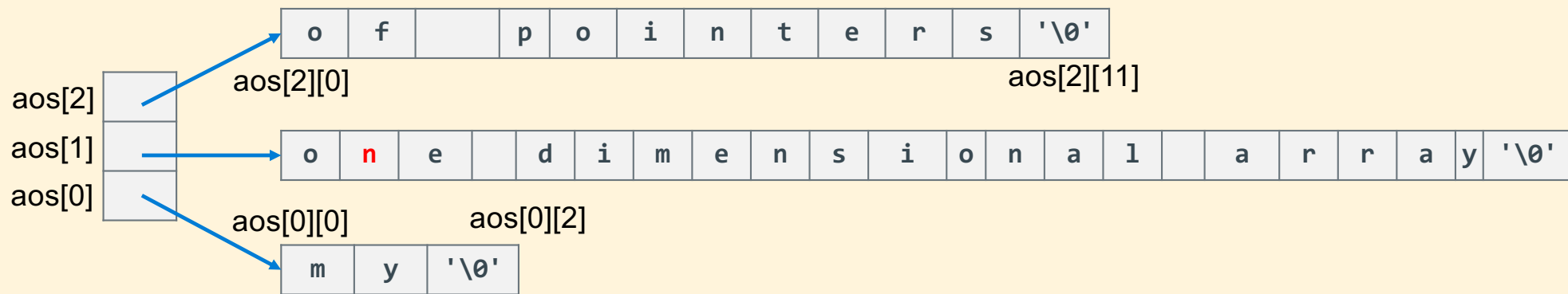|																		
aos2d[2]	c	h	a	r		a	r	r	a	y	'\0'							
aos2d[1]	t	w	o		d	i	m	e	n	s	i	o	n	a	l	a	r	r	a	y	'\0'
aos2d[0]	m	y	'\0'															

low
memory

high
memory

```
#define ROWS 3
char aos[ROWS][22] = { "my", "two dimensional", "char array"};
char (*ptc)[22] = aos;  // ptr points at a row of 22 chars

for (int i = 0; i < ROWS; i++)
    printf("%s\n", *(ptc + i));
```

x

Pointer Array to Strings (This is NOT a 2D array)

- 2D char arrays are an inefficient way to store strings (wastes memory) unless all the strings are similar lengths, so 2D char arrays *are rarely used* with string elements

- **An array of pointers** is common for strings as *"rows"* can very in length

o	f		p	o	i	n	t	e	r	s	'\0'

aos[2][0]   aos[2][11]

aos[2]

aos[1]

aos[0]

o	n	e		d	i	m	e	n	s	i	o	n	a	l		a	r	r	a	y	'\0'

aos[0][0]   aos[0][2]

m	y	'\0'

- `aos` is an array of pointers; each pointer points at a character array (also a string here)

- Not a 2D array, but any char can be accessed as if it was in a 2D array of chars
  - When I was learning, this was the most confusing syntax aspects of C!

X

main() Command line arguments: argc, argv

- Arguments are passed to main() as a pointer to an array of pointers (`**argv or *argv[]`)

  Conceptually: % `*argv[0] *argv[1] *argv[2] ….`

- `argc` is the number of VALID elements (they point at something)

- `*argv (argv[0])` is **usually** is the name of the executable file (% ./vim file.c)

- `*(argv + argc)` always contains a NULL (0) sentinel

- `*argv[] (or **argv)` elements point at **mutable strings**!

argv+3 NULL

argv+2

argv+1

argv+0

argv[2][0]

| 3 | \0 |

.% ./extract –c 3
(space between c and 3)
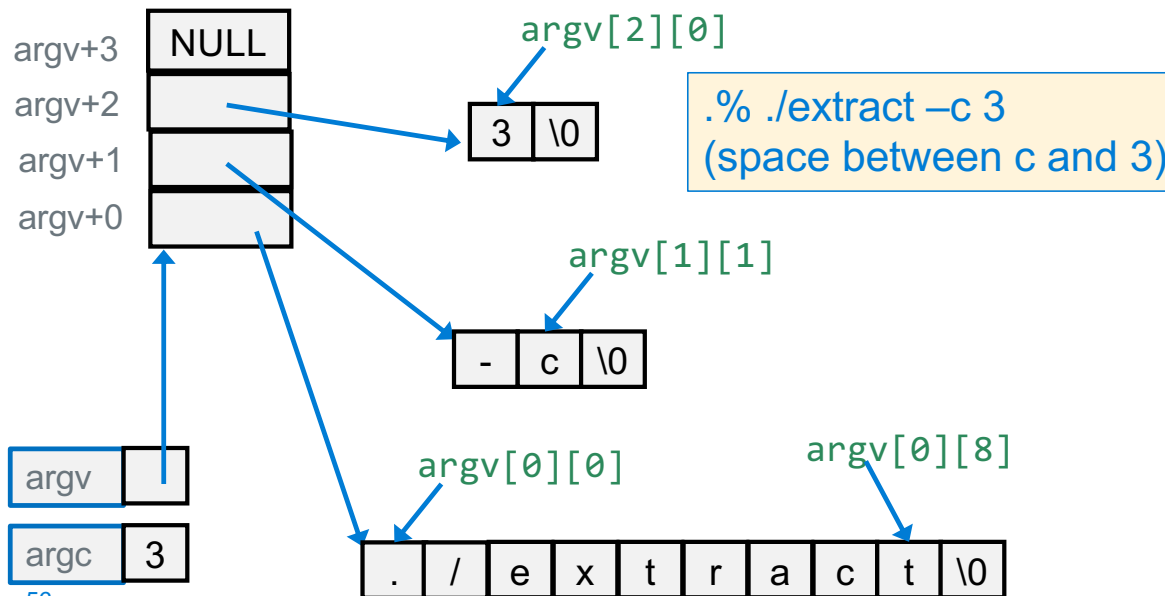
```
printf("%s\n", *(argv+0));
printf("%s\n", *(argv+1));
printf("%s\n", *(argv+2));
```

argv[1][1]

| - | c | \0 |

argv

argc 3

argv[0][0]

argv[0][8]

| . | / | e | x | t | r | a | c | t | \0 |

X

main() Command line arguments: argc, argv

argv+3 — NULL

argv+2

argv+1

argv+0

`argv[2][0] or **(argv + 2)`

`3` `\0`

.% ./extract –c 3
(space between c and 3)

`argv[1][1] or *(*(argv + 1)+1)`

`-` `c` `\0`

argv

argc — `3`

`argv[0][8]`

`argv[0][0]`

`.` `/` `e` `x` `t` `r` `a` `c` `t` `\0`
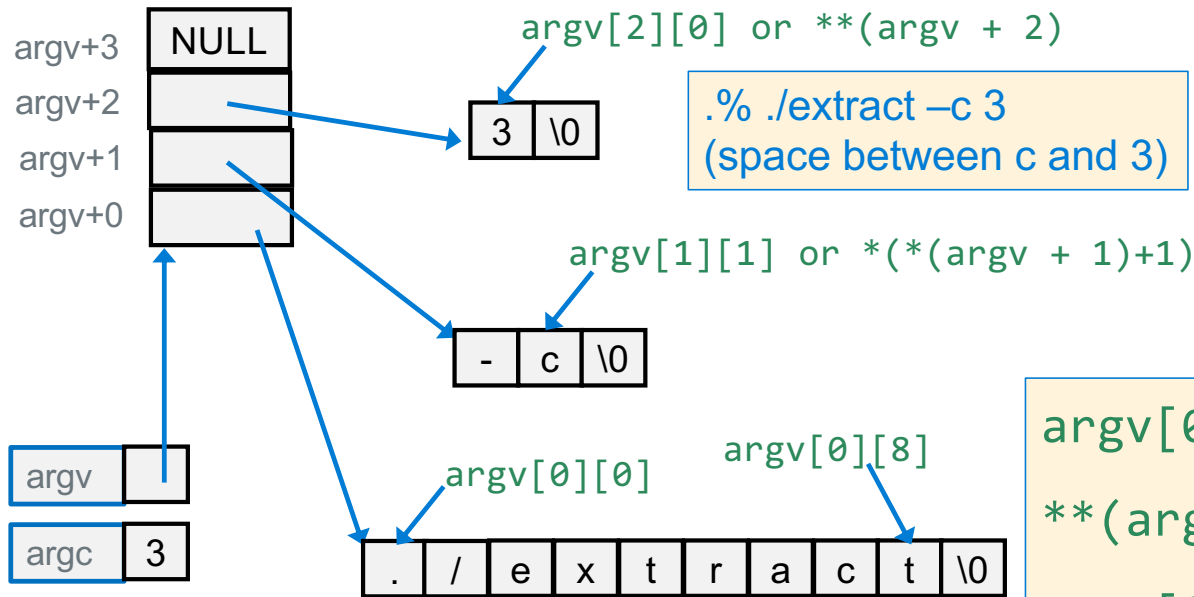
```
argv[0][0] equiv to **(argv+0)

**(argv+0) equiv **argv

argv[0][8] equiv *(*argv + 8)


char *pt = *argv;

*pt equiv to **(argv+0)

*(pt+8) equiv to *(*argv + 8)
```

X

main() Command line arguments: argc, argv

argv+3 | NULL

argv+2

argv+1

argv+0

`argv[2][0] or **(argv + 2)`

| 3 | \0 |

.% ./extract –c 3
(space between c and 3)

`argv[1][1] or *(*(argv + 1)+1)`

| - | c | \0 |

argv

argc | 3

`argv[0][0]`

`argv[0][8]`

| . | / | e | x | t | r | a | c | t | \0 |
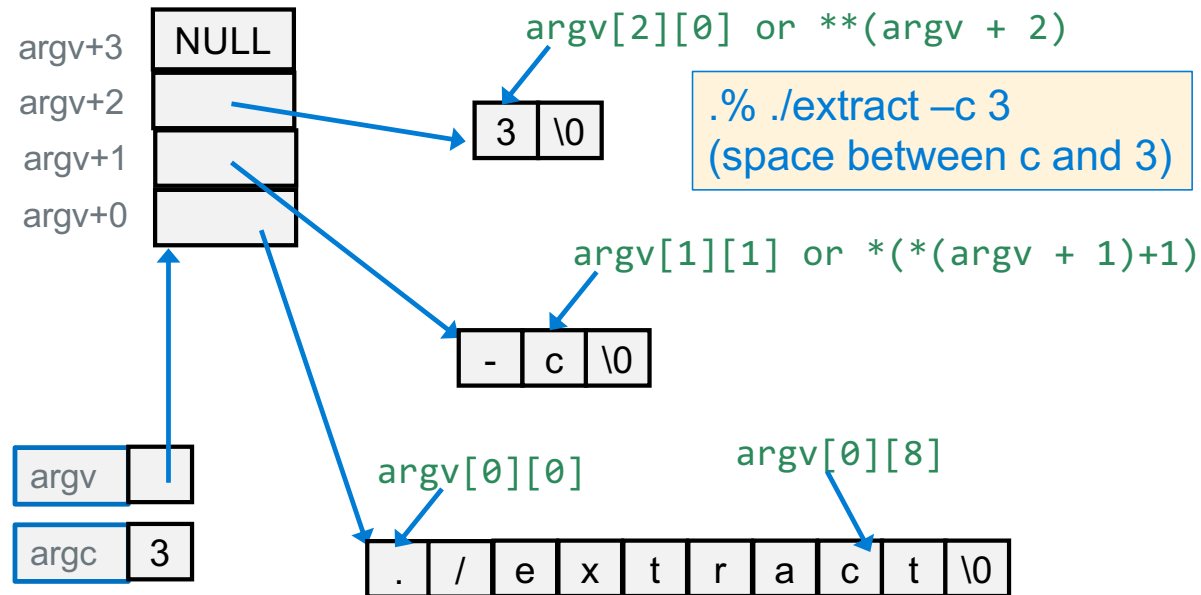
```
int main(int argc, char *argv[])
{
    for (int i = 0; argv[i] != NULL; i++) {
        for (int j = 0; argv[i][j] != '\0'; j++)
            putchar(argv[i][j]);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```
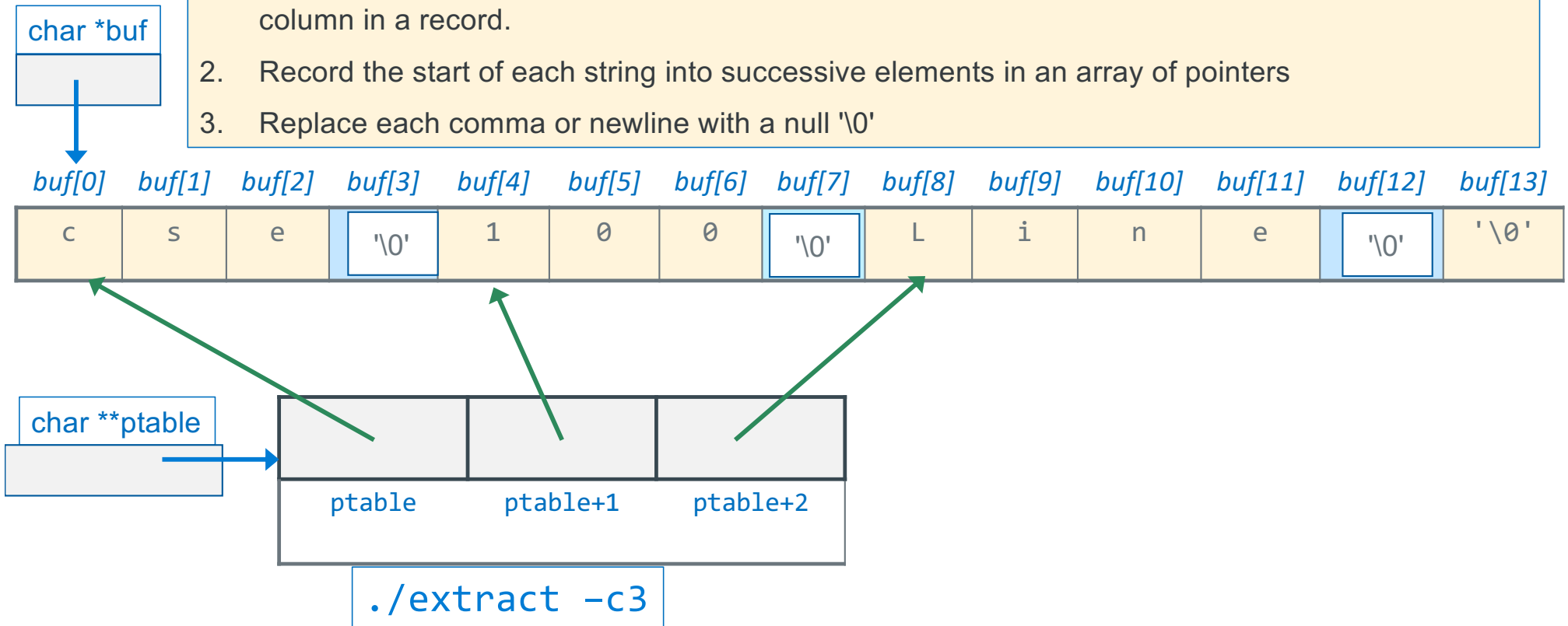
```
int main(int argc, char **argv)
{
    char *pt;
    while ((pt = *argv++) != NULL) {
        while (*pt != '\0')
            putchar(*pt++);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

x

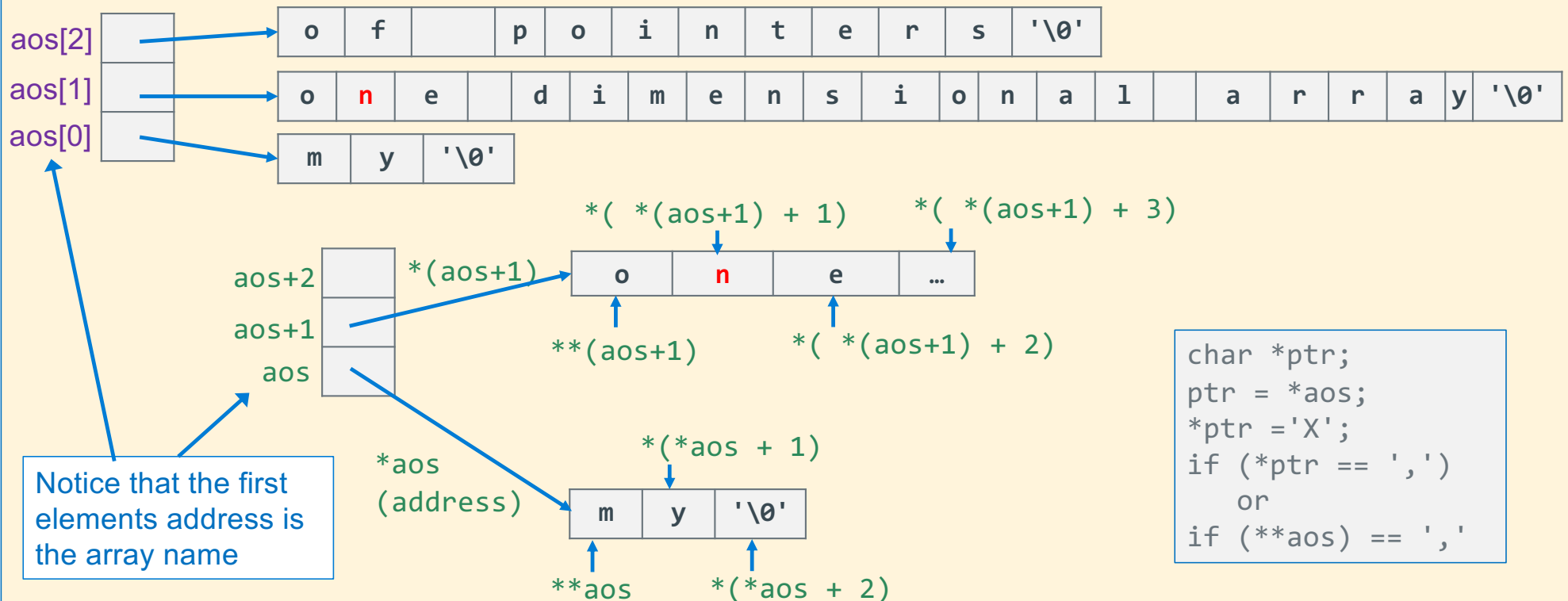# PA4: Creating a 2D Array of Mutable String Pointers

char *buf

1. Break a string of comma separated words into individual strings without copying. Do This by walking the string until you see an either a comma , or a newline \n. Each points at a field or column in a record.

2. Record the start of each string into successive elements in an array of pointers

3. Replace each comma or newline with a null '\0'

buf[0]	buf[1]	buf[2]	buf[3]	buf[4]	buf[5]	buf[6]	buf[7]	buf[8]	buf[9]	buf[10]	buf[11]	buf[12]	buf[13]
c	s	e	'\0'	1	0	0	'\0'	L	i	n	e	'\0'	'\0'

char **ptable

ptable	ptable+1	ptable+2

`./extract -c3`

X

Review: Pointer Array to Strings

How to access: `aos[1][1]` is `*(*(aos + 1) + 1)` which contains '**n**'
its address is `(*(aos + 1) + 1)`

```
char *ptr;
ptr = *aos;
*ptr ='X';
if (*ptr == ',')
    or
if (**aos) == ','
```

Notice that the first elements address is the array name

60

X

strtol() and strtoul() examples of passing a pointer to a pointer

```
long int strtol(const char *str, char **endptr, int base);

unsigned long int strtoul(const char *str, char **endptr, int base);
```

reruns the string converted to a long or unsigned long

str pointer to the string to convert

endptr pass the address of a variable that is a char pointer (output variable)

base: number base of the integral value

- **Example**: string is to contain just positive numbers >= 0 (in ascii) with no extra stuff

- If the string is not valid, then
  - `*endptr != '\0'` then string contains more than just numbers (bad input)
  - `*endptr` stores the address of the first invalid character found in the buffer pointed (`str`)

- How to use endptr when it <u>does</u> <u>not</u> contain NULL:
  - If there are other conversion errors (you can read the man page) then errno != 0
  - When conversion is ok, `errno` is unaltered (always clear it before calling these routines)

X

strtol() and strtoul() examples of passing a pointer to a pointer

```
#include <stdlib.h>
#include <errno.h>
char *endptr;
char buf[] = "33";  // test buffer string
int number;

errno = 0; // set errno to 0 (zero) before each call
number = (int)strtol(buf, &endptr, 10)
// check if the string was a proper number
// *entpr should be at the end of the string == '\0'

if ((*endptr != '\0') || (errno != 0)) {
    // handle the error
}
printf("%d\n", number);
```

X

String Literals (Read-Only) in Expressions

- When strings in quotations (*e.g.,* "string") are **part of** an **expression** (*i.e., not* part of an *array initialization*) they are called *string literals*

  ```
  printf("literal\n");
  printf("literal %s\n", "another literal");
  ```

- What is a *string literal:*
  - Is a null-terminated string in a **const char array**
  - Located in the **read-only data** segment of memory
  - Is not assigned a variable name by the compiler, so it is only accessible by the location in memory where it is stored

- **String literals** are a type of *anonymous variable*
  - Memory containing data without a name bound to them (only the address is known)

- The *string literal* in the printf()'s, are replaced with the starting address of the corresponding array (first or [0] element) when the code is compiled

X

String Literals, Mutable and Immutable arrays

```
char mess1[] = "Hello World";
char *ptr = mess1;
*(ptr + 5) = '\0'; // shortens string to "Hello"
```

- mess1 is a **mutable** array (type is char [ ]) with enough space to hold the string + '\0'
  - You **can change** array contents

```
char *mess2 = "Hello World";  // "Hello World" is a string literal
                              // mess2 is a pointer NOT an array!
```

- In the example above, "Hello World" is immutable string literal (array)
  - "Hello World" is not associated with a variable name; anonymous variable
  - "Hello World" has space to hold the string + '\0'
  - "Hello World" is read only (immutable) and cannot be modified at runtime
- mess2 is a **pointer** to an **immutable** array with space to hold the string + '\0'

x

Be Careful with C Strings and Arrays of Chars

mess2 **pointer** to an **immutable** array with space to hold the string + '\0'

- you **cannot change** array contents, but you can change what mess2 points at
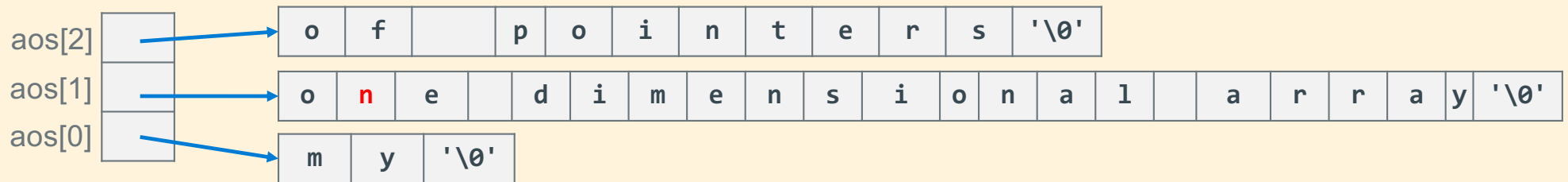
```
char *mess2 = "Hello World";   // "Hello World" is a string literal
                               // mess2 is a pointer NOT an array!
*mess = 'h';                   // undefined in C, linux seg fault
mess2 = mess1;                 // where mess2 points can be changed
```

- mess3 is an array but does not contain a '\0'
  - SO, IT IS **NOT** A VALID STRING

```
char mess3[] = {'H','e','l','l','o',' ','W','o','r','l','d'};
```

X

Pointer Array to Immutable Strings

```
char *aos[] = {"my", "one dimensional array", "of pointers"};
```

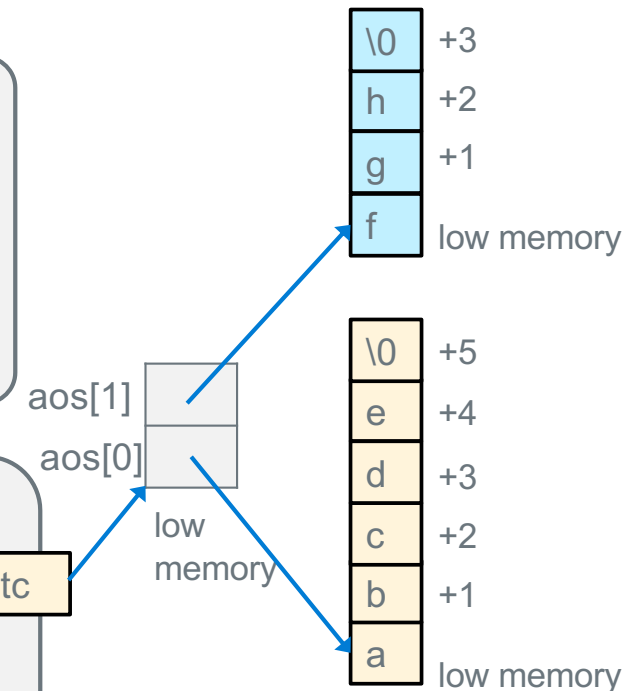| aos[2] | | → | o | f | | p | o | i | n | t | e | r | s | '\0' |
| aos[1] | | → | o | n | e | | d | i | m | e | n | s | i | o | n | a | l | | a | r | r | a | y | '\0' |
| aos[0] | | → | m | y | '\0' |

- **aos** is an array of pointers, with each pointer pointing to a read only string literal

Pointer Array to Mutable Strings

- Make an array of pointers to mutable strings requires using a cast to an array (char [ ])

- Add a NULL sentinel at the end to indicate the end of the array

```
char *aos[] = {
    (char []) {"abcde"},
    (char []) {"fgh"},
    (char *)  {NULL}
};
char **ptc = aos;
```

```
printf("%c\n", *(*(aos + 1) + 1));

while (*ptc != NULL) {
    printf("%s\n", *ptc);     // prints string

    for (int j = 0; *(*ptc + j); j++)
        putchar(*(*ptc + j)); // char in string

    putchar('\n');
    ptc++;
}
```

ptc

aos[1]

aos[0]

low memory

\0	+3
h	+2
g	+1
f	low memory

\0	+5
e	+4
d	+3
c	+2
b	+1
a	low memory

```
%./a.out
g
abcde
abcde
fgh
fgh
```

X

Extra Slides

C Precedence and Pointers

- ++ -- pre and post increment combined with pointers will create code that is complex, hard to read and difficult to maintain, so be careful!

- My advice: Always Use () to improve readability

```
int array[] = {2, 5, 7, 9, 11, 13};
int *ptr = array;
int x;
```

```
x = 1 + (*ptr++)++; // yuck!!
```

```
        2   1   3
```

```
/* Same as the one line above */
x = 1 + *ptr;    // x = 1 + *orig_ptr (2) = 3;

*ptr = *ptr + 1; //(*orig_ptr)++ is array[0]= 3;

ptr = 1 + ptr;    // ptr = &array[1] = points 5
```

69

Operator	Description	Precedence level	Associativity
( ) [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Dot operator (Member selection via object name) Arrow operator(Member selection via pointer) Postfix increment/decrement	1 highest	Left to Right
+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Prefix increment/decrement Logical NOT One's complement Indirection Address (of operand) Type cast Determine size in bytes on this implementation	2	Right to Left
* / %	Multiplication Division Modulus	3	Left to Right
+ -	Addition Subtraction	4	Left to Right
<< >>	Left shift Right shift	5	Left to Right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left to Right
== !=	Equal to Not equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
		Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
||	Logical OR	12	Left to Right
?:	Conditional operator	13	Right to Left
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

X

2D Arrays

- Generic (uniform) 2D array format:

  ```
  type name[rows][cols] = {{values},…,{values}};
  ```
  - allocates a single, <u>contiguous</u> block of memory
  - The array is organized in ***row-major*** format

```
// a 2-row, 3-column array of char
char matrix[2][3];

// a 2-row, 5-column (row length) array of ints
// Must specify row length, compiler counts rows

int grid[][5] = {
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9}
};

grid[1][2] using pointers is *( *(grid + 1) + 2)
```

[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[0][0]	[0][1]	[0][2]	[0][3]	[0][4]

1 word (int = 4 bytes)

|			
	?		high memory
grid[1][4]	9	0x0024
grid[1][3]	8	0x0020
grid[1][2]	7	0x001c
grid[1][1]	6	0x0018
grid[1][0]	5	0x0014
grid[0][4]	4	0x0010
grid[0][3]	3	0x000c
grid[0][2]	2	0x0008
grid[0][1]	1	0x0004
grid[0][0]	0	0x0000

low memory

X

2D Array Access

```c
#define LEN 6
int main(void)
{
    char a[][LEN] = {"abcde","fghij"};

    for (int i = 0; i < sizeof(a)/(sizeof(a[0][0]) * LEN); i++) {
        for (int j = 0; j < LEN; j++)
            putchar(a[i][j]);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

must supply ROW length (number of cols)

```
%./a.out
abcde
fghij
```
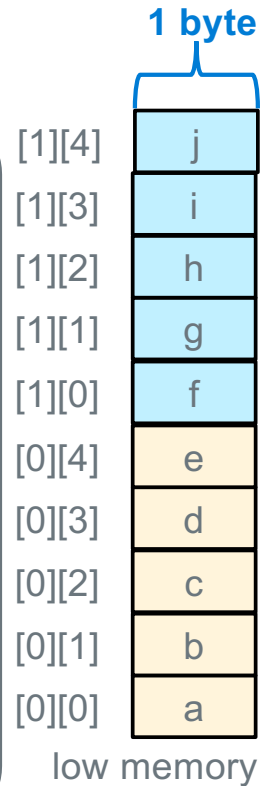
```c
char *ptc = &a[0][0]; // pointer to a char!
putchar( *(ptc + (i * sizeof(a[0][0]) * LEN) + j) );
```

1 byte

[1][4]	j
[1][3]	i
[1][2]	h
[1][1]	g
[1][0]	f
[0][4]	e
[0][3]	d
[0][2]	c
[0][1]	b
[0][0]	a

low memory

X

2D Array of Char (elements may contain strings)

- 2D array of chars (where rows may include strings) - Mutable

- Each row is the same fixed size of memory

- So, all the rows are the same length regardless of the actual string length

- The column size must be large enough for the longest string

high memory

```
char aos2d[3][22] = {"my", "two dimensional", "char array"};
```

aos2d[2]	c	h	a	r		a	r	r	a	y	'\0'	0	0	0	0	0	0	0	0	0	0	0
aos2d[1]	t	w	o		d	i	m	e	n	s	i	o	n	a	l		a	r	r	a	y	'\0'
aos2d[0]	m	y	'\0'	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

low memory

high memory

```
#define ROWS 3
char aos[ROWS][22] = { "my", "two dimensional", "char array"};
char (*ptc)[22] = aos;  // ptr points at a row of 22 chars

for (int i = 0; i < ROWS; i++)
    printf("%s\n", *(ptc + i) );
```

X