

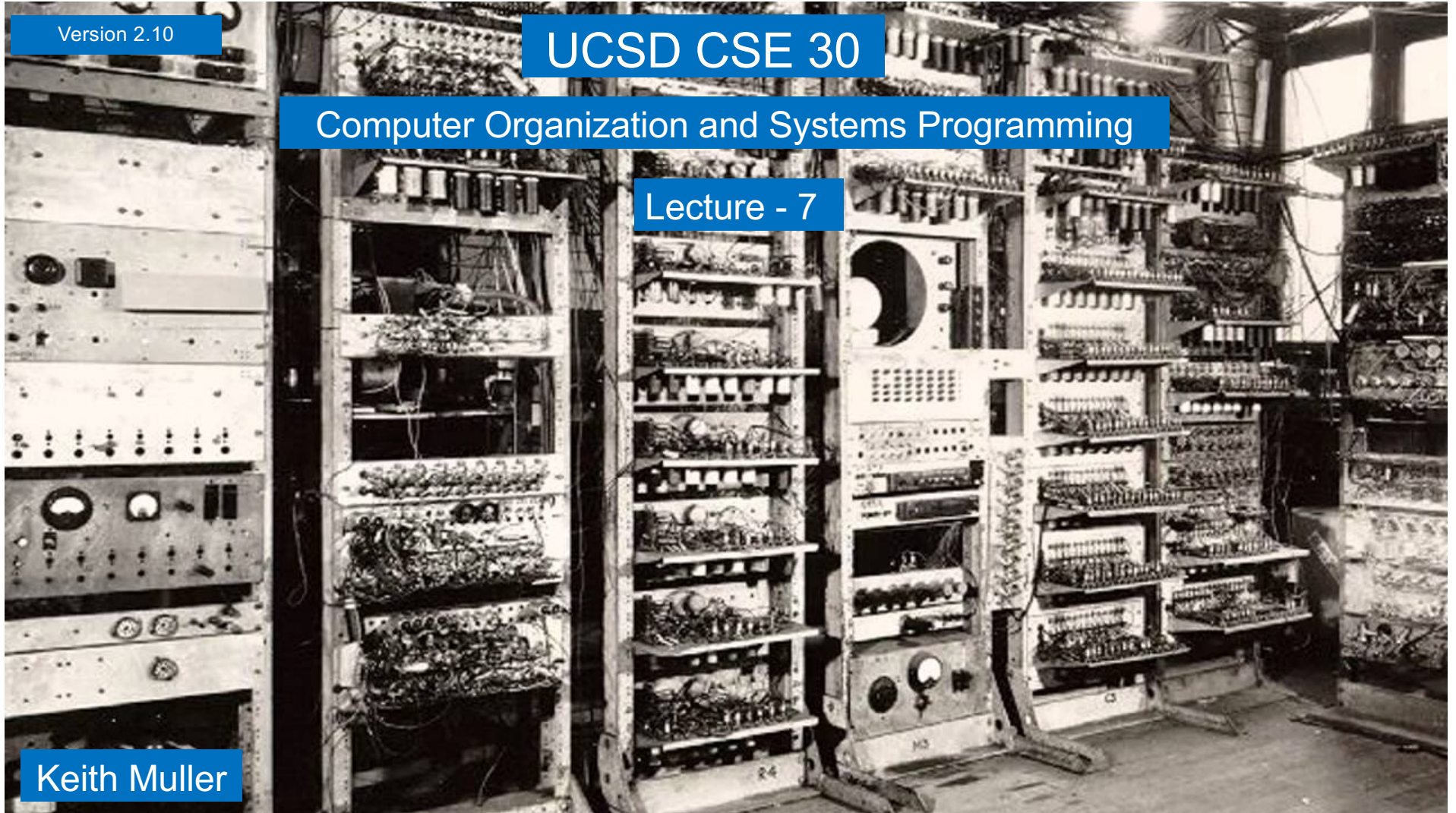
Version 2.10

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 7

Keith Muller





Fast Ways to Traverse an Array: Use a Limit Pointer

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

```
int *ptr;  
int *xptr;  
ptr = x;  
xptr = ptr + cnt;
```

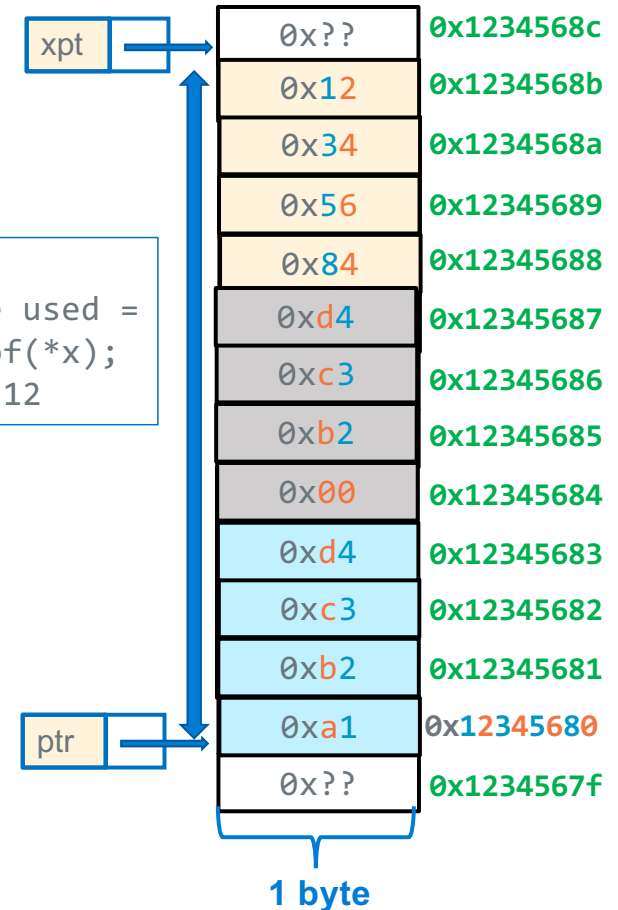
xptr is a **loop limit pointer**
it points **1 element past**
the end of the array

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

//or &x[0]

cnt = 3;
actual space used =
cnt * sizeof(*x);
= 12

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



C Precedence and Pointers

- ++ -- pre and post increment combined with pointers can create code that is complex, hard to read and difficult to maintain
- Use () to help readability

Operator	Description	Associativity
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
	Logical OR	left to right
?:	Ternary operator	right to left
= += -= *= /= %= &= ^= = <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

common	With Parentheses	Meaning
*p++	*(p++)	(1)The Rvalue is the object that p points at (2)increment pointer p to next element ++ is higher than *
(*p)++		(1)Rvalue is the object that p points at (2)increment the object
*++p	*(++p)	(1)Increment pointer p first to the next element (2)Rvalue is the object that the incremented pointer points at
++*p	++(*p)	Rvalue is the incremented value of the object that p points at

Example of a hard-to-understand pointer statement

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

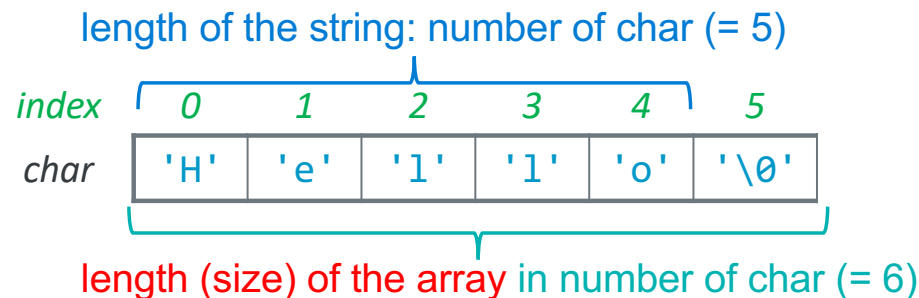
```
x = 1 + (*ptr++)++; // yuck!!
```

common	With Parentheses	Meaning
*p++	*(p++)	(1) The Rvalue is the object that p points at (2) increment pointer p to next element ++ is higher than *
(*p)++		(1) Rvalue is the object that p points at (2) increment the object
*++p	*(++p)	(1) Increment pointer p first to the next element (2) Rvalue is the object that the incremented pointer points at
++*p	++(*p)	Rvalue is the incremented value of the object that p points at

```
/* Same as the one line above */  
x = 1 + *ptr;           // x = 1 + *ptr (2) = 3;  
  
*ptr = *ptr + 1;        // (*ptr)++ is array[0]= 2 + 1;  
  
ptr = 1 + ptr;          // ptr = &array[1] = now points at 5
```

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 sentinel



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 (but only `cat` is seen as the string)
 - 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)

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


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)

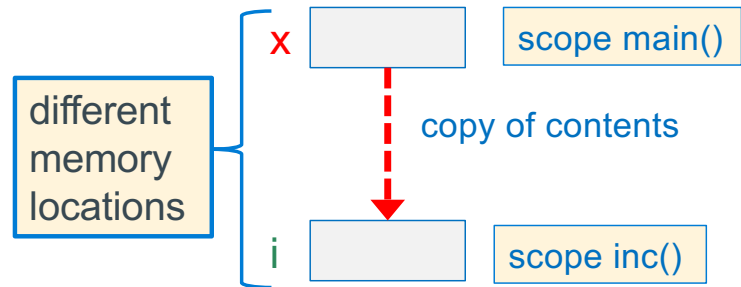
- What **Called** Function Does
 - Passed Parameters are used like local variables
 - **Modifying** the **passed parameter** in the **function** is **allowed** just like a **local variable**
 - So, writing to the parameter, **only** **changes** the **copy**
- The **return value** from a function in C is **by value**

Passing Parameters – Call by Value Example

```
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



- 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 its copy of the argument (just like any local variable) remember it does NOT change the parameter in `main()`

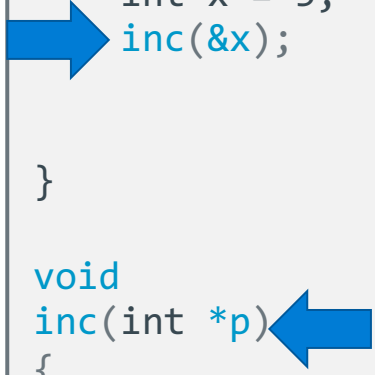
Output Parameters (Mimics Call by Reference)

- Passing a pointer parameter with the intent 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, or the name of an array (the arrays address)
- To be receive an address in the called function, define the corresponding parameter type to be a pointer (add *)
 - 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

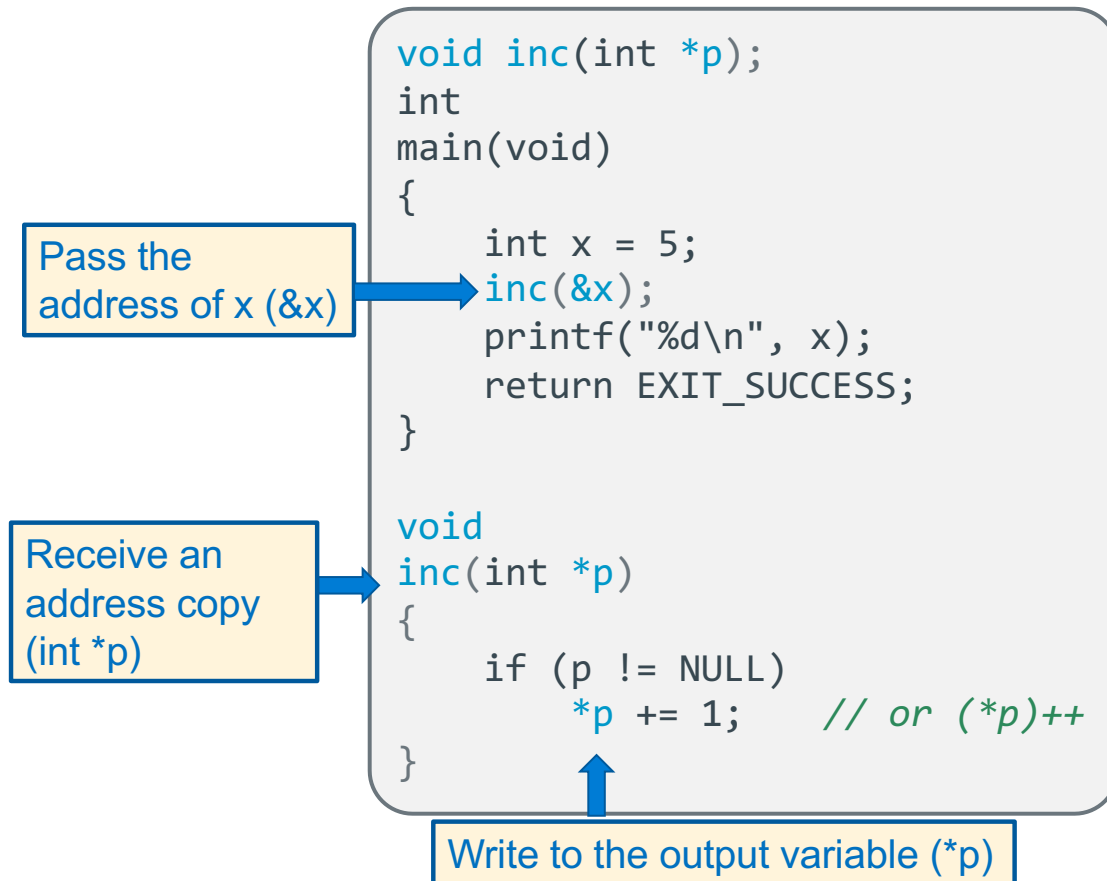
```
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);
}

void
inc(int *p)
{
}

}
```

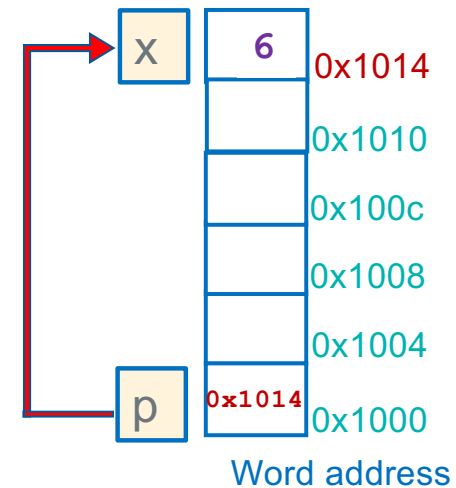


Example Using Output Parameters



At the Call to inc() in main()

1. Allocate space for p
2. Copy x's address into p



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 copy of the *pointer* is *passed by value*

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

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

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

```
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) on pi-cluster and 8 on ieng6

- 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)

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

“inside” the body of `sumAll()`, the question is:
how big is that array? all I have is a POINTER to the first element.....

so `sizeof(a)` is the size of a pointer, not the array it points at

Net result: `sz` is a 1 on picluster

```
int sumAll(int *);

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];
}
```

Arrays As Parameters, Approach 1: Pass the size

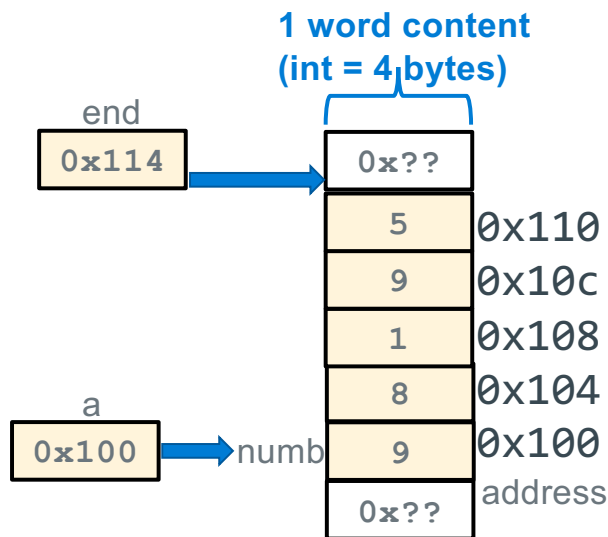
Two ways to pass array size

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 defined

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

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



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

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

same as:
sum = sum + *a;
a++;

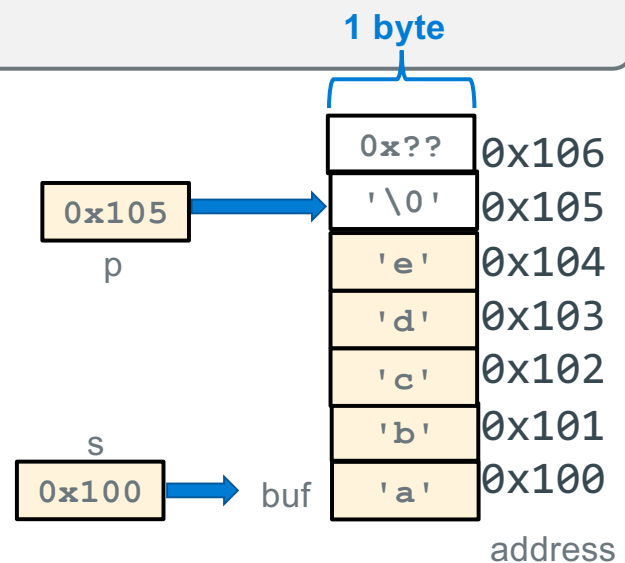
Arrays As Parameters, Approach 2: Use a sentinel element

- 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

```
int strlen(char *a); // returns number of chars in string, not counting \0
int main(void)
{
    char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // or buf[] = "abcde";

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

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



x

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

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

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**

```
int count_e(char *s) //  $O(n^2)$  !!!  
{  
    int count = 0;  
    if (s == NULL)  
        return 0;  
    for (int j = 0; j < strlen(s); j++) {  
        if (s[j] == 'e')  
            count++;  
    }  
    return count;  
}
```



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

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 */
```

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) ...
```

Simple String IO - Reading

Task	Example Function Calls
Read a string	<pre>#include <stdio.h> char *strptr; char myStr[BFSZ]; strptr = fgets(myStr, BFSZ, stdin);</pre> <div>must pass the size of the array so fgets() knows how much space there is</div>

`char *fgets(char array[], int size, FILE *stream)`

- **char *** is a pointer (address) to an **array of char**
- reads in at most **one less than size** characters from **stream** and stores them into **array**
- Reading stops after an **EOF** or a newline '\n'
 - If a newline ('\n') is read, it is stored into the buffer
 - **A terminating null byte ('\0') is always stored after the last character in the buffer**

t	h	i	s		i	s		a	s	t	r	i	n	g	\n	\0
---	---	---	---	--	---	---	--	---	---	---	---	---	---	---	----	----

- Returns a **NULL at end of file** (or a read failure), otherwise a pointer to array (pointers later...)
- See `man 3 fgets`

Pointer returns from a function call

```
char *next(char *ptr)
{
    if (ptr == NULL)
        return NULL;

    while ((*ptr != '\0') && (*ptr != ','))
        ptr++;

    if (*ptr == ',')
        return ++ptr;
    return NULL;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#define BUFSZ 512
char *next(char *);

int main()
{
    char buf[BUFSZ];
    char *ptr;

    while (fgets(buf, BUFSZ, stdin) != NULL) {
        printf("buf: %s\n", buf);

        if ((ptr = next(buf)) != NULL)
            printf("after: %s\n", ptr);
        else
            printf("no comma found\n");
    }
    return EXIT_SUCCESS;
}
```

Returning a Pointer To a Local Variable (Dangling Pointer)

- There are many situations where a function will return a pointer, but a function must never return a pointer to a memory location that is **no longer valid** such as:
 - Address of a **passed parameter copy** as the caller may or will deallocate it after the call
 - Address of a **local variable (automatic)** that is invalid on function return
- These errors are called a **dangling pointer**

n is a parameter with the scope of bad_idea it is no longer valid after the function returns

```
int *bad_idea(int n)
{
    return &n; // NEVER do this
}
```

a is an automatic (local) with a scope and **lifetime** within bad_idea2 a is no longer a valid location after the function returns

```
int *bad_idea2(int n)
{
    int a = n * n;
    return &a; // NEVER do this
}
```

```
/*
 * this is ok to do
 * it is NOT a dangling
 * pointer
 */

int *ok(int n)
{
    static int a = n * n;
    return &a; // ok
}
```

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

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

```
// strncpy adds a length limit on copy
char str1[6];
int cnt = (int)(sizeof(str1) / sizeof(str1[0]));

strncpy(str1, "hello", cnt); // \0 copied
strncpy(str1, "hello", cnt - 1); // \0 not copied
```

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

    while ((*s0++ = *s1++) && --len) //watch short circuit here
        ;
    return str;
}
```


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

String Literals, Mutable and Immutable arrays - 1

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

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

`mess1[]` Hello World\0

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

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

`mess2` → Hello World\0 ← read only string literal

- `mess3` is a **pointer** to a mutable array

```
char *mess3 = (char []) {"Hello World"}; // mutable string  
*(mess3 + 1) = '\0'; // ok
```

using the cast `(char [])`
makes it mutable

`mess3` → Hello World\0 ← mutable string

You cannot write to an immutable literal

- You can use & to get the address of an anonymous variable as shown
 - Though the Rvalue of "Hello" is the address
- You cannot write to an immutable literal

```
char *pt;  
  
// pt = "Hello";  
pt = (char *) &"Hello";  
*pt = 'a'; // bus error  
printf("%s\n", pt);
```

2D Arrays

- Generic (uniform) 2D array format:

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

- allocates a single, contiguous 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}  
};
```

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

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

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

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

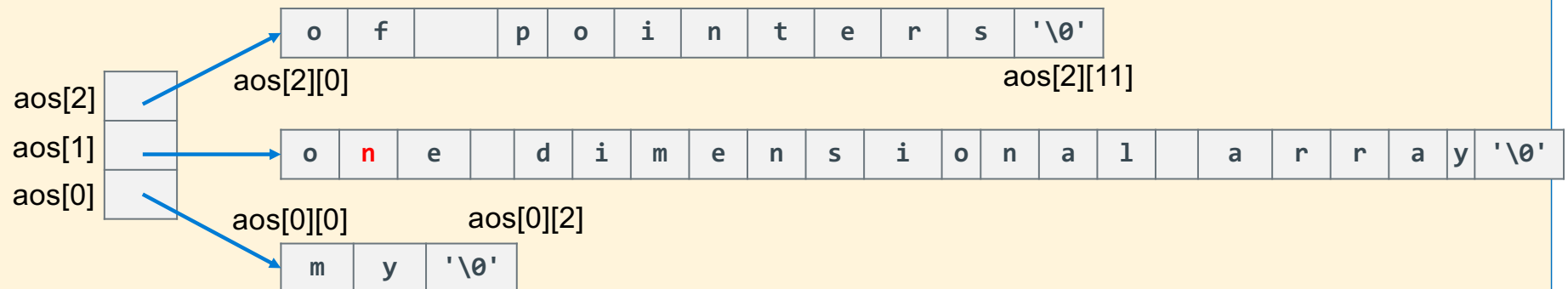
```
#define ROWS 3
char aos[ROWS][22] = { "my", "two dimensional", "char array"};
char (*ptc)[22] = aos; // ptr points at a row of 22 chars

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

high memory

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 vary in length
- `char *aos[3];`



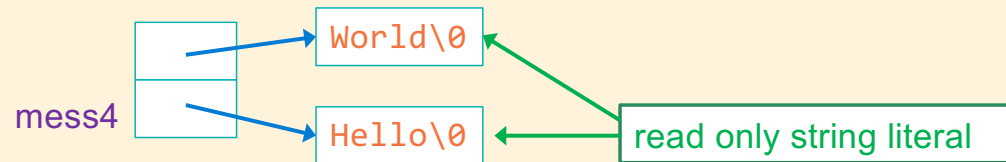
- `aos` is an **array of pointers**; each pointer points at a **character array** (also a string here)
- **Not a 2D array**, but any char can be accessed as if it was in a 2D array of chars
 - When I was learning, this was the most confusing syntax aspects of C

String Literals, Mutable and Immutable arrays - 2

- `mess4` is an array of pointers to immutable arrays

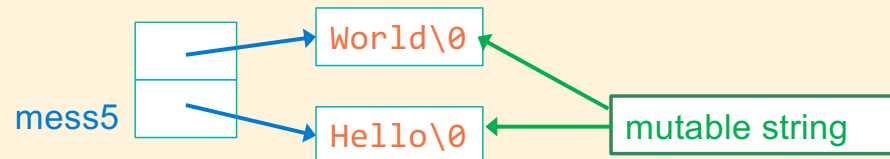
```
char *mess4[] = {"Hello","World"}; // immutable string  
*(*mess4 + 1) = '\0'; // bus error
```

Bus error: writing
read only memory
Seg fault: writing
unallocated memory



- `mess5` is an array of pointers to mutable arrays

```
char *mess5[] = { (char []){"Hello"}, (char []){"World"}};  
*(*mess5 + 1) = '\0'; // OK!
```

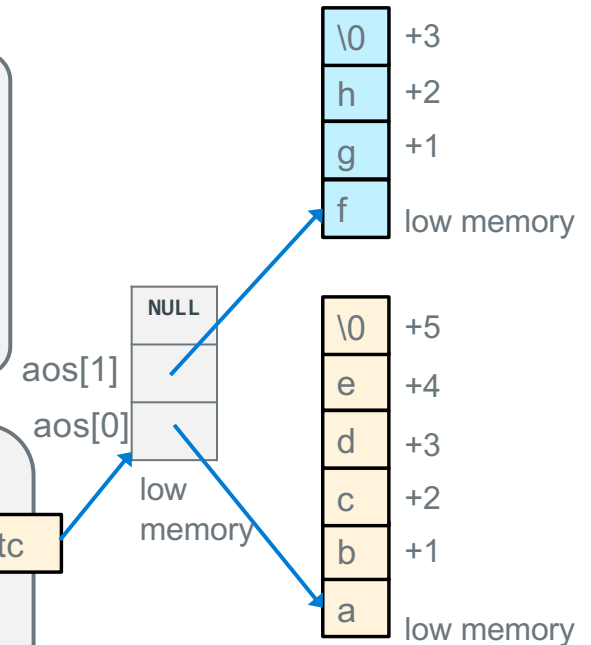


Pointer Array to Mutable Strings

- 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++;  
}
```

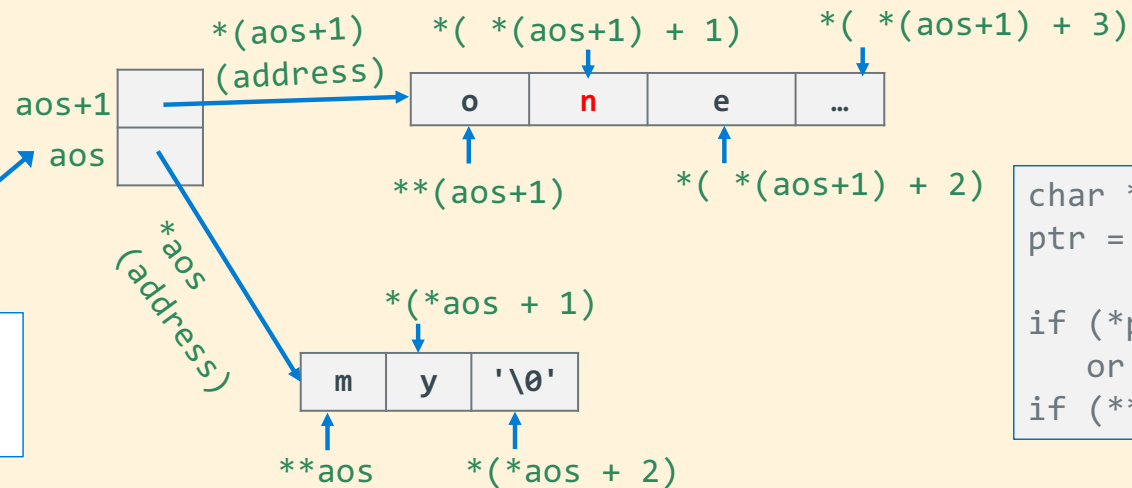
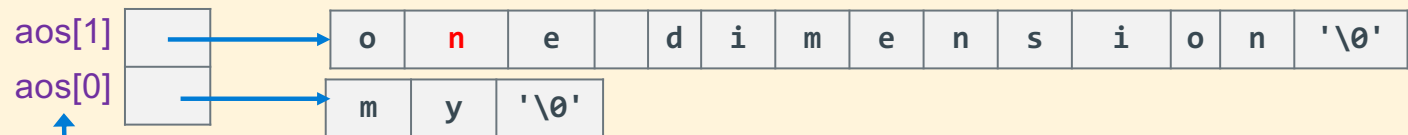


```
%./a.out  
g  
abcde  
abcde  
fgh  
fgh
```


Pointer Array to Strings

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

aos+2 is not shown due to space limits on the slide



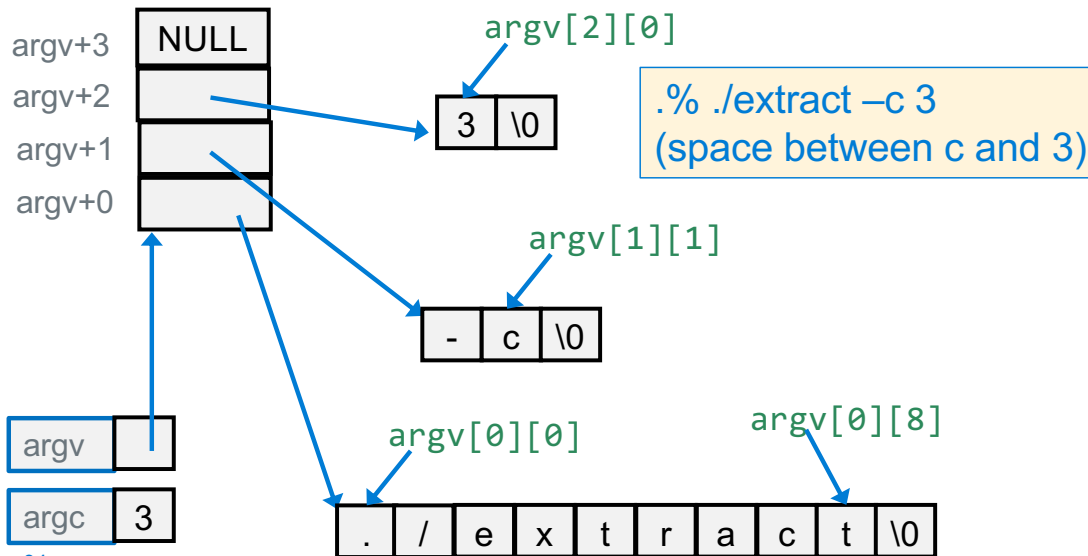
```
char *ptr;
ptr = *aos;

if (*ptr == ',')
    or
if (**aos) == ','
```

Notice that the first elements address is the array name

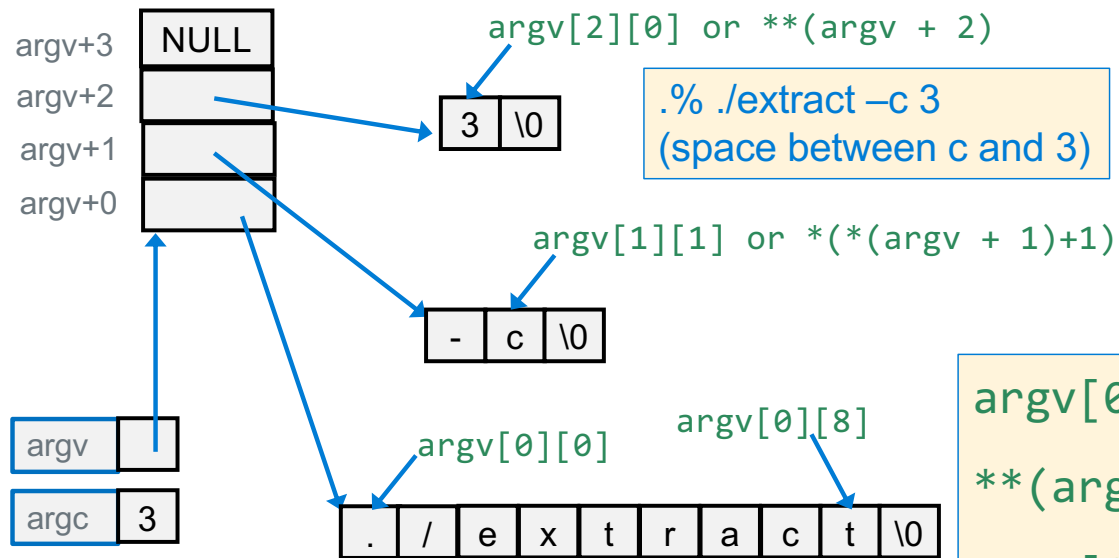
main() Command line arguments: argc, argv

- 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!**



```
printf("%s\n", *(argv+0));
printf("%s\n", *(argv+1));
printf("%s\n", *(argv+2));
```

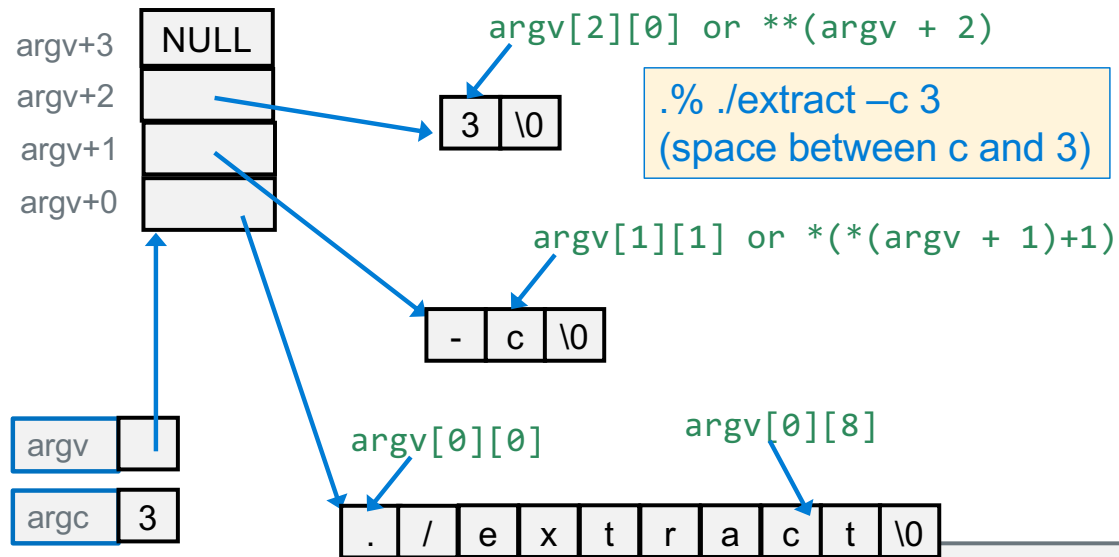
main() Command line arguments: argc, argv



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

```
char *pt = *argv;  
*pt equiv to **argv  
*(pt+8) equiv to * (*argv + 8)
```

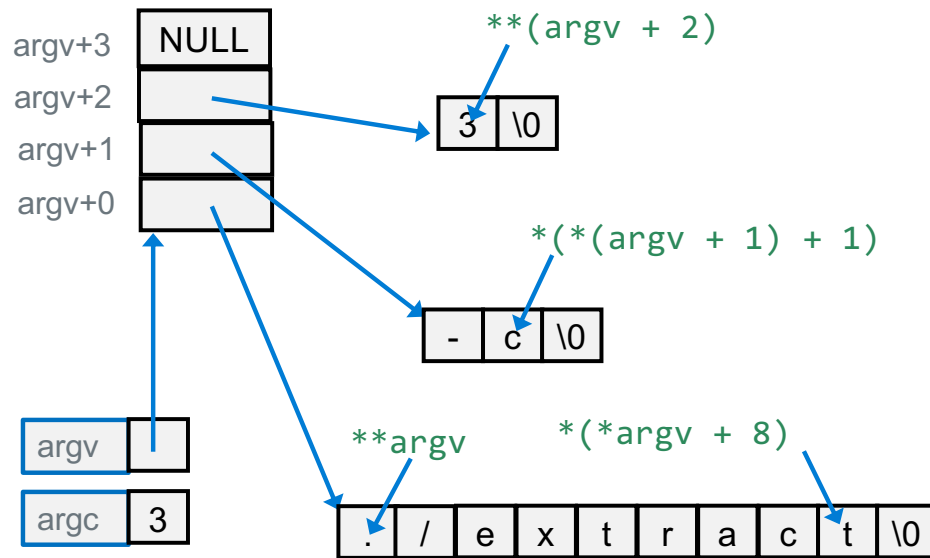
Printing argv char at a time



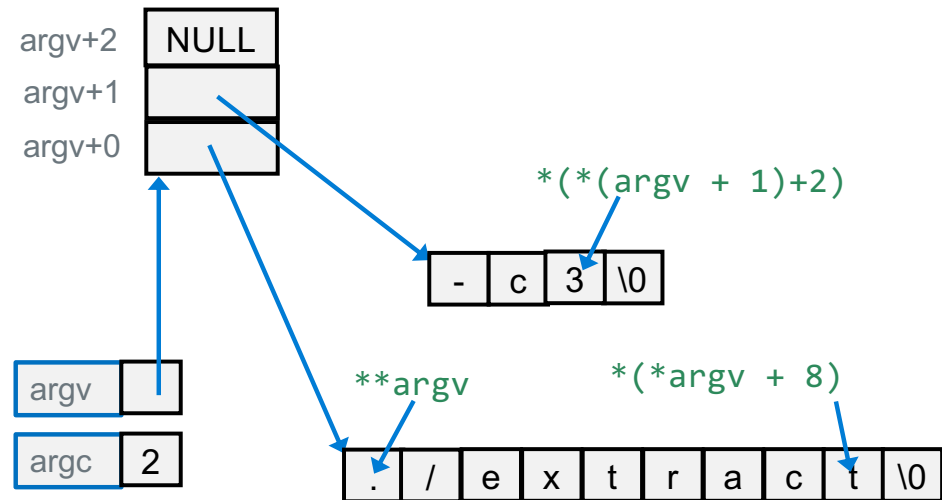
```
int main(int argc, char *argv[])
{
    (void)argc; // shut up the compiler
    for (int i = 0; argv[i] != NULL; i++) {
        for (int j = 0; argv[i][j] != '\0'; j++)
            putchar(argv[i][j]);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

```
int main(int argc, char **argv)
{
    char *pt;
    (void)argc; // shut up the compiler
    while ((pt = *argv++) != NULL) {
        while (*pt != '\0')
            putchar(*pt++);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

main() Command line arguments: argc, argv



`./extract -c 3`
(space between c and 3)



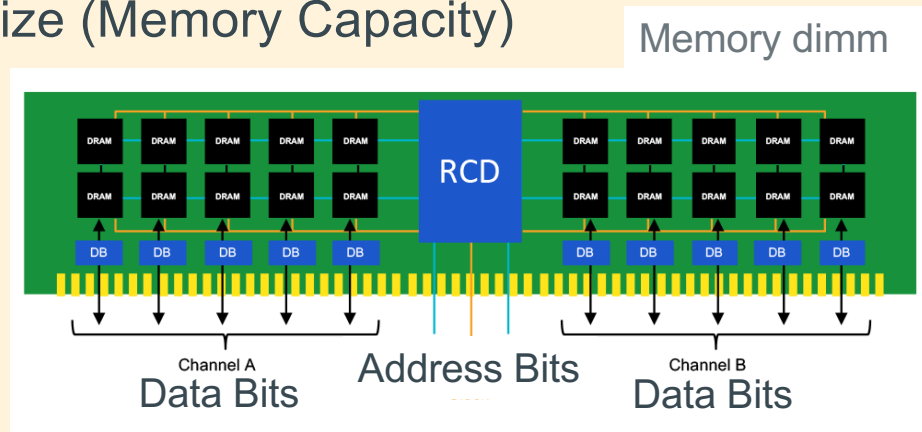
`./extract -c3`
(No space between c and 3)

Extra Slides

-

Memory Size

- Since memory addresses are implemented in hardware using binary
 - The **Size (number of byte sized cells)** of Memory is specified in **powers of 2**
- Memory size/capacity in **bytes** is specified by the “**Number of bits**” in an address
 - 32 bits of address = $2^{32} = 4,294,967,296$
 - Address Range is 0 to $2^{32} - 1$ (unsigned)
- Shorthand notation for address size (Memory Capacity)
 - KB = 2^{10} (K=1024) kilobyte
 - MB = 2^{20} megabyte
 - GB = 2^{30} gigabyte
 - TB = 2^{40} terabyte
 - PB = 2^{50} petabyte



Fixed size types in C (later addition to C)

- Sometimes programs need to be written for a particular range of integers or for a particular size of storage, regardless of what machine the program runs on
- In the file `<stdint.h>` the following fixed size types are defined for use in these situations:

Signed Data types	Unsigned Data types	Exact Size
<code>int8_t</code>	<code>uint8_t</code>	8 bits (1 byte)
<code>int16_t</code>	<code>uint16_t</code>	16 bits (2 bytes)
<code>int32_t</code>	<code>uint32_t</code>	32 bits (4 bytes)
<code>int64_t</code>	<code>uint64_t</code>	64 bits (8 bytes)

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
```

Pointer Practice

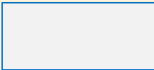
`int *ptr;` Declares a variable, `ptr`, which is a pointer to (it contains the address of) an `int` in memory

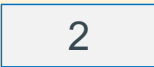
`int x = 5;`
`int y = 2;` Declares two variables, `x` and `y`, that contain `ints`, and *initializes* them to 5 and 2, respectively

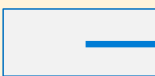
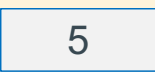

`ptr = &x;` Sets `ptr` to contain the address of `x` ("`ptr` points to `x`")

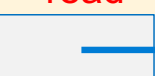
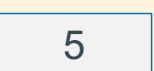

`y = 1 + *ptr;` Sets `y` to "1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y = 1 + x;`
"Dereference `ptr`"

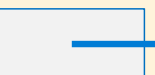


`x = *(&y);` Sets `x = y`; The `*` and `&` cancel each other. get the address of `y` and then get the contents pointed by that address

`ptr` 

`x`  write
`y`  write

`ptr`  `x`  5
`y`  2

`ptr`  `x`  5 read
`y`  6 write

`ptr`  `x`  6 write
`y`  6 read

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 used by the string

- **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 does not 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)

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);
```

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

- 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

<i>index</i>	0	1	2	3	4	5
<i>char</i>	'H'	'e'	'l'	'l'	'o'	'\0'

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

```
char *strcpy(char *s0, char *s1)  
{  
    char *str = s0;  
  
    if ((s0 == NULL) || (s1 == NULL))  
        return NULL;  
    while (*s0++ = *s1++)  
        ;  
    return str; // address of dest string  
}
```