

Version 2.13

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 8

Keith Muller

Vax 11/780 1980





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 + 2 = 3;
*ptr = *ptr + 1;   // (*ptr)++ is array[0] = 2+1 = 3;
ptr = 1 + ptr;     // ptr = &array[1] = ptr points at 5
```

```
x = 1 + ++(*ptr++);
```

```
*ptr = *ptr + 1;   // (*ptr)++ is array[0]=2+1=3
x = 1 + *ptr;      // x = 1 + 3 = 4;
ptr = 1 + ptr;     // ptr = &array[1]; ptr -> 5
```

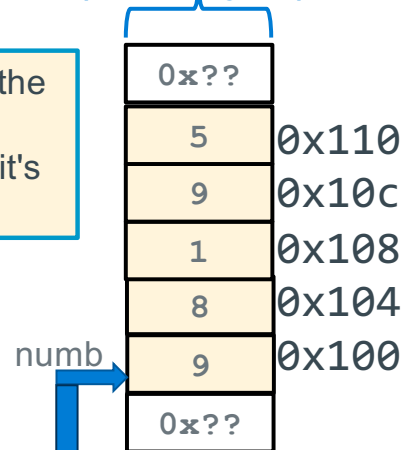
x

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

Observe numb is the name of an array (whose Rvalue is it's starting address)

1 word content
(int = 4 bytes)



Remember a is parameter copy so is a separate variable that contains a pointer to num

```
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];
```

```
    }
```

```
}
```

this 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

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 */
```

Pointer returns from a function call (NULL Examples)

This function returns a pointer to the character that follows the first comma ','

```
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
}
```

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 + 1) = '\0'; // Not OK (bus error)
```

`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

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 (fills rest with zeros '\0')

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

high
memory

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

low
memory

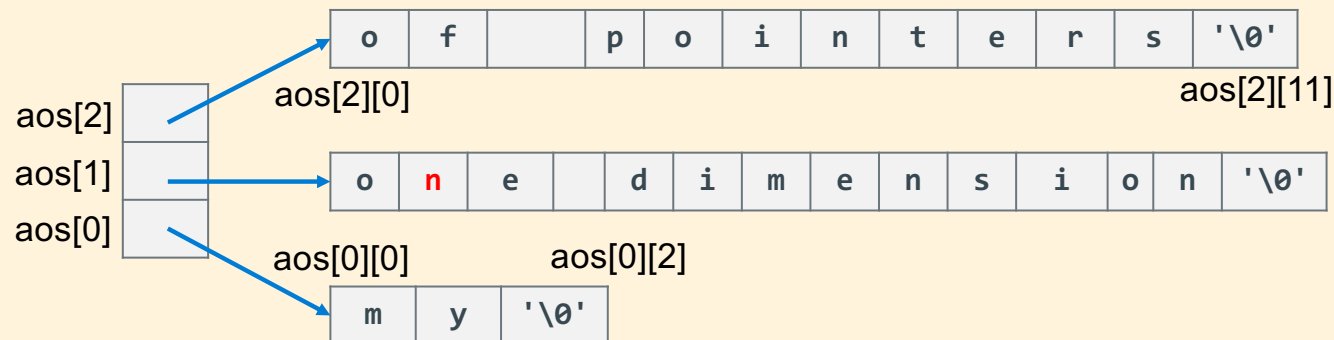
```
#define ROWS 3
char aos[ROWS][22] = { "my", "two dimensional", "char array"};
char (*ptc)[22] = aos; // ptc points at a row of 22 chars

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

high
memory

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

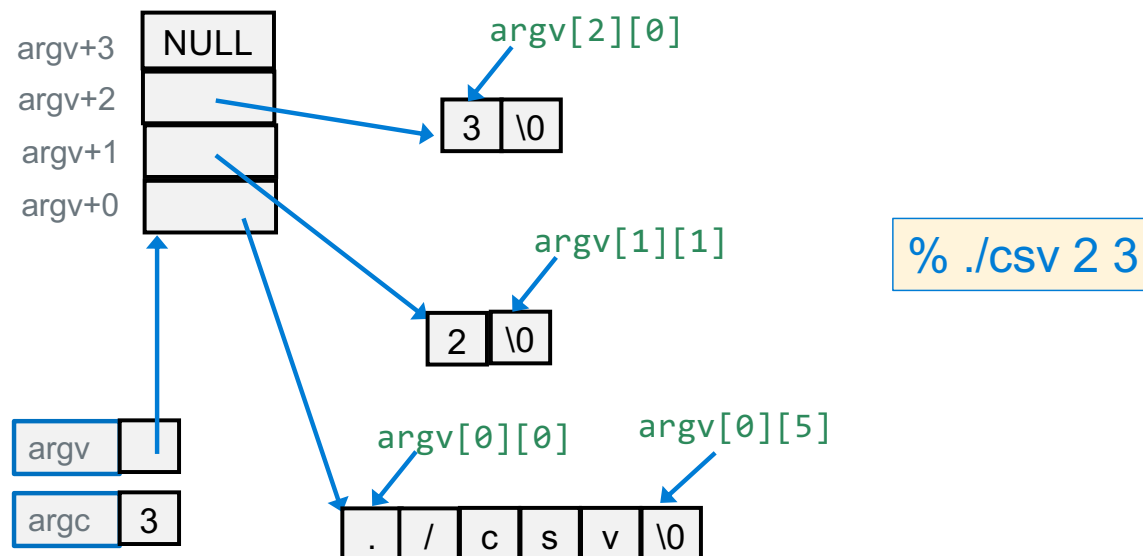
- 2D char arrays are an inefficient way to store strings (wastes memory) unless all the strings are similar lengths, so 2D char arrays are *rarely used* with string elements
- **An array of pointers** is common for strings as "rows" can vary in length
- `char *aos[3];`



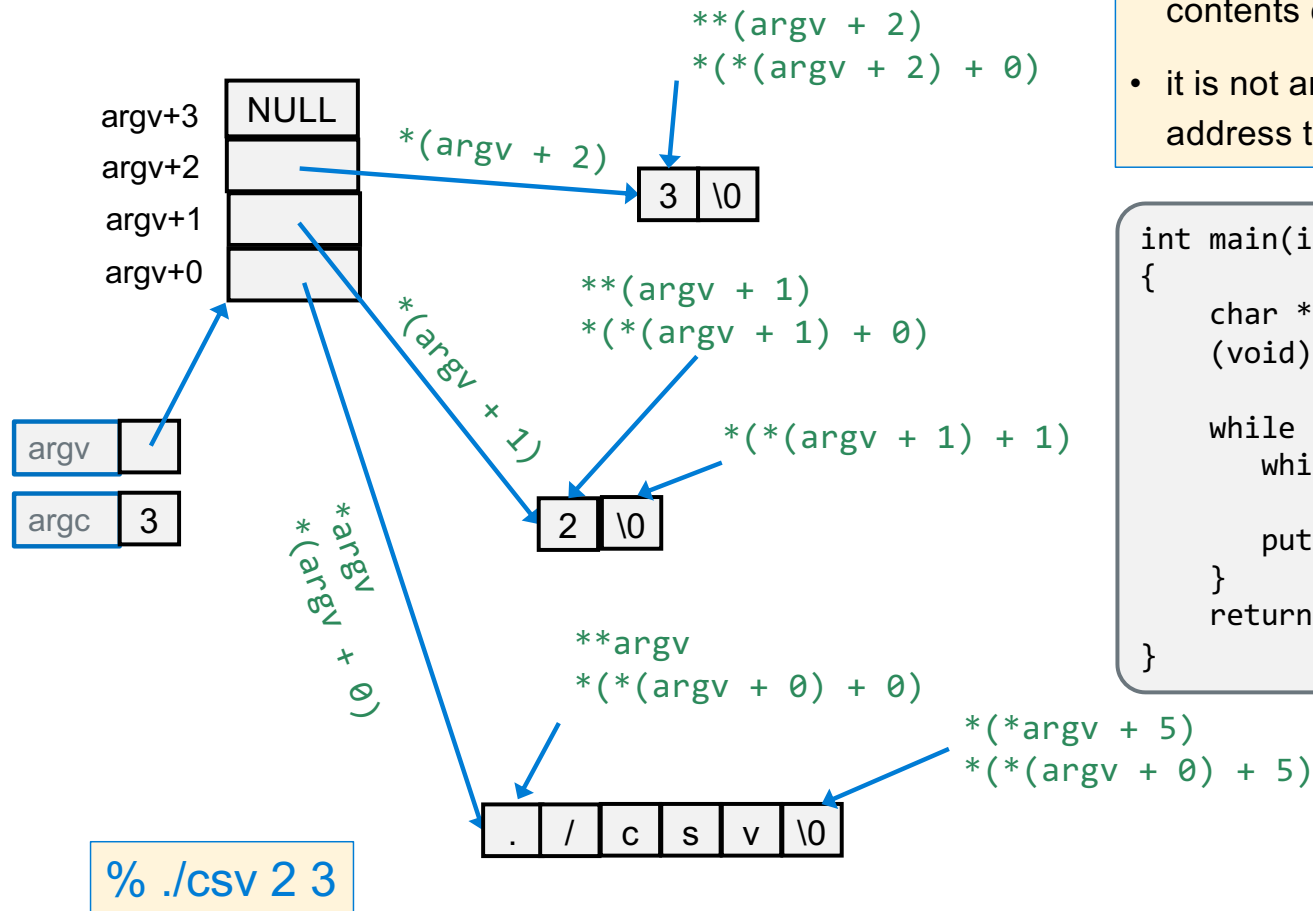
- `aos` is an array of pointers; each pointer points at a character array (also a string here)
- **Not a 2D array**, but any char can be accessed as if it was in a 2D array of chars
 - When I was learning, this was the most confusing syntax aspects of C

main() Command line arguments: argc, argv

- Arguments are passed to main() as a pointer to an array of pointers to char arrays (strings)(**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] or *(argv + argc) always contains a NULL (0) sentinel
- argv elements point at **mutable strings!**



Accessing argv char at a time



- `argv` is a pointer variable, whose contents can be changed
- it is not an array name, which is just an address that cannot be changed

```
int main(int argc, char **argv)
{
    char *pt;
    (void)argc; // shut up the compiler

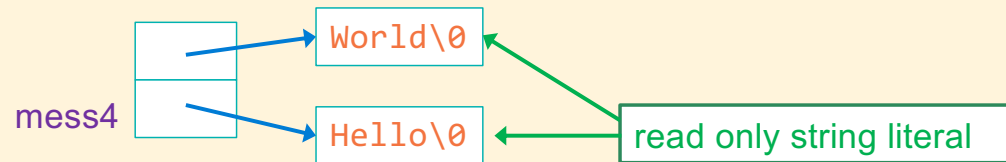
    while ((pt = *argv++) != NULL) {
        while (*pt != '\0')
            putchar(*pt++);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

Defining an Array of Pointer to Strings

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

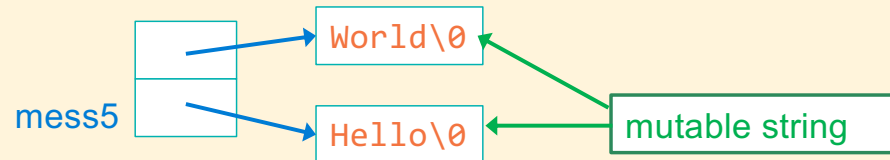
```
char *mess4[] = {"Hello","World"}; // immutable string  
*(*mess4 + 1) = '\0'; // bus error
```

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!
```

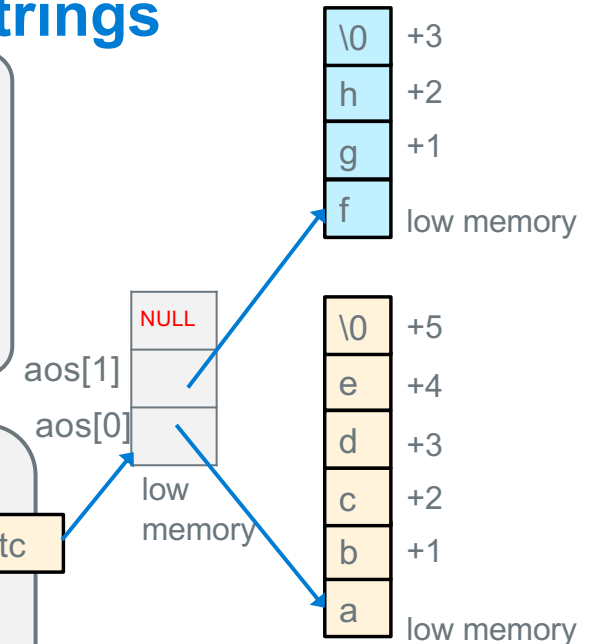


Defining an Array of Pointers to Mutable Strings

- 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  
abcde  
abcde  
fgh  
fgh
```

Pointers to Functions (Function Pointers)

- Similar in concept to an array name, a function name ends up being the address of the first instruction in a function

- A function pointer variable contains the address of a function

- Generic format: `returnType (*name)(type1, ..., typeN)`

- Looks like a function prototype with extra * in front of name
- Why are parentheses around (*name) needed?

`returnType *name(type1, ..., typeN) //wrong`

- Above says name is a function returning a pointer to returnType

- Using the function:

`(*name)(arg1, ..., argN)`

`name(arg1, ..., argN)`

Calls the pointed-to function with the given arguments and returns the return value

Pointers to Function Example

```
int add1(int);
int sqr(int);
void array_update(int (*)(int), int *, int);
void print_array(int *, int);

int main(void)
{
    int array[] = {4, 8, 15, 16, 23, 42};
    int cnt = sizeof(array)/sizeof(array[0]);

    print_array(array, cnt);
    array_update(add1, array, cnt);
    print_array(array, cnt);
    array_update(sqr, array, cnt);
    print_array(array, cnt);
    return EXIT_SUCCESS;
}
```

```
void array_update(int (*f)(int), int *a, int cnt)
{
    while (a < endpt) {
        *a = f(*a);
        a++;
    }
}
```

```
void print_array(int *a, int cnt)
{
    int *endpt = a + cnt;

    while (a < endpt)
        printf("%d ", *a++);
    printf("\n");
}
```

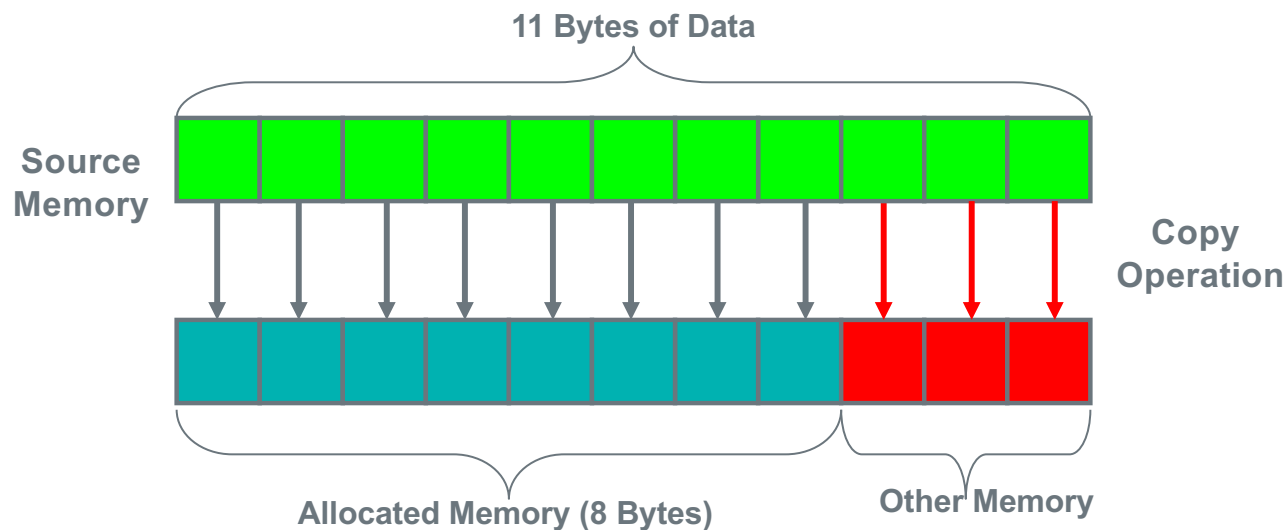
```
int add1(int i)
{
    return i + 1;
}

int sqr(int i)
{
    return i * i;
}
```

```
./a.out
4 8 15 16 23 42
5 9 16 17 24 43
25 81 256 289 576 1849
```

string buffer overflow: common security flaw

- A **buffer overflow** occurs when data is written **outside the boundaries** of the **memory allocated to target variable** (or target buffer)
- **strcpy()** is a very *common source of buffer overrun security flaws*:
 - always ensure that the **destination array is large enough** (and don't forget the null terminator)
- **strcpy()** can cause **problems** when the **destination** and **source regions overlap**



strcpy() buffer overflow: over-write of an adjacent variable

```
int main(void)
{
    char s1[] = "before";
    char r2[] = "xyz";
    char s2[] = "after";

    printf("s2: %s\nr2: %s\nr2: %s\n", s2, r2, s1);

    strcpy(r2, "hello");

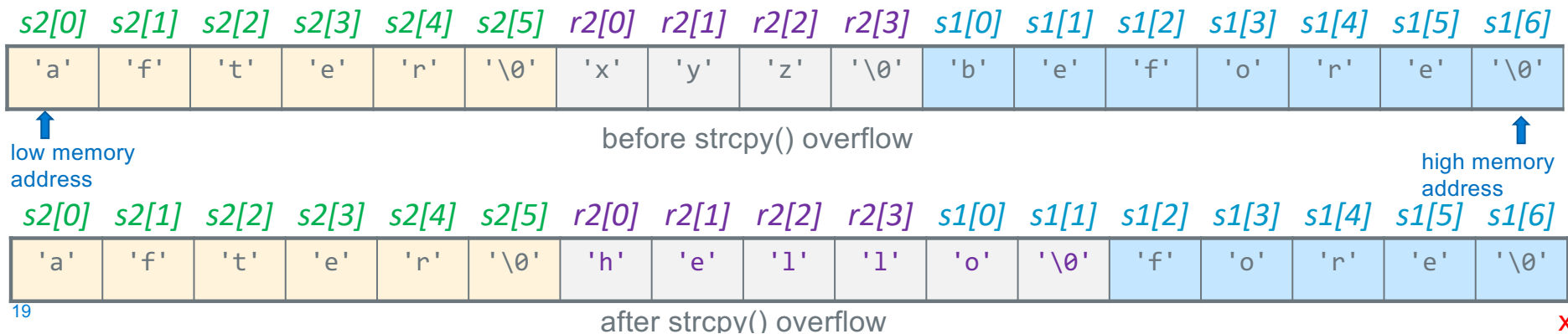
    printf("\ns2: %s\nr2: %s\nr2: %s\n", s2, r2, s1);
    return EXIT_SUCCESS;
}
```

these are mutable
arrays, not literals

compile on pi-cluster with
gcc test.c

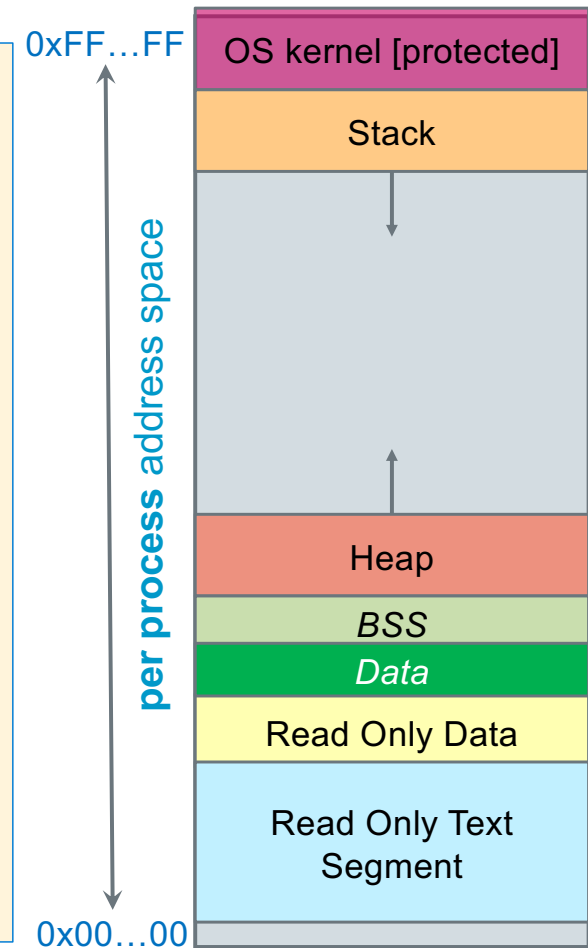
```
./a.out
s2: after
r2: xyz
s1: before

s2: after
r2: hello
s1: o
```



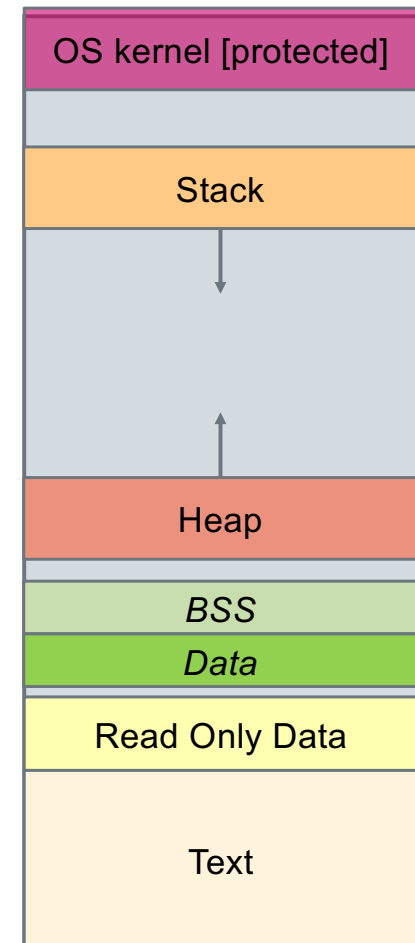
Process Memory Under Linux

- When your **program is running** it has been loaded into memory and is **called a process**
- **Stack segment:** Stores **Local** variables
 - Allocated and freed at function call entry & exit
- **Data segment + BSS:** Stores **Global** and **static** variables
 - **Allocated/freed** when the process **starts/exits**
 - **BSS** - Static variables with an implicit initial value
 - **Static Data** - Initialized with an explicit initial value
- **Heap segment:** Stores **dynamically-allocated** variables
 - Allocated with a function call
 - Managed by the stdio library malloc() routines
- **Read Only Data:** Stores **immutable** Literals
- **Text:** Stores your code in machine language + libraries



The Heap Memory Segment

- **Heap**: "pool" of memory that is available to a program
 - Managed by C runtime library and linked to your code; **not managed by the OS**
- Heap memory is **dynamically** *"borrowed"* or *"allocated"* by calling a library function
- When heap memory is no longer needed, it is *"returned"* or *deallocated* for **reuse**
- Heap memory has a lifetime from allocation until it is deallocated
 - Lifetime is independent of the scope it is allocated in (it is like a static variable)
- If too much memory has already been allocated, the library will attempt to borrow additional memory from the OS and will fail, returning a NULL



Heap Dynamic Memory Allocation Library Functions

<code>#include <stdlib.h></code>	args	Clears memory
<code>void *malloc(...)</code>	<code>size_t size</code>	no
<code>void *calloc(...)</code>	<code>size_t nmemb, size_t memsize</code>	yes
<code>void free(...)</code>	<code>void *ptr</code>	no

- **void *** means these library functions return a pointer to **generic (untyped) memory**
 - Be careful with void * pointers and pointer math as void * points at untyped memory
 - The assignment to a typed pointer "*converts*" it from a void *
- **size_t** is an **unsigned integer data type**, the result of a **sizeof()** operator

```
int *ptr = malloc(sizeof(*ptr) * 100); // allocate an array of 100 ints
```

- please read: % man 3 malloc

Use of Malloc

```
void *malloc(size_t size)
```

- Returns a pointer to a **contiguous** block of **size** bytes of **uninitialized memory** from the heap
 - The block is **aligned to an 8-byte (arm32) or 16-byte (64-bit arm/intel) boundary**
 - **returns NULL** if allocation failed (also sets **errno**) **always CHECK for NULL RETURN!**
- Blocks **returned on different calls to malloc()** are **not necessarily adjacent**
- **void *** is implicitly cast into **any pointer type on assignment to a pointer variable**

```
char *bufptr;
```

```
/* ALWAYS CHECK THE RETURN VALUE FROM MALLOC!!!! */
```

```
if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {  
    fprintf(stderr, "Unable to malloc memory");  
    return NULL;  
}
```

Using and Freeing Heap Memory

- void **free**(void *p)
 - Deallocates the **whole block pointed to by p** to the pool of available memory
 - **Freed memory is used in future allocations** (**expect the contents to change after freed**)
 - Pointer **p** must be the same address as **originally returned** by one of the heap allocation routines `malloc()`, `calloc()`, `realloc()`
 - Pointer argument to `free()` is not changed by the call to `free()`
- **Defensive programming**: **set the pointer to NULL** after passing it to `free()`

```
char *bufptr;  
  
if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {  
    fprintf(stderr, "Unable to malloc memory");  
    return NULL;  
}  
free(bufptr);           // returns memory to the heap  
bufptr = NULL;          // set bufptr to NULL
```


Heap Memory "Leaks"

- A **memory leak** is when you **allocate memory** on the heap, **but never free it**

```
void  
leaky_memory (void)  
{  
    char *bytes = malloc(BLKSZ * sizeof(*bytes));  
    ...  
    /* code that never deallocates the memory */  
    return; // you lose the address in bytes when leaving scope  
}
```

- **Best practice:** free up memory **you allocated** when you no longer need it
 - If you keep allocating memory, you may run out of memory in the heap!
- **Memory leaks** may cause **long running programs to fault** when they **exhaust OS memory limits**
- **Valgrind** is a tool for finding memory leaks (not pre-installed in all linux distributions though!)

Valgrind – Finding Buffer Overflows and Memory leaks

```
1 #define SZ 50
2 #include <stdlib.h>
3 int main(void)
4 {
5     char *buf;
6     if ((buf = malloc(SZ * sizeof(*buf))) == NULL)
7         return EXIT_FAILURE;
8     *(buf + SZ) = 'A';
9     // free(buf);
10    return EXIT_SUCCESS;
11 }
```

```
% valgrind -q --leak-check=full --leak-resolution=med -s ./valgexample
==651== Invalid write of size 1
==651==    at 0x10444: main (valg.c:8)
==651== Address 0x49d305a is 0 bytes after a block of size 50 alloc'd
==651==    at 0x484A760: malloc (vg_replace_malloc.c:381)
==651==    by 0x1041B: main (valg.c:6)
==651==
==651== 50 bytes in 1 blocks are definitely lost in loss record 1 of 1
==651==    at 0x484A760: malloc (vg_replace_malloc.c:381)
==651==    by 0x1041B: main (valg.c:6)
==651==
==651== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Writing outside of allocated
buffer space

Memory not freed

More Dangling Pointers: Reusing "freed" memory

- When a pointer points to a memory location that is no longer “valid”
- Really hard to debug as the use of the return pointers may not generate a seg fault

```
char *dangling_freed_heap(void)
{
    char *buff = malloc(BLKSZ * sizeof(*buff));
    ...
    free(buff); // memory pointed at buf may be reused
    return buff;
}
```

- `dangling_freed_heap()` type code often causes the allocators (`malloc()` and friends) to **seg fault**
 - Because it corrupts data structures the heap code uses to manage the memory pool

strdup(): Allocate Space and Copy a String

```
char *strdup(char *s);
```

- **strdup** is a function that has a **side effect** of returning a **null-terminated**, heap-allocated string copy of the provided text
- Alternative: **malloc** and copy the string
- The caller is responsible for freeing this memory when no longer needed

```
char *str = strdup("Hello, world!");  
*str = 'h';
```

```
free(str); // caller correctly frees up space allocated by strdup()  
str = NULL;
```

Calloc()

```
void *calloc(size_t elementCnt, size_t elementSize)
```

calloc() variant of malloc() but zeros out every byte of memory before returning a pointer to it (so this has a runtime cost!)

- First parameter is the number of elements you would like to allocate space for
- Second parameter is the size of each element

```
// allocate 10-element array of pointers to char, zero filled  
char **arr;  
arr = calloc(10, sizeof(*arr));  
if (arr == NULL)  
    // handle the error
```

- Originally designed to allocate arrays but works for any memory allocation
 - calloc() multiplies the two parameters together for the total size
- calloc() is more expensive at runtime (uses both cpu and memory bandwidth) than malloc() because it must zero out memory it allocates at runtime
- Use calloc() only when you need the buffer to be zero filled prior to FIRST use

Introduction to Structs – An Aggregate Data Type

- **Structs** are a **collection (or aggregation) of values** grouped **under a single name**
 - Each **variable in a struct** is called a **member** (sometimes **field** is used)
 - Each **member** is identified with a **name**
 - Each **member** can be (and quite often are) **different types, include other structs**
 - Like a Java class, but no associated methods or constructors with a struct
- Structure definition **does not** define a variable instance, nor does it allocate memory:
 - It creates a **new variable type** uniquely identified by its **tagname**:
"struct tagname" includes the **keyword struct** and the **tagname** for this type

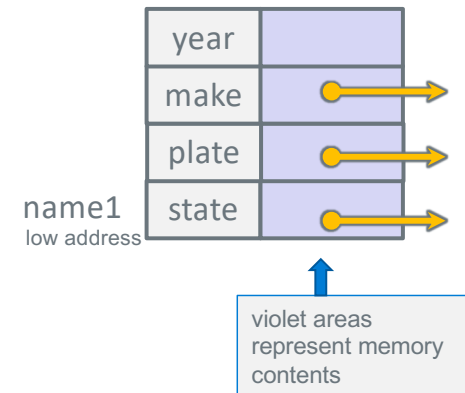
Easy to forget
semicolon!

```
struct tagname {  
    type1 member1;  
    ...  
    typeN memberN;  
};
```

```
struct vehicle {  
    char *state;  
    char *plate;  
    char *make;  
    int year;  
};
```

Struct Variable Definitions

```
struct vehicle {  
    char *state;  
    char *plate;  
    char *make;  
    int year;  
};  
struct vehicle name1;
```



- Variable definitions like any other data type:

```
struct vehicle name1, *pn, ar[3];
```

type: "struct vehicle"

single variable instance

pointer

array

Accessing members of a struct

- Like arrays, struct variables are aggregated contiguous objects in memory
- The `.` structure operator which *"selects"* the requested field or member

```
struct date {           // defining struct type
    int month;          // member month
    int day;            // member date
};
```

day	
month	

struct date type definition

```
struct date bday; // define a struct instance
bday.month = 1;
bday.day = 24;

// alternative initializer syntax
struct date new_years_eve = {12, 31};
struct date final = {.day= 24, .month= 1};
```

day	24
month	1

bday definition

Accessing members of a struct with pointers

```
struct date {           // defining struct type
    int month;          // member month
    int day;            // member date
};
```

day	
month	

- Define a *pointer* to a struct

```
struct date *ptr = &bday;
```

- Two ways to reference a member via a struct pointer (. is higher precedence than *):

1. Use * and . operators: (*ptr).month = 11;

2. Use -> operator for shorthand: ptr->month = 11;

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

Accessing members of a struct

```
struct date {           // defining struct type
    int month;           // member month
    int day;             // member date
};
```

- You can create an array of structs and initialize them

```
struct date quarter[] =
    { {1,2}, {3,4}, {5,6}, {7,8}, {9,10} };
int cnt = sizeof(quarter)/sizeof(*quarter); // = 5
```

quarter[4]	day	10
	month	9
quarter[3]	day	8
	month	7
quarter[2]	day	6
	month	5
quarter[1]	day	4
	month	3
quarter[0]	day	2
	month	1

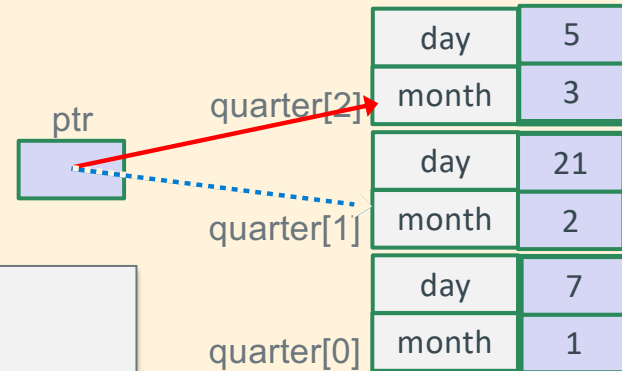
Accessing members of a struct

```
struct date quarter[3];  
struct date *ptr;
```

```
ptr = quarter + 1;    // array name = address  
ptr->month = 2;  
ptr->day = 21;        // or (*ptr).day = 21;
```

```
(ptr-1)->month = 1;   // or (*(ptr-1)).month = 4;  
(ptr-1)->day = 7;
```

```
(++ptr)->month = 3;  
ptr->day = 5;
```



Typedef usage with Struct – Another Style Conflict

- *Typedef* is a way to create an *alias* for another data type (not limited to just structs)
`typedef <data type> <alias>;`
 - After typedef, the alias can be used interchangeably with the original data type
 - e.g., `typedef unsigned long int size_t;`
- *Many claim typedefs* are easier to understand than tagged struct variables
 - `typedef with structs` are not allowed in the cse30 style guidelines (Linux kernel standards)

```
struct nm {  
    /* fields */  
};  
typedef struct nm item;  
  
item n1;  
struct nm n2;  
item *ptr;  
struct nm *ptr2;
```

```
typedef struct name2_s {  
    int a;  
    int b;  
} name2_s;  
  
name2_s var2;  
name2_s *ptr2;
```

```
typedef struct {  
    int a;  
    int b;  
} pair;  
  
pair var3;  
pair *ptr3;
```

Copying Structs

- You can assign the member value(s) of the whole struct from a struct of the same type
– Performance Caution: *this copies the entire contents during execution*

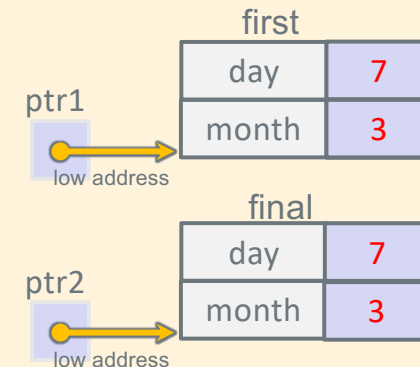
- Individual fields can also be copied

```
struct date first = {1, 1};  
struct date final = {.day= 31, .month= 12};
```

```
struct date *pt1 = &first;  
struct date *pt2 = &final;
```

```
final.day = first.day; // both day are 1  
final = first; // copies whole struct
```

```
pt2->month = 3;  
*pt1 = *pt2; // copies whole struct  
pt2->day = 7;  
pt1->day = pt2->day; // both days are now 7
```



Caution: Shallow Copies of Pointer Members

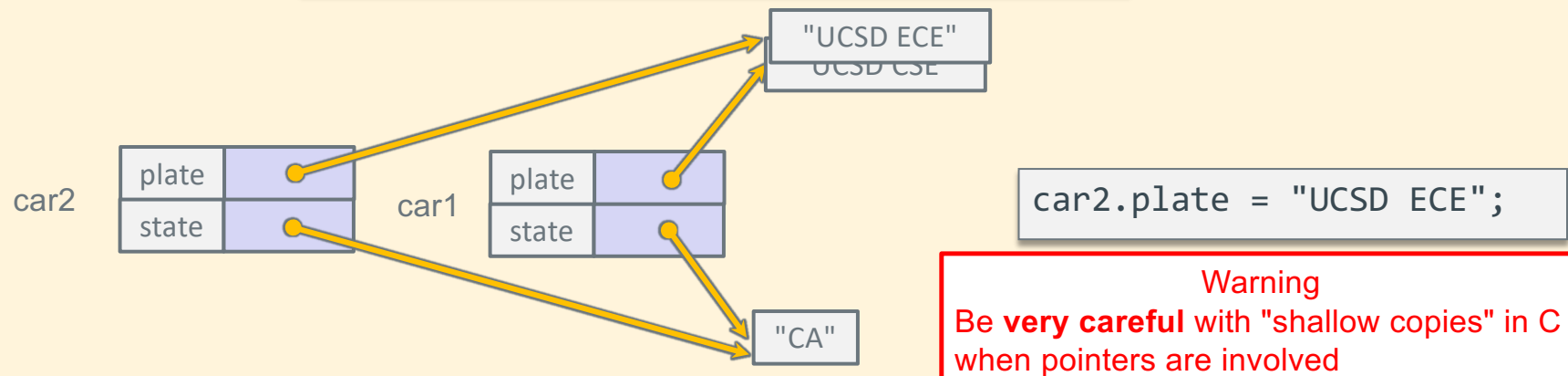
```
struct vehicle {  
    char *state;  
    char *plate;  
};
```

```
struct vehicle car1 = {"CA", "UCSD CSE"};  
struct vehicle car2;
```



- When you assign one struct to another, it copies member contents including pointer values!

```
car2 = car1; // copies members exactly
```

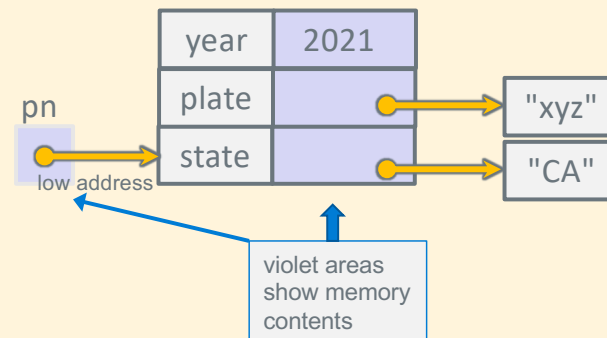


Deep Copies of Structs

- **Must** Allocate space that is pointed at by member pointers independently (they are not part of the struct, only the pointers are)

```
struct vehicle {  
    char *state;  
    char *plate;  
    int year;  
};  
struct vehicle car1;  
  
pn = &car1;
```

```
car1.state = strdup("CA");  
pn->plate = strdup("xyz");  
pn->year = 2021;
```



Struct: Copy and Member Pointers --- "Deep Copy"

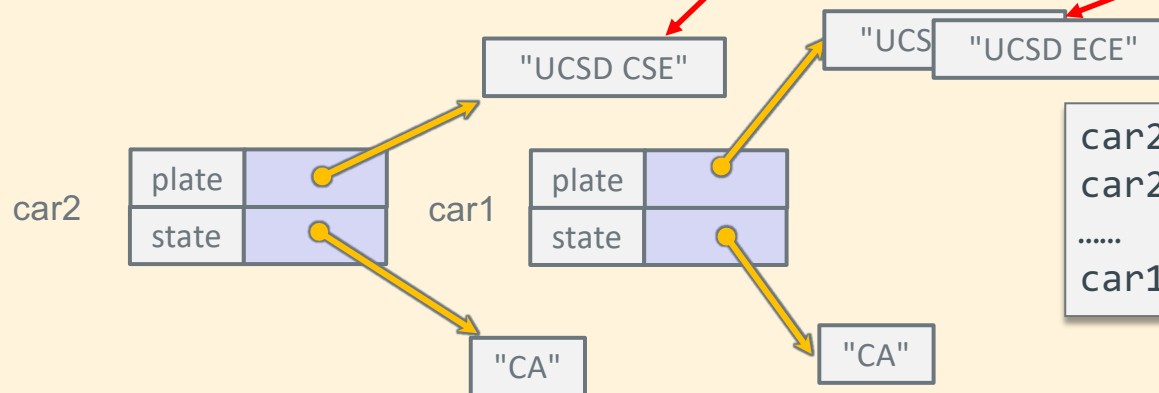
```
struct vehicle {  
    char *state;  
    char *plate;  
};
```

```
struct vehicle car1 = {"CA", "UCSD CSE"};  
struct vehicle car2;
```

mutable strings (heap memory)

immutable strings (read-only data)

- Use `strdup()` to copy the strings



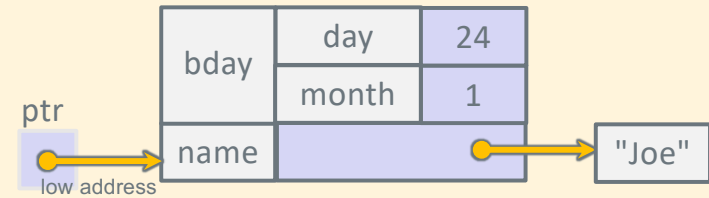
```
car2.plate = strdup(car1.plate);  
car2.state = strdup(car1.state);  
.....  
car1.plate = "UCSD ECE";
```


Nested Structs

- Structs like any other variable can be a member of a struct, this is called a **nested struct**

```
struct date {  
    int month;  
    int day;  
};
```

```
struct person {  
    char *name;  
    struct date bday;  
};
```

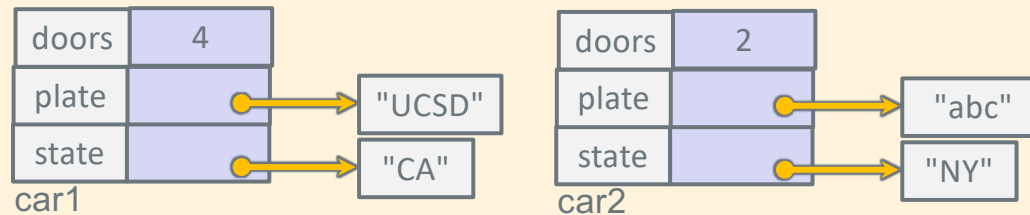


```
struct person first;  
struct person *ptr;  
ptr = &first;  
  
first.name = "Sam"; // immutable string  
first.name = (char []) {"Joe"}; // mutable string, lost address to Sam  
  
first.bday.month = 1;  
first.bday.day = 24;  
  
// below is the same as above  
ptr->bday.month = 1;  
ptr->bday.day = 24;
```

Comparing Two Structs

- You cannot compare entire structs, you must compare them one member at a time

```
struct vehicle {  
    char *state;  
    char *plate;  
    int doors;  
};
```



```
struct vehicle car1 = {"CA", "UCSD", 4};  
struct vehicle car2 = { (char []) {"NY"}, (char []) {"abc"}, 2};
```

```
if ((strcmp(car1.state, car2.state) == 0) &&  
    (strcmp(car1.plate, car2.plate) == 0) &&  
    (car1.doors == car2.doors)) {  
    printf("Same\n");  
} else {  
    printf("Different\n");  
}
```

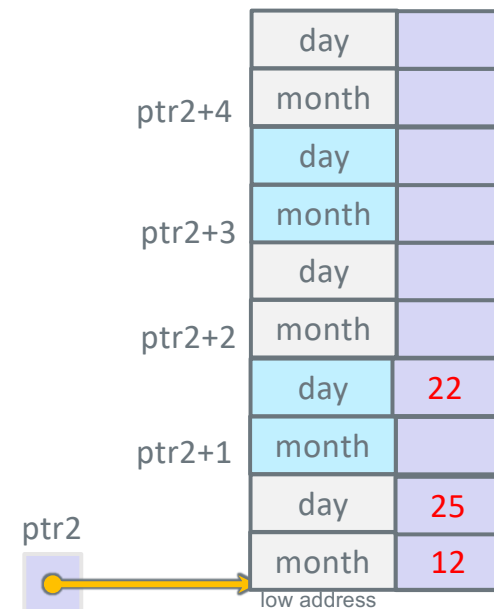
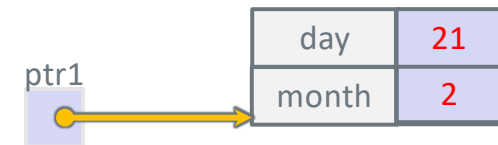
Struct Arrays: Dynamic Allocation

```
#define HOLIDAY 5
struct date *pt1 = malloc(sizeof(*pt1));
struct date *pt2 = malloc(sizeof(*pt2) * HOLIDAY);
```

```
(*ptr1).month = 2;
(*ptr1).day = 21;

pt2->month = 12;
pt2->day = 25;
(pt2+1)->day = 22; //or (*(pt2+1)).month

free(pt1);
pt1 = NULL;
free(pt2);
pt2 = NULL;
```



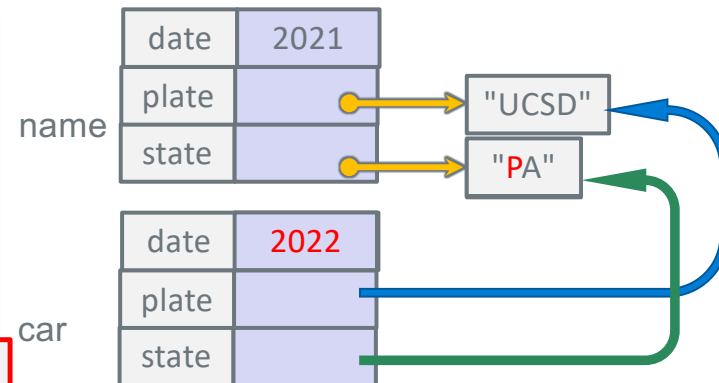
Formal Parameter structs: contents set with shallow copies!

- **WARNING:** When you pass a struct, **you pass a copy** of all the struct members
 - This is a shallow copy (shallow copy – so if you have members that are pointers watch out)
- More often code will pass the pointer to a struct to **avoid the copy costs**
 - Be careful and not modify what the pointer points to (unless it is an output parameter)
- Tradeoffs:
 - Passing a pointer is cheaper and takes less space unless struct is small
 - **Member access cost:** indirect accesses through pointers to a struct member **may** be a bit more expensive and **might be harder for compiler to optimize**
 - For small structs like a struct date **passing a copy is fine**
 - **For** large structs always use pointers (arrays of struct, pass a pointer)
- **For me, I always pass pointers to structs as parameters regardless of size**

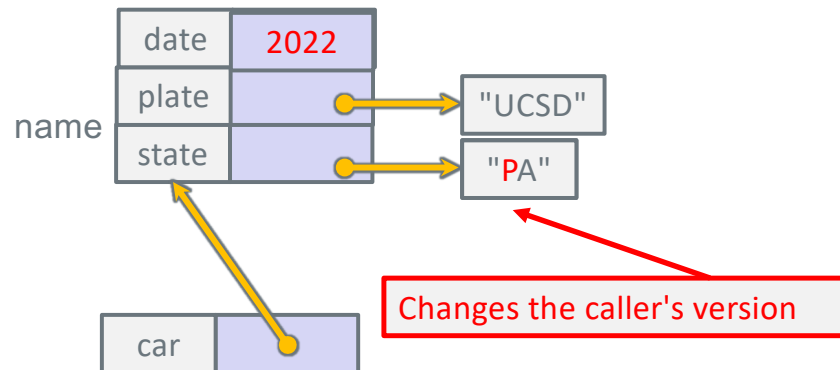
Struct Function Parameters – Be Careful it is not like arrays

```
void change1(struct vehicle car)
{
    car.date = 2022; // oops!
    *(car.state) = "P";
}
...
change1(name);
```

Changes the parameter copy



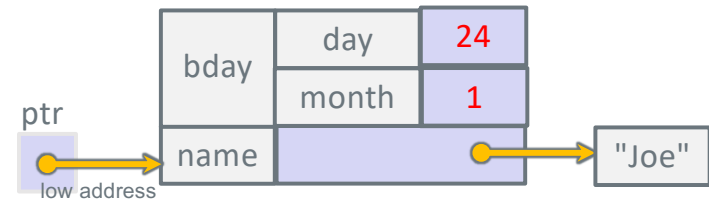
```
void change2(struct vehicle *car)
{
    car->date = 2022;
    *(car->state) = "P";
}
...
change2(name);
```



Struct as an Output Parameter: Deep Copy Example

```
struct date {  
    int month;  
    int day;  
};
```

```
struct person {  
    char *name;  
    struct date bday;  
};
```



```
int fill(struct person *ptr, char *name, int month, int day)
{
    ptr->bday.month = month;
    ptr->bday.day = day;
    if ((ptr->name = strdup(name)) == NULL)
        return -1;
    return 0;
}

...-----calling function -----
    struct person first;
    if (fill(&first, "Joe", 1, 24) == 0)
        printf("%s %d %d\n", first.name, first.bday.month, first.bday.day);
    ...
```