Version 1.06

UCSD CSE 30

Computer Organization and Systems Programming
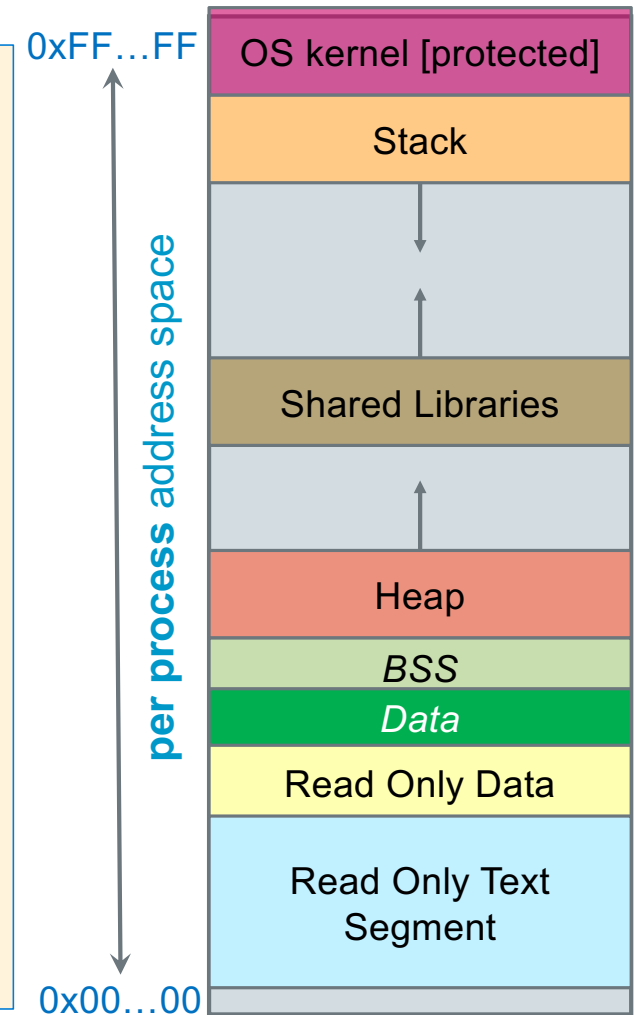
C Programming Part 4

Lecture 10 – Oct 24, 2022

Keith Muller

Xerox Sigma 7 1970

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

0xFF...FF

per process address space

OS kernel [protected]
Stack
Shared Libraries
Heap
BSS
Data
Read Only Data
Read Only Text Segment

0x00...00

X

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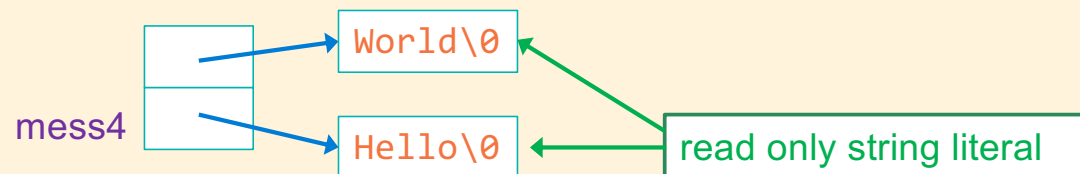
using the cast (char [])
makes it mutable

mess3 [ ] → `Hello World\0` ← mutable string

x

String Literals, Mutable and Immutable arrays - 2

- mess4 is an array of pointers to immutable arrays
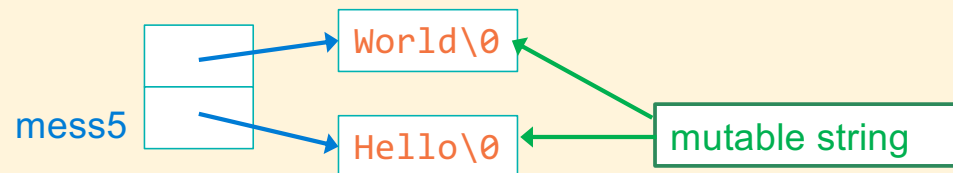
```
char *mess4[] = {"Hello","World"};  // immutable string
*(*mess4 + 1)  = '\0';              // bus error
```

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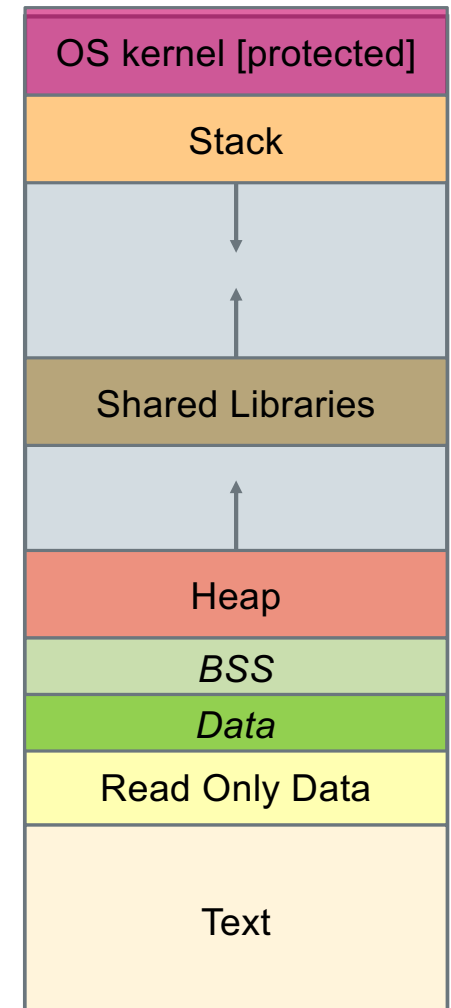Seg fault: writing unallocated memory

World\0

mess4

Hello\0

read only string literal

- mess5 is an array of pointers to mutable arrays

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

World\0

mess5

Hello\0

mutable string

4

x

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

| OS kernel [protected] |
| Stack |
| |
| Shared Libraries |
| |
| Heap |
| *BSS* |
| *Data* |
| Read Only Data |
| Text |

5

X

Heap Dynamic Memory Allocation Library Functions

`#include <stdlib.h>`	args	Clears memory
`void *malloc(…)`	`size_t size`	no
`void *calloc(…)`	`size_t nmemb, size_t memsize`	yes
`void *realloc(…)`	`void *ptr, size_size`	no
`void free(...)`	`void *ptr`	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 (not allowed in C, but allowed in gcc). 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**

X

Use of Malloc

`void *malloc(size_t size)`

- Returns a pointer to a **contiguous** block of `size` bytes **of <u>uninitialized</u> 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

```c
#include <stdlib.h>                 // need this for malloc() etc
char *getbuf(int cnt)
{
    char *bufptr;
    /* ALWAYS CHECK THE RETURN VALUE FROM MALLOC!!!! */
    if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {
        fprintf(stderr, "Unable to malloc memory");
        return NULL;
    }
    return bufptr;    // the calling function must free memory
}
```

X

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 allocation (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()

```
#define COLCNT 1024
char *ptr, *endptr, *bufptr;

bufptr = getbuf(COLCNT);      // do not lose bufptr!, NULL check not shown
ptr = bufptr;
endptr = ptr + COLCNT;
while (ptr < endptr)
        *ptr++ = 'a';          // fill each array element with 'a'
free(bufptr);                  // returns memory to the heap
bufptr = NULL;                 // set bufptr to NULL
```

x

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;
}
```

- 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!)

X

Valgrind – Finding Buffer Overflows and Memory leaks

```c
 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     return EXIT_SUCCESS;
10 }
```

```
% 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

10

X

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);
    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

X

strdup(): Allocate Space and Copy a String

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

- **strdup** is a function that returns a **null-terminated**, heap-allocated string copy of the provided text
- Alternative: **malloc** and copy the string

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

free(str);
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

13

X

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:
  - It creates a new variable type uniquely identified by its tagname:
    "struct tagname" includes the keyword struct and the tagname for this type
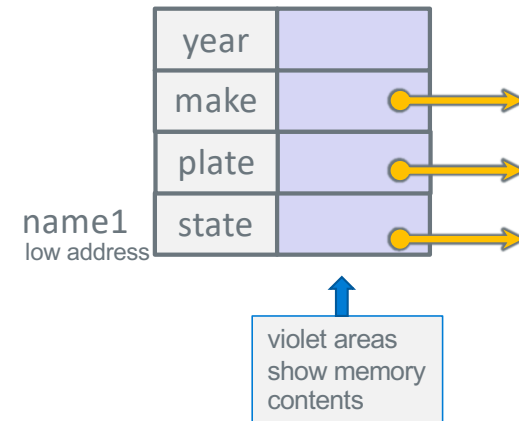
Easy to forget semicolon!

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

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

X

Struct Variable Definitions

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


year
make
plate
name1   state
low address

violet areas show memory contents

- Variable definitions like any other data type:

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

type: "struct vehicle"

single variable instance

pointer

array

X

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;
    int day; // members date struct
};
```

day	
month	

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

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

struct date bday

day	24
month	1

X

Accessing members of a struct with pointers

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

day	
month	

- Now create a *pointer* to a struct

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

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

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

- 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

X

Accessing members of a struct

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

- 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
```

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

X

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;
```

	day	5
quarter[2]	month	3
ptr	day	21
quarter[1]	month	2
	day	7
quarter[0]	month	1

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

X

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;
```

X

Copying Structs

- You can assign the member value(s) of the whole struct from a struct of the same type
  - *this copies the entire contents!*

- 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
```

first

day	7
month	3

ptr1

low address

final

day	7
month	3

ptr2

low address

X
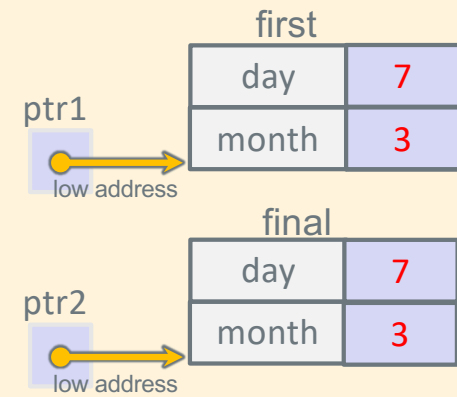
Struct: Copy and Member Pointers

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

```
struct vehicle name  = {"CA", "UCSD CSE"};
struct vehicle name2;
```

| plate | ● | → | "UCSD CSE" |
| state | ● | → | "CA" |

name

immutable strings
(read only data)

- When you assign one struct to another it just copies the member fields!

```
name2 = name;   // copies members Only
```
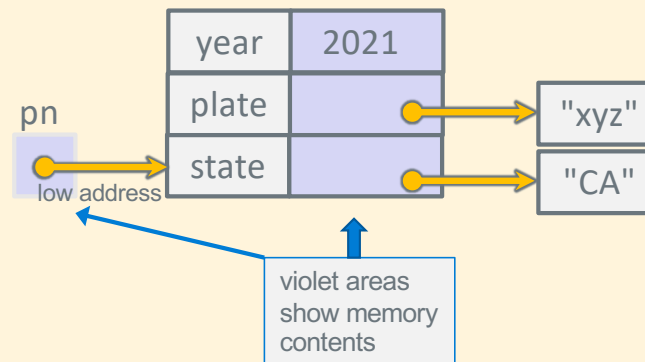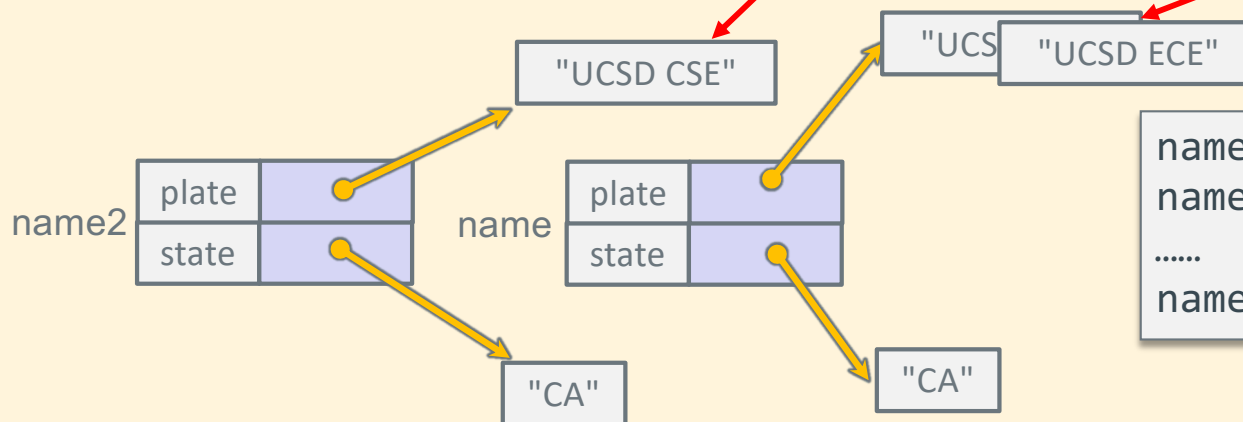
"UCSD ECE"
UCSD CSE

| plate | ● |
| state | ● |

name2

| plate | ● |
| state | ● |

name

"CA"
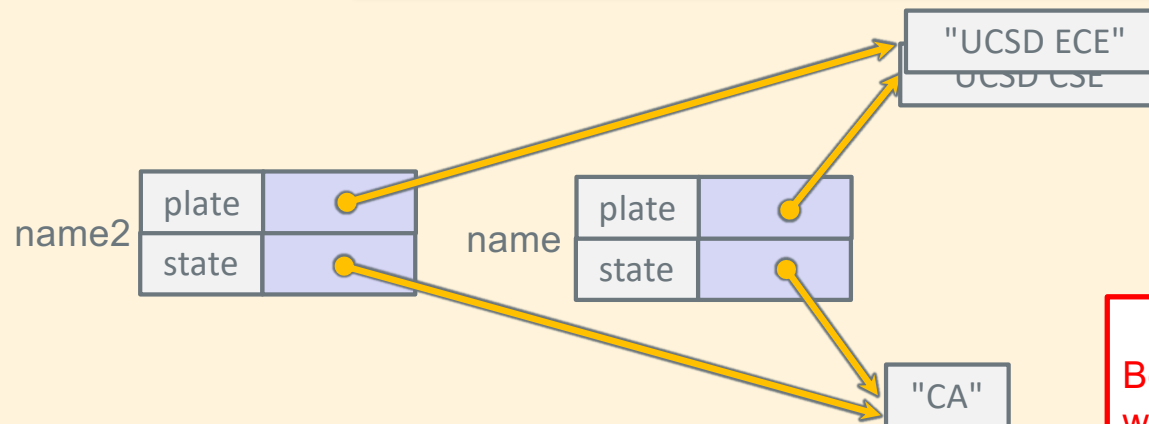
```
name2.plate = "UCSD ECE";
```

Warning
Be **very careful** with "shallow copies" in C
when pointers are involved

22

X

Memory Allocation Structs with Pointer Members

- **Safety first:** Allocate anything that is pointed at by a struct member independently (they are not part of the struct, only the pointers are)

```c
struct vehicle {
  char *state;
  char *plate;
  int year;
};
struct vehicle name1;
pn = &name1;
```

pn

low address

year	2021
plate	
state	

"xyz"

"CA"

violet areas show memory contents

```c
name1.state = strdup("CA");
pn->plate = strdup("xyz");
pn->year = 2021;
```

X

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

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

```
struct vehicle name  = {"CA", "UCSD CSE"};
struct vehicle name2;
```

mutable strings (heap memory)

- Use strdup() to copy the strings

immutable strings (read-only data)

"UCSD CSE"

"UCS   "UCSD ECE"

plate	
state	

name2

plate	
state	

name

"CA"

"CA"

```
name2.plate = strdup(name.plate);
name2.state = strdup(name.state);
……
name.plate = "UCSD ECE";
```

X

Nesting 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;
};
```
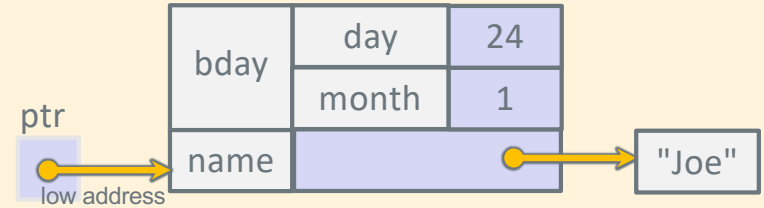


```
struct person first;
struct person *ptr;
ptr = &first;

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;
```

X

Version 1.06

UCSD CSE 30

Computer Organization and Systems Programming

C Programming Part 4

Lecture 11 – Oct 26, 2022

Keith Muller

Xerox Sigma 7 1970

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;
};
```
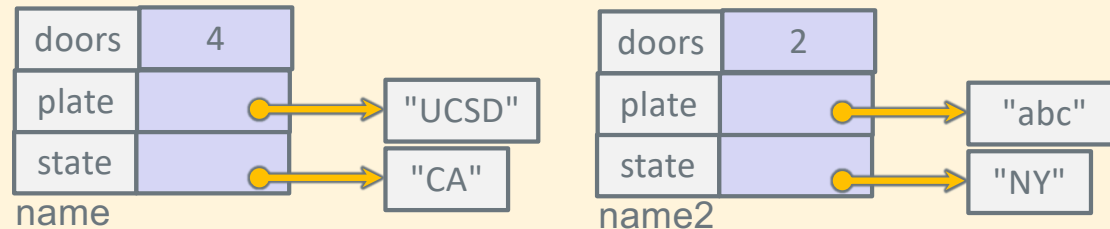


```
struct vehicle name  = {"CA", "UCSD", 4};
struct vehicle name2 = { (char []) {"NY"}, (char []) {"abc"}, 2};
```

```
if ((strcmp(name.state, name2.state) == 0) &&
    (strcmp(name.plate, name2.plate) == 0) &&
    (name.doors == name2.doors)) {
    printf("Same\n");
} else {
    printf("Different\n");
}
```

X

Struct: Arrays and Dynamic Allocation

- Allocate individual structs and arrays of structs using malloc()
  - Remember . is higher precedence than *:

```
#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;
```

ptr1

day	21
month	2

ptr2+4

day	
month	

ptr2+3

day	
month	

ptr2+2

day	
month	

ptr2+1

day	22
month	

ptr2

day	25
month	12

low address

x
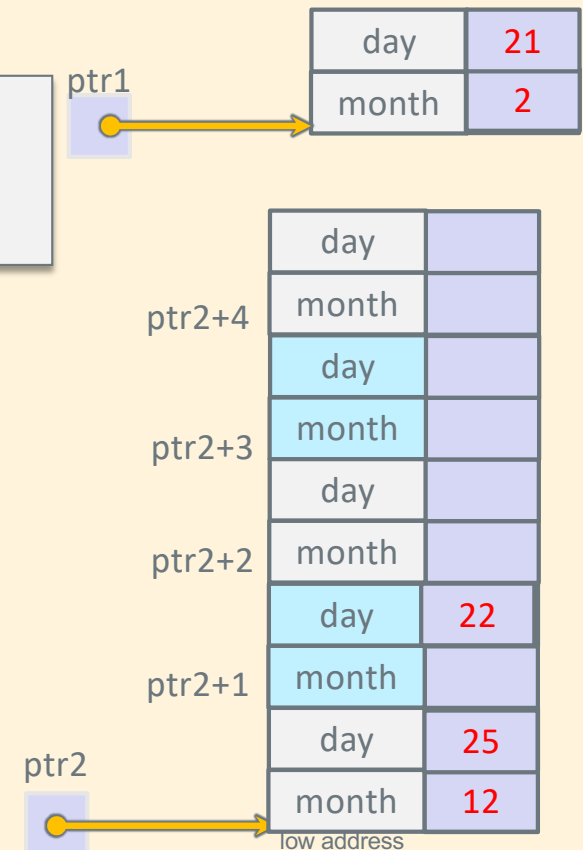
Struct As A Parameter to Functions

- 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, being an array is always a pointer)
  - For me, I always use pointers regardless of size, but that is just maybe a decades old habit…

X

Struct as a Parameter to Functions – Be Careful it is not like arrays
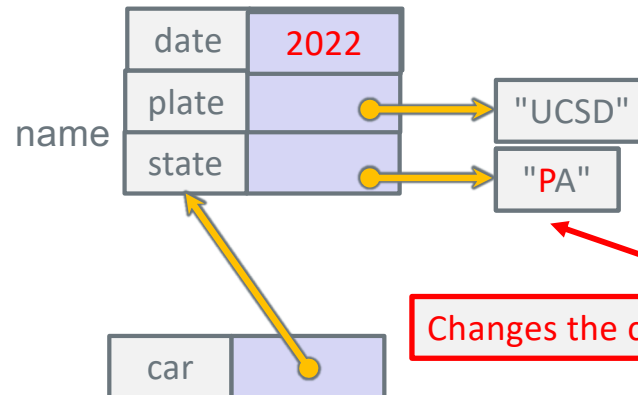
```
void change1(struct vehicle car)
{
    car.date = 2022; // oops!
  *(car.state) = "P";
}
…
change1(name);
```

Changes two different structs

date	2021
plate	
state	

name

"UCSD"

"PA"

date	2022
plate	
state	

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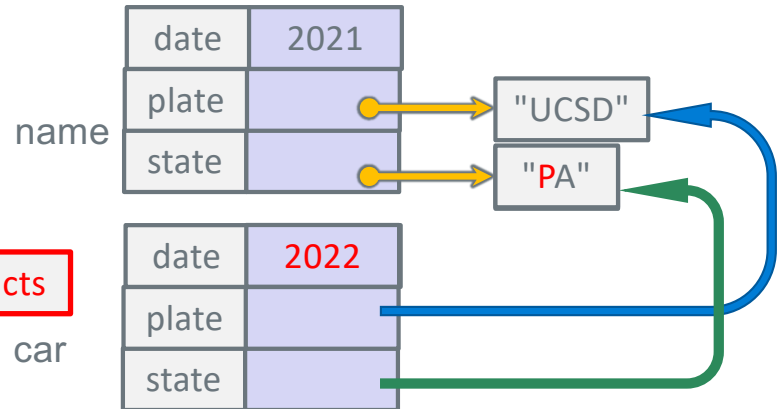
```
void change2(struct vehicle *car)
{
    car->date = 2022;
  *(car->state) = "P";
}
…
change2(name);
```

date	2022
plate	
state	

name

"UCSD"

"PA"

car	

Changes the caller's version

Struct as an Output Parameter: Deep Copy Example

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

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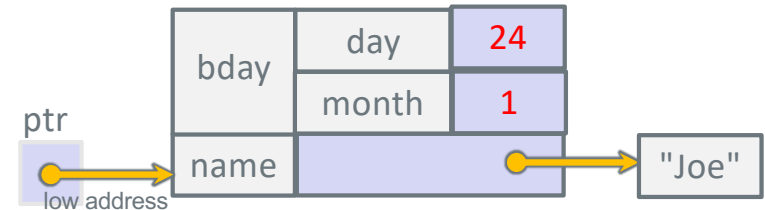


```
int fill(struct person *ptr, char *name, int month, int day)
{
    ptr->bday.month = month;
    ptr->bday.day = day;
    if ((ptr->name = strdup(name)) == NULL)
        return -1;
    return 0;
}
…-----------------calling function ----------
    struct person first;
    if (fill(&first, "Joe", 1, 24) == 0)
        printf("%s %d %d\n", first.name, first.bday.month, first.bday.day);
…
```

X

Review: Singly Linked Linked List - 1

head

| Payload Z | • | → | Payload Y | • | → | Payload X | ∅ |

- Is a **linear collection of nodes** whose order is not specified by their relative location in memory, like an array

- Each node consists of a payload and a pointer to the next node in the list
  - The pointer in the last node in the list is NULL (or 0)
  - The head pointer points at the first node in the list (the head is not part of the list)

- Nodes are easy to insert and delete from any position without having to re-organize the entire data structure

- Advantages of a linked list:
  - Length can easily be changed (expand and contract) at execution time
  - Length does not need to be known in advance (like at compile time)
  - List can continue to expand while there is memory available

X

Review: Singly Linked Linked List - 2

head

| 1933 | ● | 1955 | ● | 2020 | ∅ |

- Memory for each node is typically allocated dynamically at execution time (*i.e.,* using *heap memory – malloc() etc.*) when a new node is added to the list

- Memory for each node may be freed at execution time, using free() when a node is removed from the list

- Unlike arrays, linked list nodes are usually **not** arranged (located) sequentially in adjacent memory locations

- No fast and convenient way to "jump" to any specific node.

- Usually the list must be **traversed (walked)** from the *head* to locate if a *specific payload* is stored in any node

- Obviously, the cost in traversing a linked list is O(n)

X

Linked List Using Self-Referential Structs

- A **self-referential struct** is a struct that has one or more members that are **pointers** to a **struct variable of the same type**
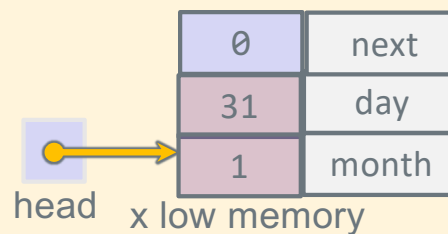
- Self-referential member ⟶

```
struct node {
    int date;
    struct node *next;
};
```

  - points to same type – itself
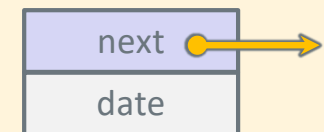
- Example:

```
struct node2 {
    int month;
    int day;
    struct node2 *next;
};
struct node2 x;
```

```
struct node2 *head
head = &x;
```

0	next
31	day
1	month

head   x low memory

```
x.month = 1;
x.day = 31;
x.next = NULL;
```

next
date

Creating a Node & Inserting it at the Front of the List

```c
// create node; insert at front when passed head
struct node *
creatNode(int year, char *name, struct node *link)
{
    struct node *ptr = malloc(sizeof(*ptr));
    if (ptr != NULL) {
        if ((ptr->name = strdup(name)) == NULL) {
            free(ptr);
            return NULL;
        }
        ptr->year = year;
        ptr->next = link;
    }
    return ptr;
}
```

Must duplicate the string because of buffer reuse

```c
struct node {
    int year;
    char *name;
    struct node *next;
};
```

```c
// calling function body
struct node *head = NULL; // insert at front
struct node *ptr;
char buf[BUFSZ];

if (fgets(buf, BUFSZ, stdin) != NULL) { // reads joe
    if ((ptr = creatNode(2020, buf, head)) != NULL) {
        head = ptr; // error handling not shown
    }
}
if (fgets(buf, BUFSZ, stdin) != NULL) {  // reads sam
    if ((ptr = creatNode(1955, buf, head)) != NULL) {
        head = ptr; // error handling not shown
    }
}
```
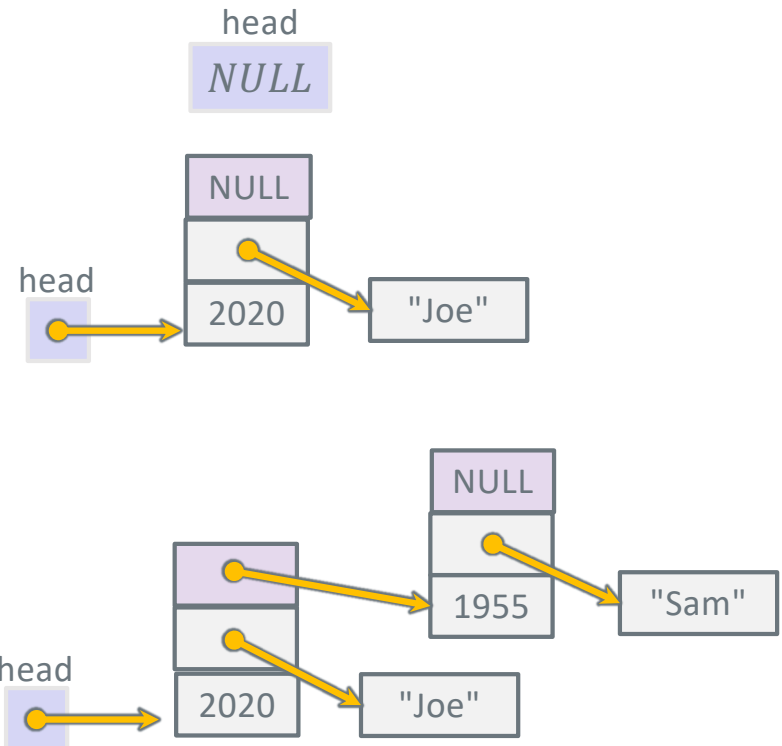

head
NULL

head → [NULL | 2020] → "Joe"

head → [• | 1955] → "Sam"   [NULL | 2020] → "Joe"

35

X
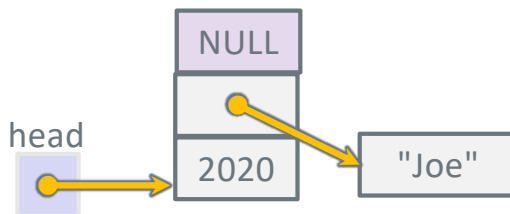
Creating a Node & Inserting it at the End of the List

```c
// create a node and insert at the end of the list
struct node *
insertEnd(int year, char *name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL;  // base case
    struct node *new;

    if ((new = creatNode(year, name, NULL)) == NULL)
        return NULL;

    while (ptr != NULL) {
        prev = ptr;
        ptr = ptr->next;
    }
    if (prev == NULL)
        return new;
    prev->next = new;
    return head;
}
```
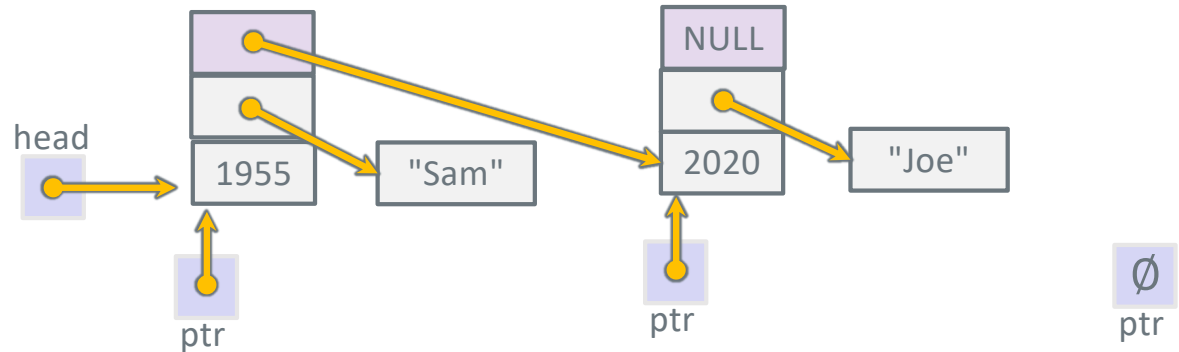
head

NULL

NULL

head

2020 → "Joe"

NULL

1955 → "Sam"

head

2020 → "Joe"

```c
struct node *head = NULL;   // insert at end
struct node *ptr;
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)
    head = ptr;
```

36

X

"Dumping" the Linked List
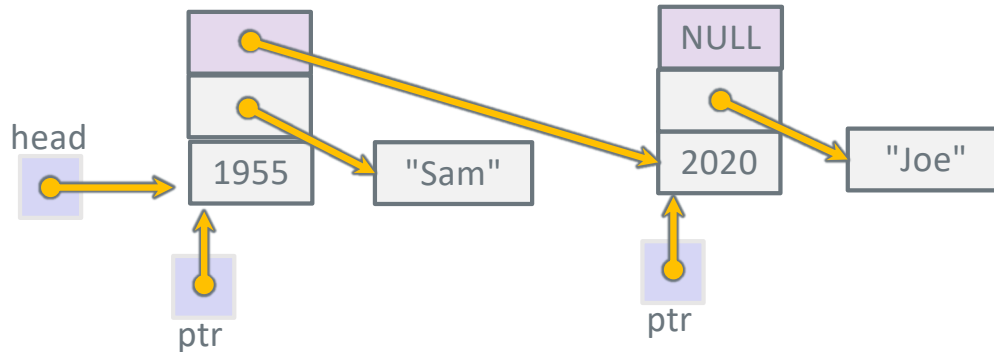
"walk the list from head to tail"

```
struct node {
  int year;
  char *name;
  struct node *next;
};
```

head

1955   "Sam"   NULL   2020   "Joe"

ptr   ptr   Ø
                ptr

```
struct node *head;
struct node *ptr;
…
printf("\nDumping All Data\n");
ptr = head;
while (ptr != NULL) {
    printf("year: %d name: %s\n", ptr->year, ptr->name);
    ptr = ptr->next;
}
```

```
Dumping All Data
year: 1955 name: Sam
year: 2020 name: Joe
```

37

X

Finding A Node Containing a Specific Payload Value


```
struct node {
    int year;
    char *name;
    struct node *next;
};
```

head

1955   "Sam"   2020   "Joe"

NULL

ptr   ptr

```
struct node *findNode(char *name, struct node *ptr)
{
    while (ptr != NULL) {
        if (strcmp(name, ptr->name) == 0)
            break;
        ptr = ptr->next;
    }
    return ptr;
}
```

Returns pointer if found
NULL otherwise

```
struct node *found;

if ((found = findNode("Joe", head)) != NULL)
    printf("year: %d name: %s\n", found->year, found->name);
```

38

X

Deleting a Node in a Linked List

```c
// returns head pointer; may have changed…
struct node *deleteNode(int name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL;

    while (ptr != NULL) {
        if (strcmp(name, ptr->name) == 0)
            break;
        prev = ptr;
        ptr = ptr->next;
    }
    if (ptr == NULL) // not found return head
        return head;
    if (ptr == head) // remove first node new head
        head = ptr->next;
    else
        prev->next = ptr->next; // remove not head
    free(ptr->name);  // free strdup() space
    free(ptr);
    return head;
}
```
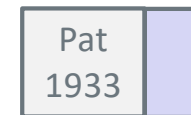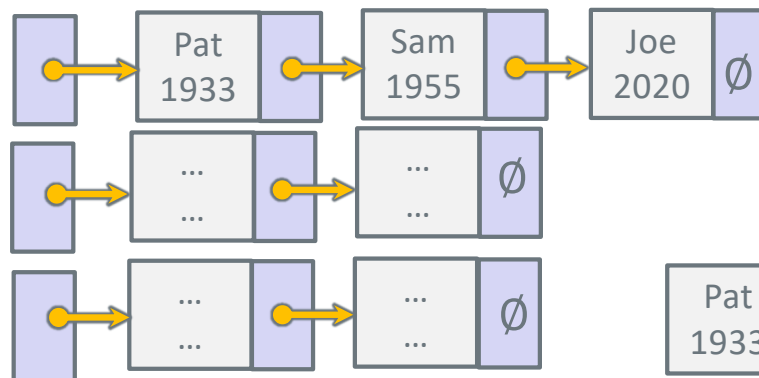
head

1955   "Sam"   NULL

2020   "Joe"

ptr   ptr

head

NULL

1955   "Sam"

ptr
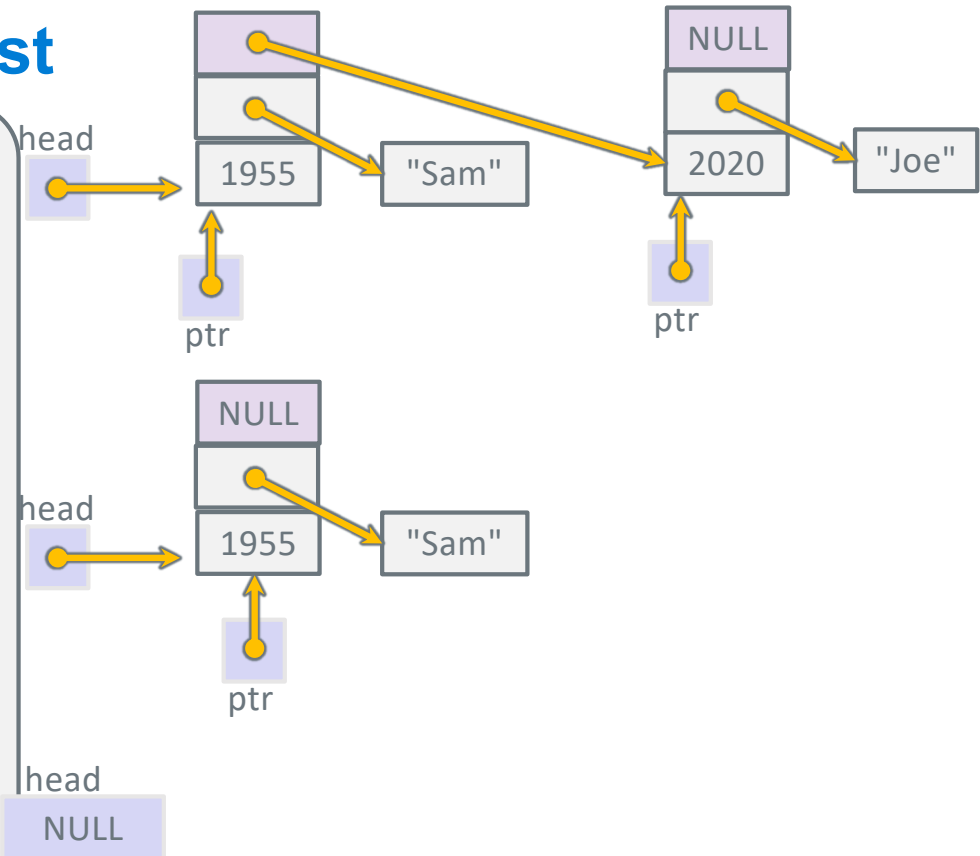
head

NULL

```c
struct node *head = NULL;
head = deleteNode("Joe", head);
head = deleteNode("Sam", head);
```

X

Improving On Linked List Performance
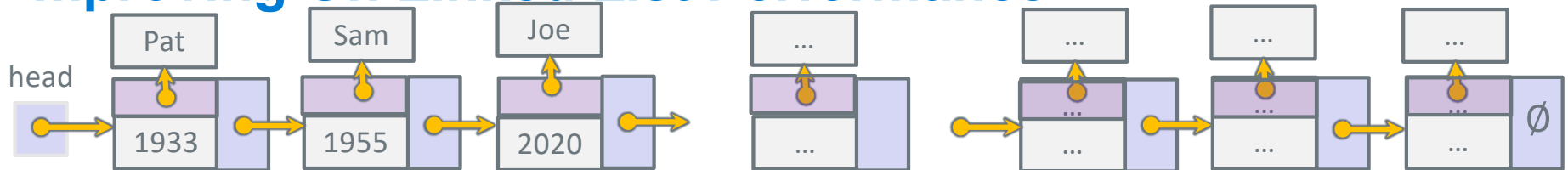
- When linked lists get long, the cost of finding an entry continues to increase O(n)

- How to improve search time?

- Break the single linked list into multiple **shorter length** linked lists
  - Shorter lists are faster to search

- **Requires a function** that takes a lookup key and selects just one of the shortened lists

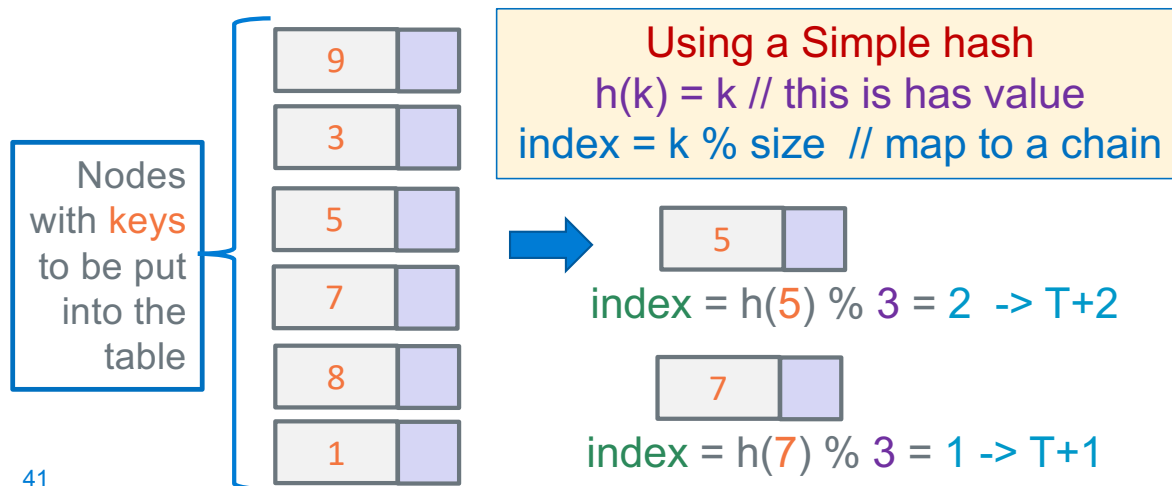How do you determine on which linked list an entry is stored?
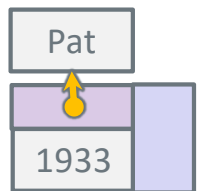
x

Hashing

- Hash table is an array of pointers to the head of different linked lists (called hash chains)

- **Each data item must have a unique key that identifies it (e.g., auto license plate)**
  - $h(k)$ is the *hash value* of key k to *encode the key k into an integer*

- Use the Hash value to map to **one entry** in the hash table T[ ] of size TABLESIZE
  - Index = $h(k)$ % TABLESIZE   (mod operator % maps a keys hash value to table index)

- Keys that hash to the same array index (*collide*) are put on a linked list

- After hashing a key, you then traverse the selected linked list to find the entry

Nodes with keys to be put into the table

9	
3	
5	
7	
8	
1	

Using a Simple hash
h(k) = k // this is has value
index = k % size // map to a chain

| 5 | |

index = h(5) % 3 = 2 -> T+2

| 7 | |

index = h(7) % 3 = 1 -> T+1

Hash Table T[3] of linked list head pointers
TABLESIZE = 3

*(T+2) → | 5 | • | → | 8 | Ø |

*(T+1) → | 7 | • | → | 1 | Ø |

*T → | 3 | • | → | 9 | Ø |

41

X
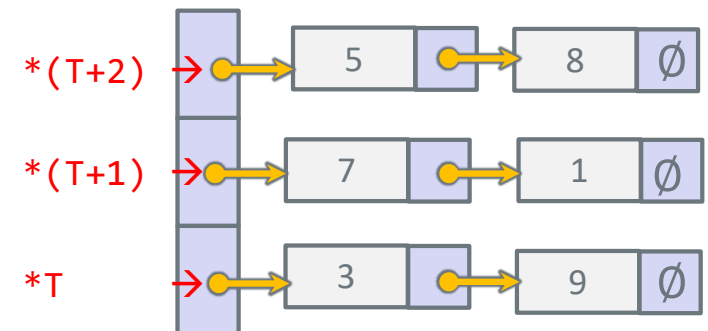
Hash Table With Collision Chaining (multiple linked lists)

Hash Table T[3]
TABLESIZE = 3
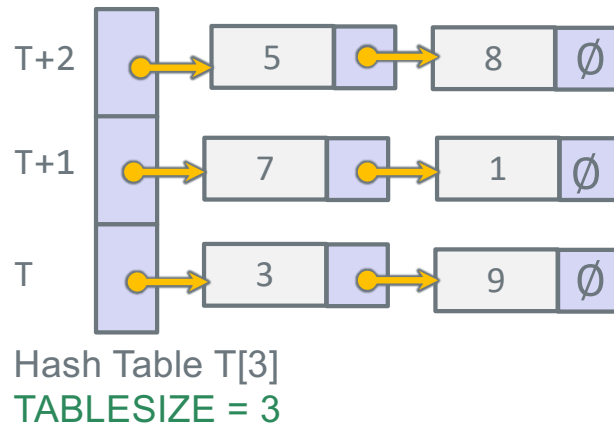
- Make TABLESIZE *prime as* keys are typically not randomly distributed, and have a *pattern*
  - Mostly even, mostly multiples of 10, etc.
  - In general: mostly multiples of some k
- If k is a factor of TABLESIZE, then only (TABLESIZE/k) slots will ever be used!

1. Calculate index `i = hash(key) % TABLESIZE`

2. Go to array element `i`, i.e., `T+i` that contains the head pointer for collision chain

3. Walk the linked list for element, add element, remove element, etc. from the linked list

4. New items added to the hash table are typically added at the front or at the end of the *collision chain* linked list *(when multiple keys hash to same index .. they **collide**)*

- **Hash arrays** need an **index number to select a chain**, so if we have a string, we must first convert to a number
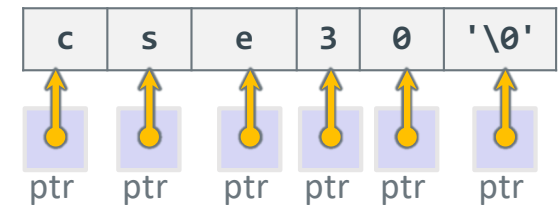
x

PA5: A Simple 32-bit String Hash Function in C (djb2)

```c
uint32_t hash(char *str)
{
    uint32_t hash = 0U;
    uint32_t c;

    while ((c = (unsigned char)*str++) != '\0')
        hash = c + (hash << 6) + (hash << 16) - hash;

    return hash;
}
```

Signed Data types	Unsigned Data types	Exact Size
int32_t	uint32_t	32 bits (4 bytes)

c	s	e	3	0	'\0'

| ptr | ptr | ptr | ptr | ptr | ptr |

- Many different algorithms for string hash function (Dan Berman's djb2 above)
  - << is the **left** bit shift operator (later in course)
- **Fast to compute**, has a reasonable key distribution for short 8-bit ASCII strings into 32-bit unsigned ints

43

X

Allocating the Hash Table (collision chain head pointers)
Good use for calloc()

```c
#define TBSZ 3
int main(void)
{
    struct node *ptr;
    struct node **tab;
    uint32_t index;

    if ((tab = calloc(TBSZ, sizeof(*tab))) == NULL) {
        fprintf(stderr,"Cannot allocate hash table\n");
        return EXIT_FAILURE;
    }
// continued on next slide
```

tab+2 | ∅
tab+1 | ∅
tab | ∅

TABLESIZE = 3

x

Inserting Nodes into the Hash Table (at the end)

```
#define TBSZ 3
unit32_t index;

index = hash("Pat") % TBSZ;
if ((ptr = insertEnd(1933, "Pat", *(tab + index))) != NULL)
    *(tab + index) = ptr;


index = hash("Sam") % TBSZ;
if ((ptr = insertEnd(1955, "Sam", *(tab + index))) != NULL)
    *(tab + index) = ptr;


index = hash("Sally") % TBSZ;
if ((ptr = insertEnd(2020, "Sally", *(tab + index))) != NULL)
    *(tab + index) = ptr;


index = hash("Zoe") % TBSZ;
if ((ptr = insertEnd(2022, "Zoe", *(tab + index))) != NULL)
    *(tab + index) = ptr;
```
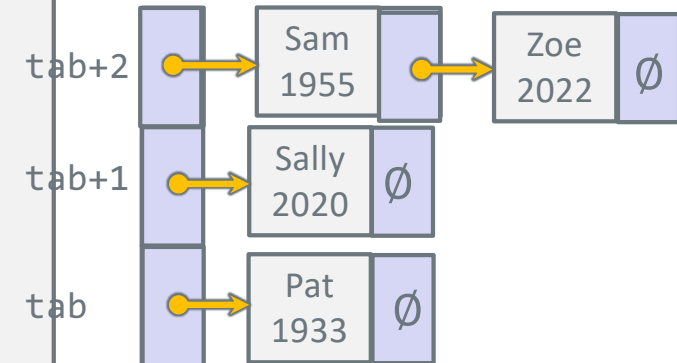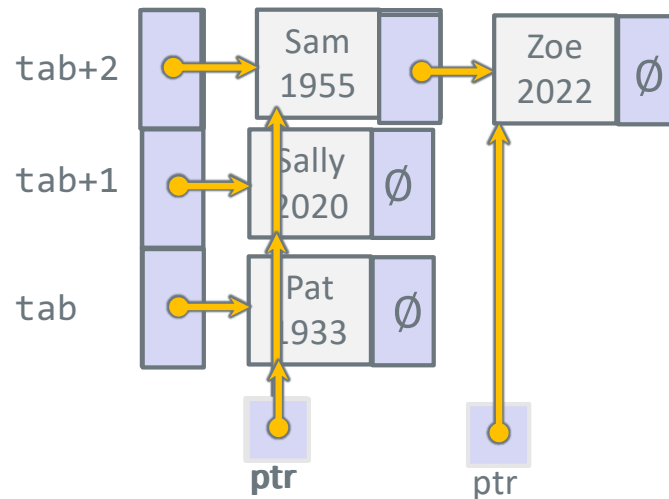
```
struct node {
    int year;
    char *name;
    struct node *next;
};
```


Notice
Substitute **createNode()** for **insertEnd()** to insert nodes at the **front** of the collision chain **instead** of at the **end** of the collision chain
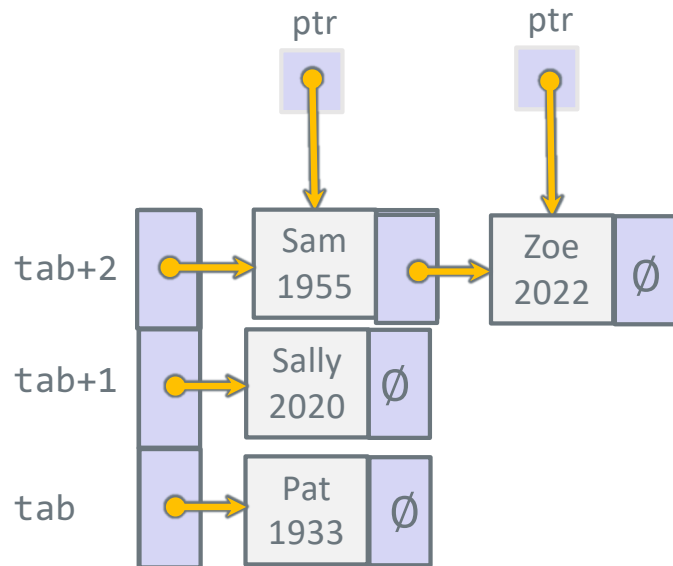
45

X

"Dumping" the Hash Table (traversing all Nodes)


```
Dumping All Data
chain: 0
Year: 1933 Name: Pat
chain: 1
Year: 2020 Name: Sally
chain: 2
Year: 1955 Name: Sam
Year: 2022 Name: Zoe
```

```c
printf("\nDumping All Data\n");
for (index = 0U; index < TBSZ; index++) {
    ptr = *(tab + index);
    printf("chain: %d\n", index);

    while (ptr != NULL) {
        printf("Year: %d Name: %s\n", ptr->year, ptr->name);
        ptr = ptr->next;
    }
}
```

X

Finding a Node with a Specific Payload Value


```c
// same routine as shown in a previous slide
struct node *findNode(char *name, struct node *ptr)
{
    while (ptr != NULL) {
        if (strcmp(name, ptr->name) == 0)
            break;
        ptr = ptr->next;
    }
    return ptr;
}
```

```c
index = hash("Zoe") % TBSZ;
if ((ptr = findNode("Zoe", *(tab + index))) != NULL)
    printf("Found Year: %d name: %s\n", ptr->year, ptr->name);
else
    printf("Not Found Zoe\n");
```
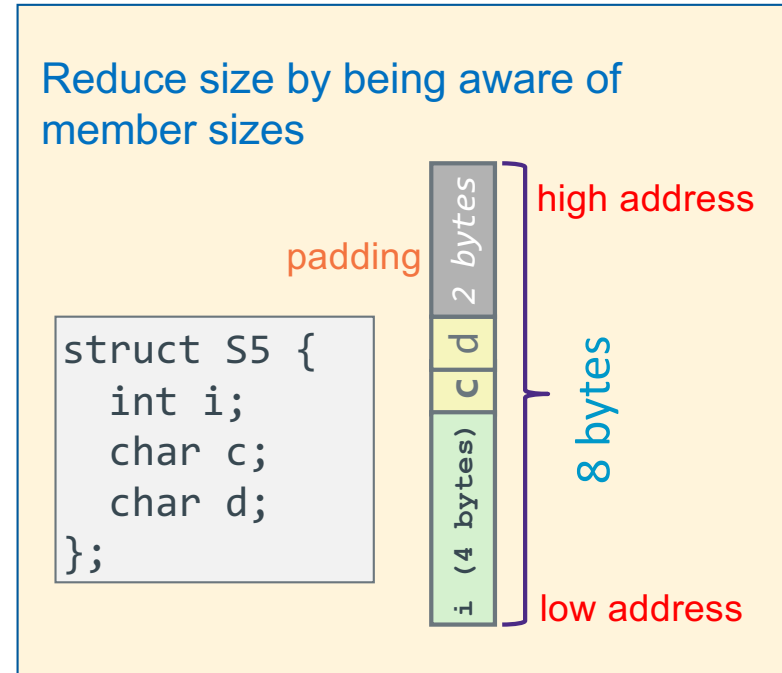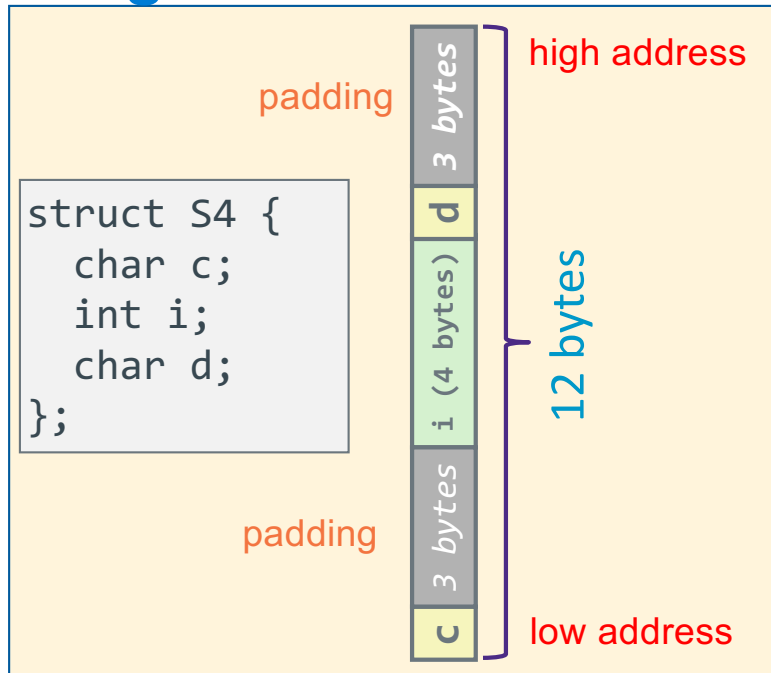
X

Sizing Struct Members

- struct size depends on the order of the fields listed in the struct

```
struct S4 {
  char c;      // byte aligned
  int i;       // 4 byte aligned
  char d;      // byte aligned
};
```

- Structs uses contiguously-allocated region of memory,

- compilers **are required to follow member order**, and HW alignment requirements

  1. not allowed to re-arrange member order in memory
  2. struct starting address: aligned to the requirements of largest member
  3. Add memory space between members (pad or unused space), so the next member starts at the required memory alignment
  4. Structs may add padding so total size is always a whole multiple of the size of the largest member (for struct arrays)

X

Re-Sizing Struct Members

```
struct S4 {
   char c;
   int i;
   char d;
};
```

padding

3 bytes

d

i (4 bytes)

3 bytes

c

high address

12 bytes

padding

low address

Reduce size by being aware of member sizes

```
struct S5 {
   int i;
   char c;
   char d;
};
```

padding

2 bytes

d

c

i (4 bytes)

high address

8 bytes

low address

- re-order the fields to decrease space wasted by member alignment padding

- Remember by C specifications, the compiler will not do this for you…

- To get the byte offset (from the start) of any member of a struct

```
#include <stddef.h>
size_t cnt = offsetof(struct_name, member_name);
```

X