Version 2.03

UCSD CSE 30

Computer Organization and Systems Programming

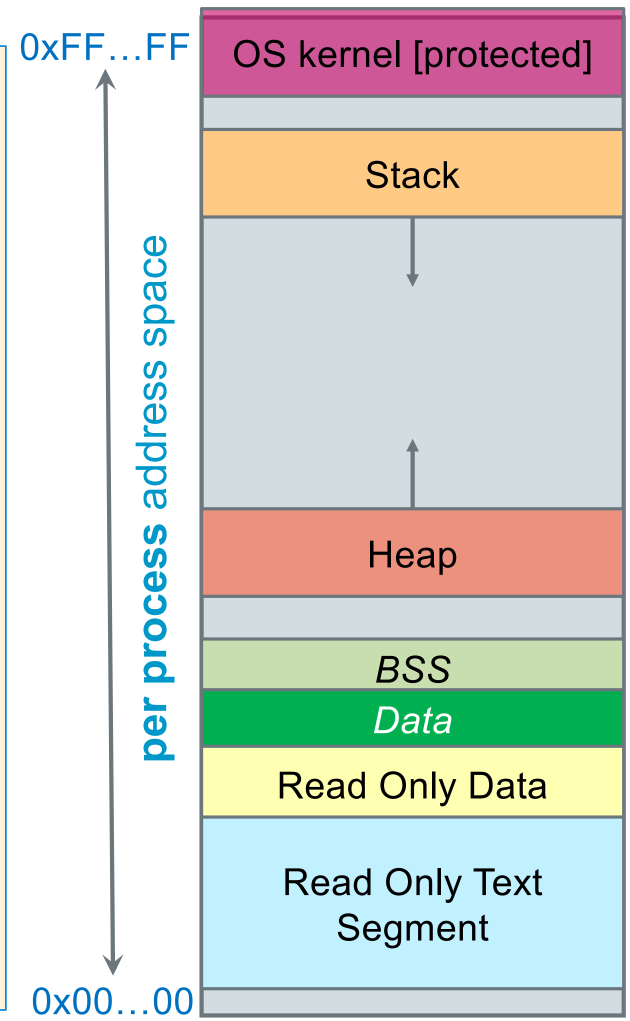
C Part 3

Keith Muller

Vax 11/780 1980

Process Memory Under Linux

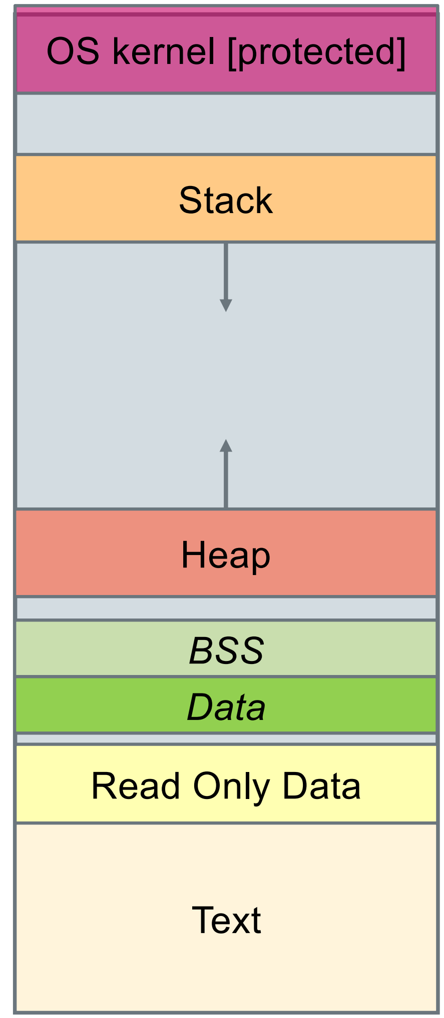
- When your program is running it has been loaded into memory and is called a process
- *Stack segment:* *Stores Local variables*
  - Allocated and freed at function call entry & exit
- *Data segment + BSS:* *Stores Global* and *static* variables
  - Allocated/freed when the process starts/exits
  - 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



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



x

Heap Dynamic Memory Allocation Library Functions

#include <stdlib.h>	args	Clears memory at runtime
void *malloc(…)	size_t size	no
void *calloc(…)	size_t nmemb, size_t memsize	yes
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**
  - When assigned to a typed pointer, it *"converts"* it from a void * to the type of the pointer variable
- **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**

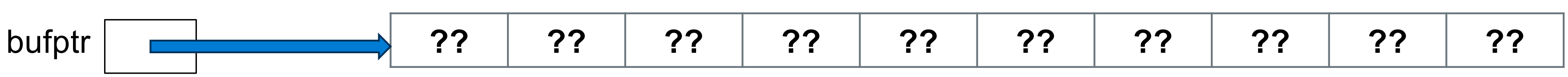
x

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**

```
    char *bufptr;

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

    if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {
        fprintf(stderr, "Unable to malloc memory");
        return NULL;
    }
// allocates a character array with 10 elements
```



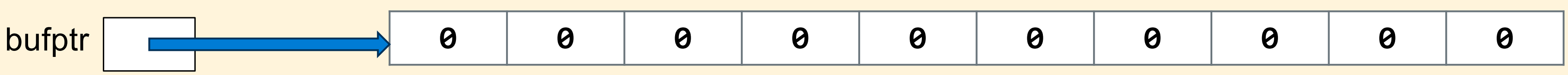
x

Calloc()

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

- `calloc()` variant of **`malloc()`** but zeros out every byte of memory during program execution 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

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 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;
  }
  // other code
  free(bufptr);                    // returns memory to the heap
  bufptr = NULL;                  // set bufptr to NULL
```


X

Mis-Use of Free() - 1

- Call **`free()`**
  - With the same address that you obtained with malloc() (or other allocators)
  - It is NOT an error to pass **`free()`** a pointer to NULL

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes)) != NULL) {
    /* some code */
    free(bytes + 5); // Program aborts free(): invalid pointer
```

- **Freeing unallocated memory:** Only call free() to free memory address that you obtain from one of the allocators (malloc(), calloc(), etc.)

```
char *ptr = "cse30";
          …
    /* some code */
    free(ptr);           // Program aborts free(): invalid pointer
```


X

Mis-Use of Free() - 2

- Continuing to write to memory after you **`free() it`** is likely to corrupt the heap or return changed values
  - Later calls to heap routines (**`malloc(), calloc(), strdup())`** may fail or seg fault

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes)) != NULL) {
    /* some code */
    free(bytes);
    strcpy(bytes, "cse30");   // INVALID! used after free
……
```

- **Double Free:** Freeing allocated memory more than once will cause your program to abort (terminate)

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes)) != NULL) {
    /* some code */
    free(bytes);
    free(bytes); // Program abort double free detected….
……
```


X

More Dangling Pointers: Continuing to use "freed" memory

- Review: **Dangling 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;   // but it is returned to the caller anyway - bad
}
```

- `dangling_freed_heap()` **may cause** the allocators (**`malloc()`**and friends) to **seg fault when called later to allocate memory**
  - Why? Because it corrupts data structures the heap code uses to manage the memory pool (it often stores meta-data in the freed memory)

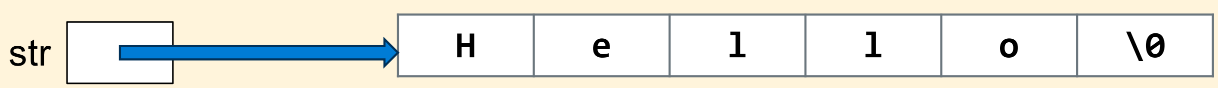
x

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 with strncpy();
- The caller is responsible for freeing this memory when no longer needed

```
char *str = strdup("Hello");
*str = 'h';  // str points at a mutable string

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



Heap Memory "Leaks"

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

```
void
leaky_memory (void)
{
    char *buf = malloc(BLKSZ * sizeof(*bytes));
…
    /* code that never calls free() to deallocates the memory */
    return;  // you lose the address in buf 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!)

x

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

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, 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!

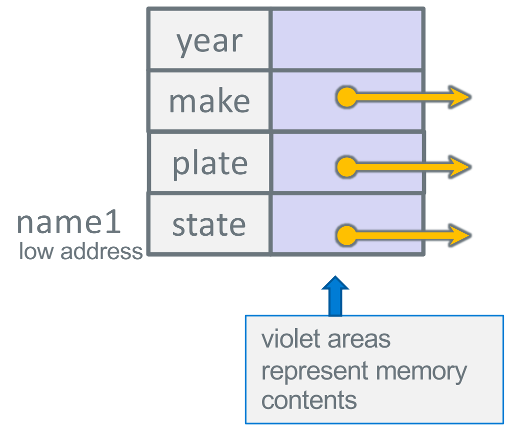
```
struct tagname {
  type1 member1;
  ...
  typeN memberN;
};
```

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


x

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

x

Accessing members of a struct

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

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

struct date type definition

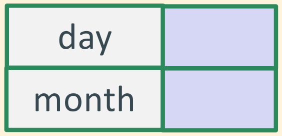
```
struct date bday; // define a struct instance
bday.month = 1;
bday.day = 24;

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

bday definition

x

Accessing members of a struct with pointers

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



- 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

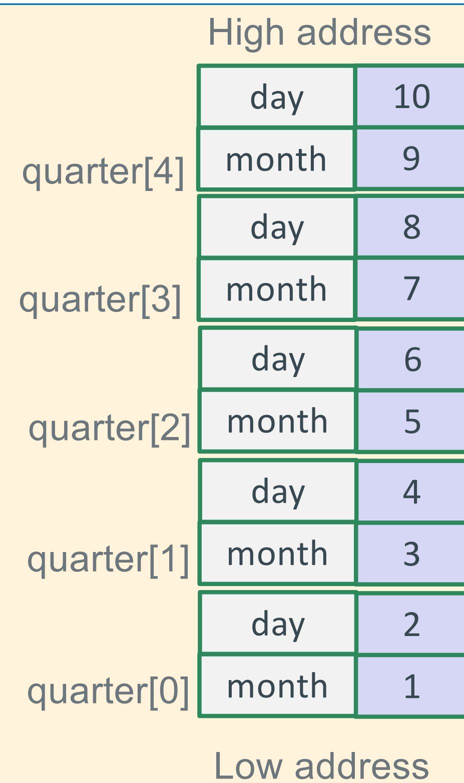
X

Accessing members of a struct

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

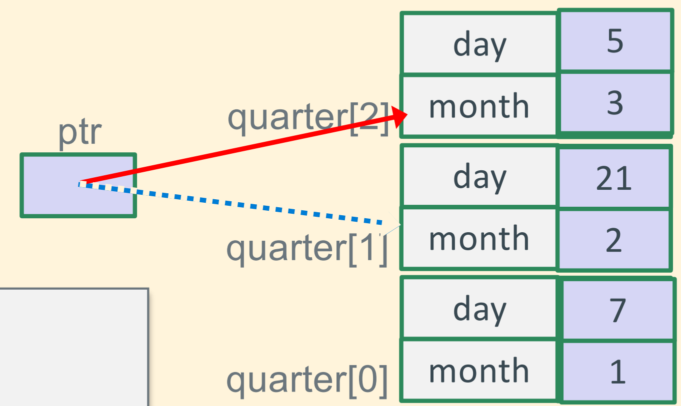
- You can create an array of structs and initialize them

```c
struct date quarter[] =
             { {1,2}, {3,4}, {5,6}, {7,8}, {9,10} };

int cnt = sizeof(quarter)/sizeof(*quarter);  // = 5
quarter[0].month = 1;
quarter[0].day = 2;
```



x

Accessing members of a struct




```c
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;
```


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;`
- *Some claim typedefs* are easier to understand than tagged struct variables, others not
  - 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;
```

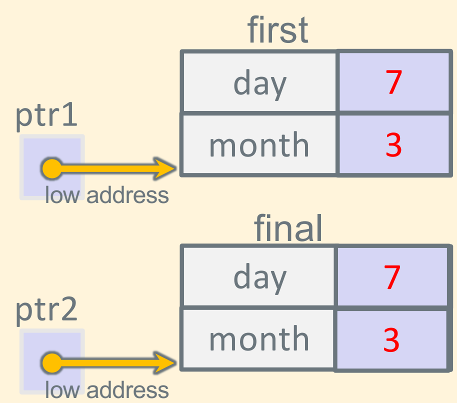
```
typedef struct {
    int a;
    int b;
} pair;

pair var3;
pair *ptr3;
```


X

Assigning Structs in an expression

- You can assign (copy) each member value of a struct from a struct of the same type
  Performance Caution: *this copies **the contents** of each struct member during execution*
- Individual members 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
```

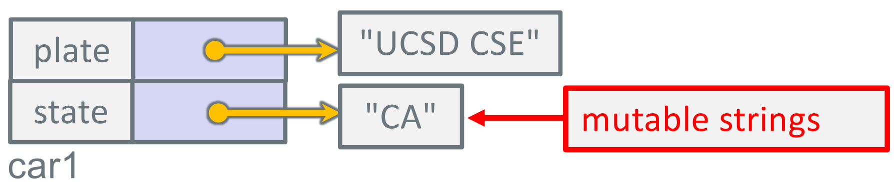


x

Caution: Assignment is a Shallow Copy of struct members

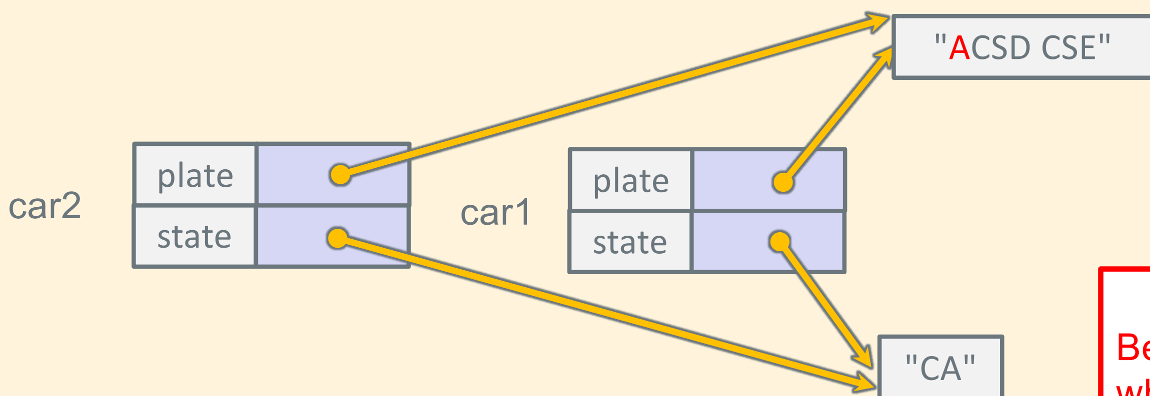
```c
struct vehicle {
  char *state;
  char *plate;
};
```

```c
struct vehicle car1  = {(char []) {"CA"}, (char []){"UCSD CSE"}};
struct vehicle car2;
```



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

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




```c
*(car2.plate) = 'A';
```

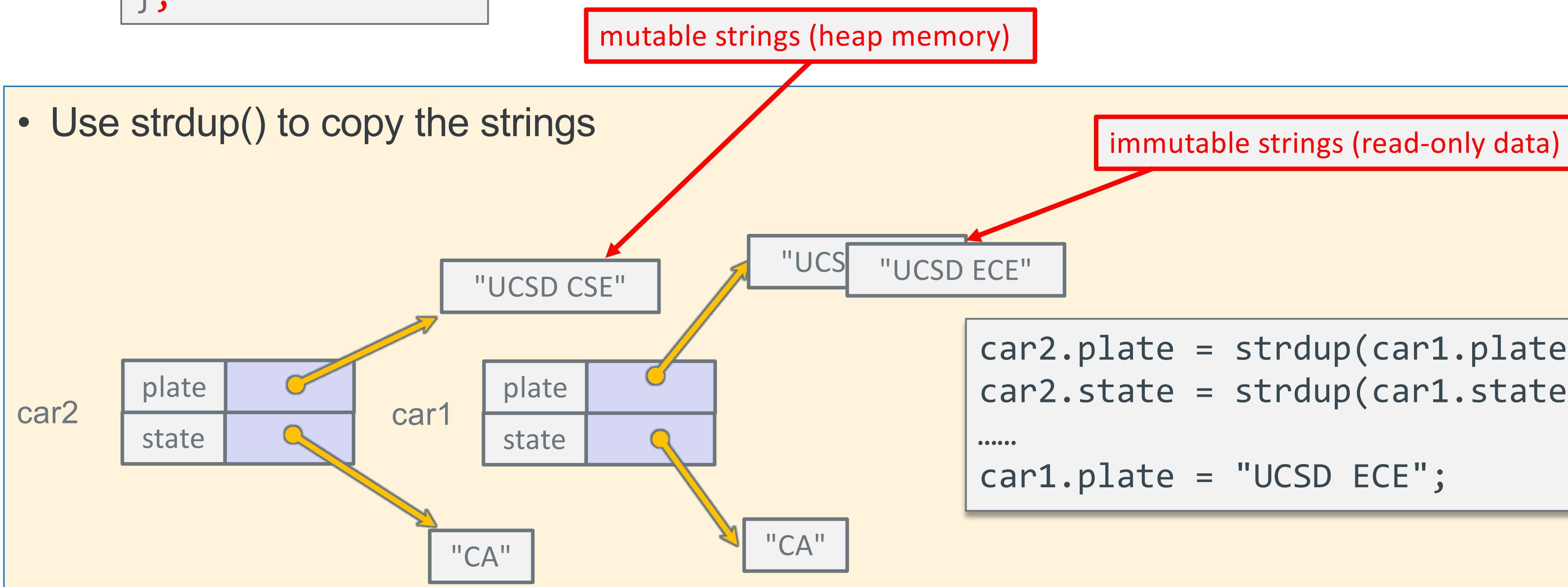
Warning
Be **very careful** with "shallow copies" in C when pointers are involved

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

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

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

- Use strdup() to copy the strings



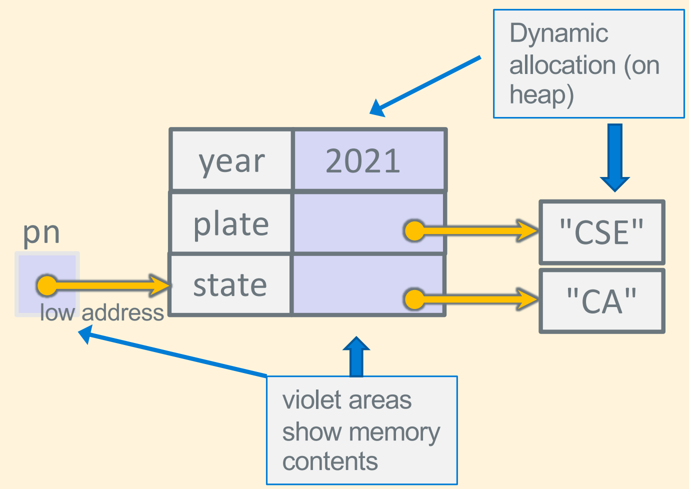

```c
car2.plate = strdup(car1.plate);
car2.state = strdup(car1.state);
……
car1.plate = "UCSD ECE";
```


x

Deep Copies of Structs

- **Must** first **allocate space** to be pointed at by member pointers **independently** (they are not part of the struct, only the pointers are) then copy what they point at

```
struct vehicle {
  char *state;
  char *plate;
  int year;
};
struct vehicle car1  = {"CA", "CSE", 2021};

struct vehicle *pn = malloc(sizeof(*pn));
// check if malloc() failed not shown
```




```
pn->state = strdup(car1.state);
pn->plate = strdup(car1.plate);
pn->year = car1.year;
```

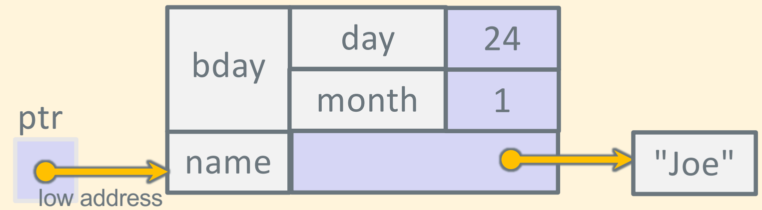

x

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;
};
```



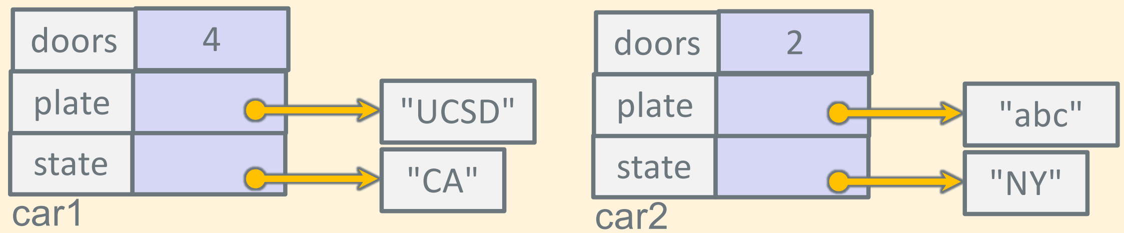

```
struct person first;
struct person *ptr;
ptr = &first;

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;
```


x

Comparing Two Structs

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

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

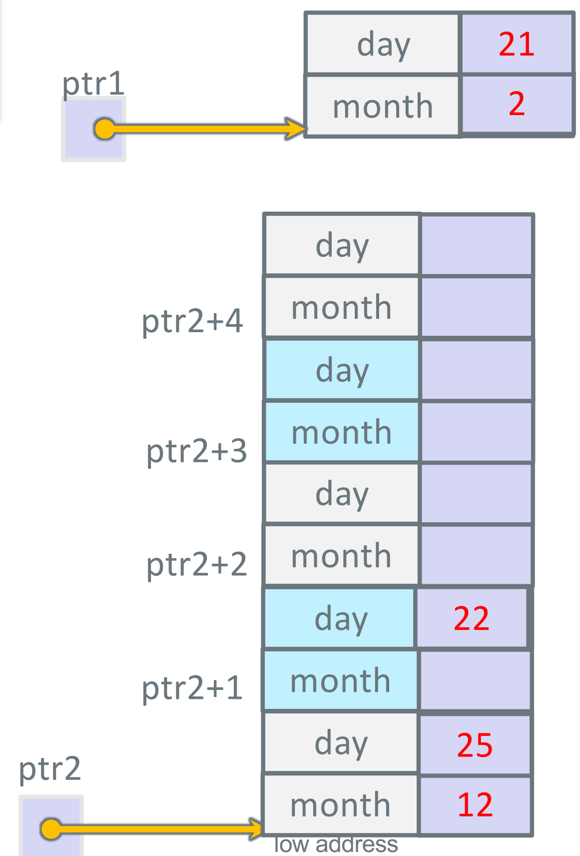



```c
struct vehicle car1  = {"CA", "UCSD", 4};
struct vehicle car2 = { (char []) {"NY"}, (char []) {"abc"}, 2};
```

```c
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);
```




```
(*pt1).month = 2;
(*pt1).day = 21;

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

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


x

Formal Parameter structs: contents set with shallow copies!

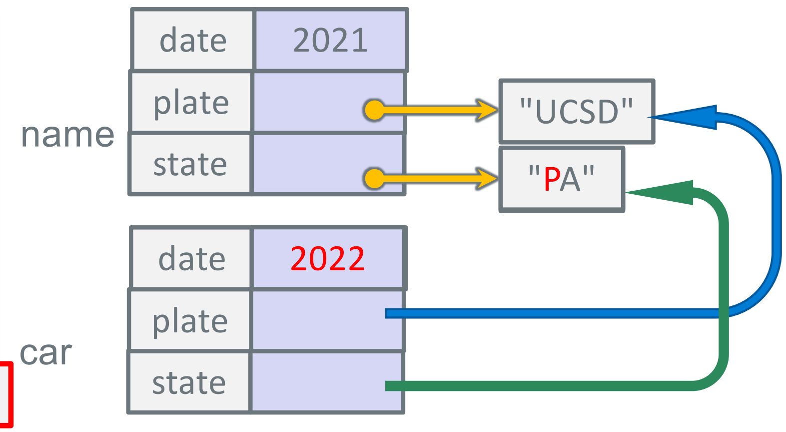
- WARNING: When you pass a struct, **you pass a copy** of all the struct members
  - This is a shallow copy (shallow copy – so if you have members that are pointers watch out)
- More often code will pass the pointer to a struct to avoid the copy costs
  - Be careful and not modify what the pointer points to (unless it is an output parameter)
- Tradeoffs:
  - Passing a pointer is faster and takes less space
  - 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**

x

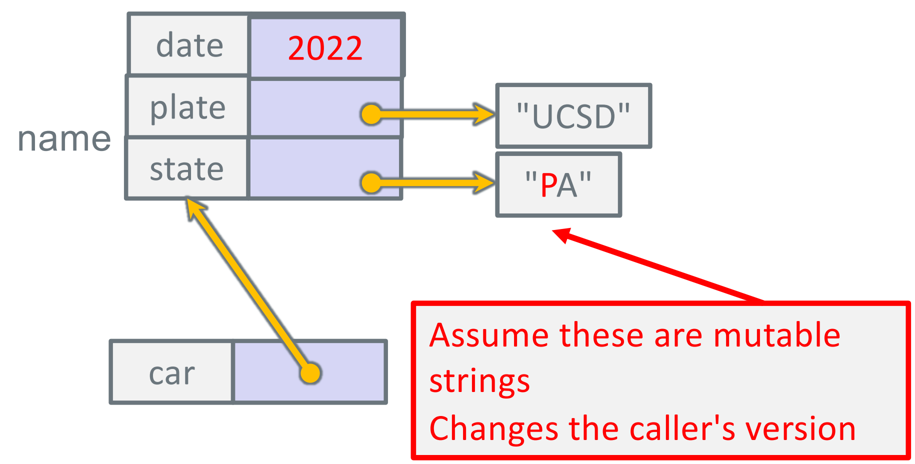
Struct Function Parameters – Be Careful it is not like arrays

```
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);
```

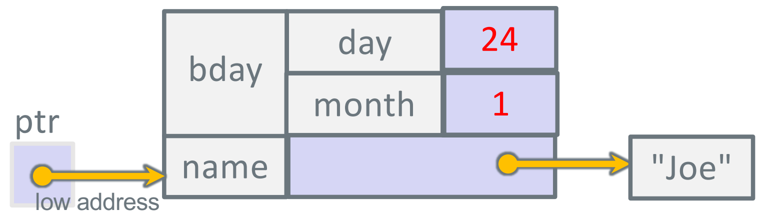


x

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

- 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

Review: Singly Linked Linked List - 2

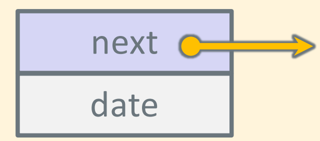
- Memory for each node is typically allocated dynamically at execution time (*i.e.,* using *heap memory – malloc() etc.*) when a new node is added to the list
- Memory for each node may be freed at execution time, using free() when a node is removed from the list
- Unlike arrays, linked list nodes are 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**

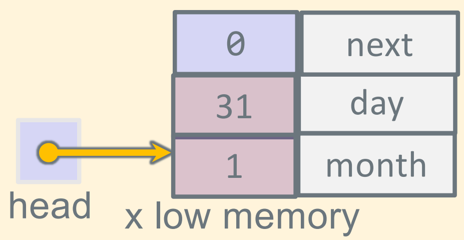
```
struct node {
  int date;
  struct node *next;
};
```



- Self-referential member
  - points to same type – itself
- Example:

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

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




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


x

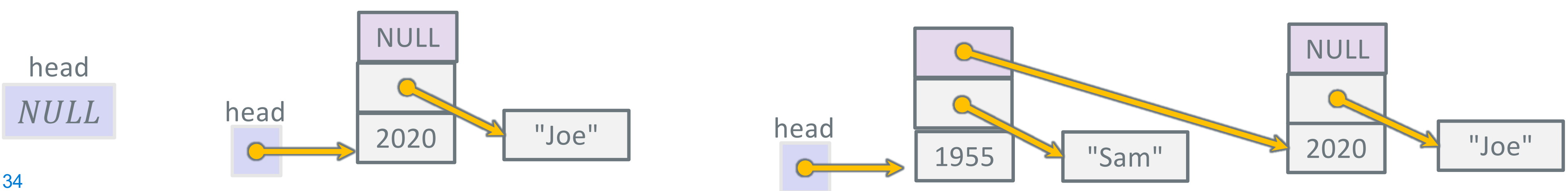
Creating a Node & Inserting it at the Front of the List

```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;
}
```

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

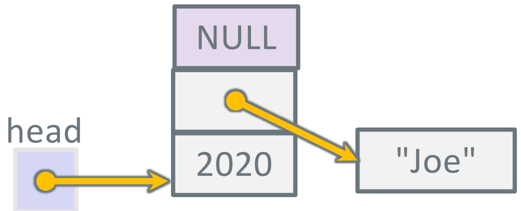
```c
// calling function body
struct node *head = NULL; // insert at front
struct node *ptr;

if ((ptr = creatNode(2020, "Joe", head)) != NULL) {
        head = ptr; // error handling not shown
}
if ((ptr = creatNode(1955, "Sam", head)) != NULL) {
        head = ptr; // error handling not shown
}
```

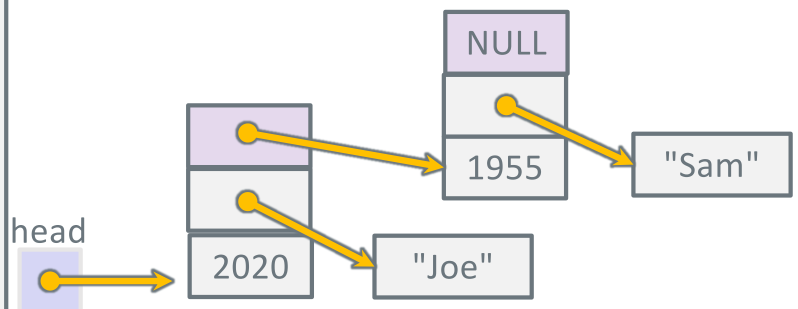
Never use head here!
you can lose your linked list
if creatNode() fails!



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

```
// 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;
}
```





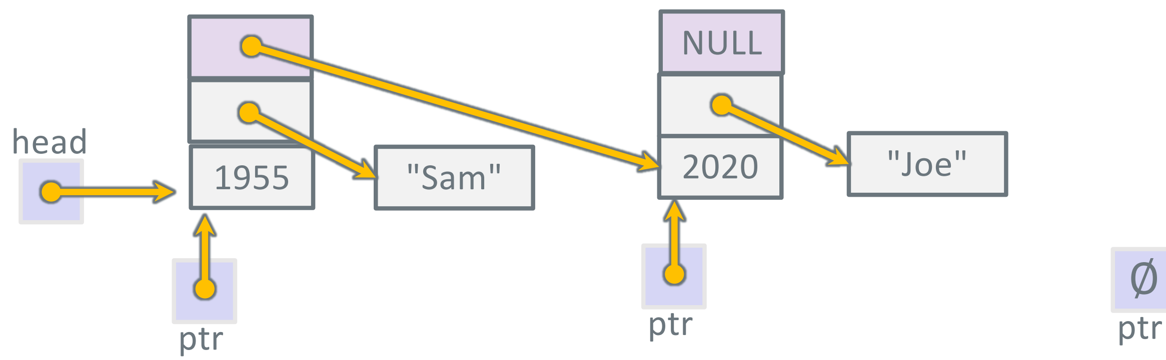

```
struct node *head = NULL;  // insert at end
struct node *ptr;
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)
    head = ptr;
```


x

"Dumping" the Linked List

"walk the list from head to tail"

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

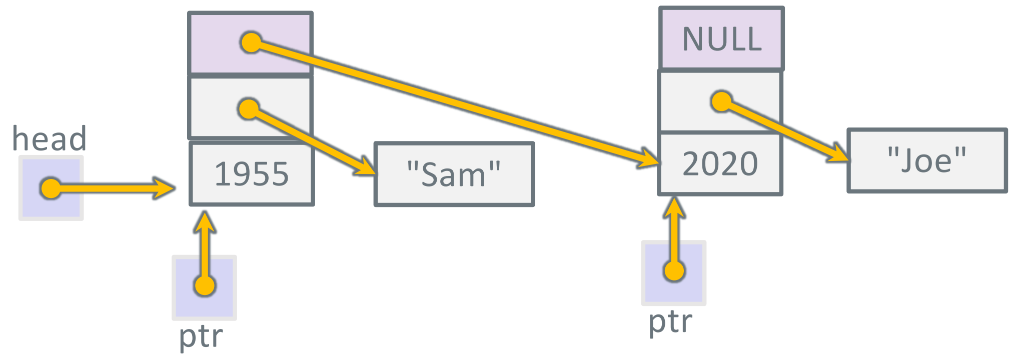



```
struct node *head;
struct node *ptr;
…
printf("\nDumping All Data\n");
ptr = head;
while (ptr != NULL) {
    printf("year: %d name: %s\n", ptr->year, ptr->name);
    ptr = ptr->next;
}
```

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


x

Finding A Node Containing a Specific Payload Value




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

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

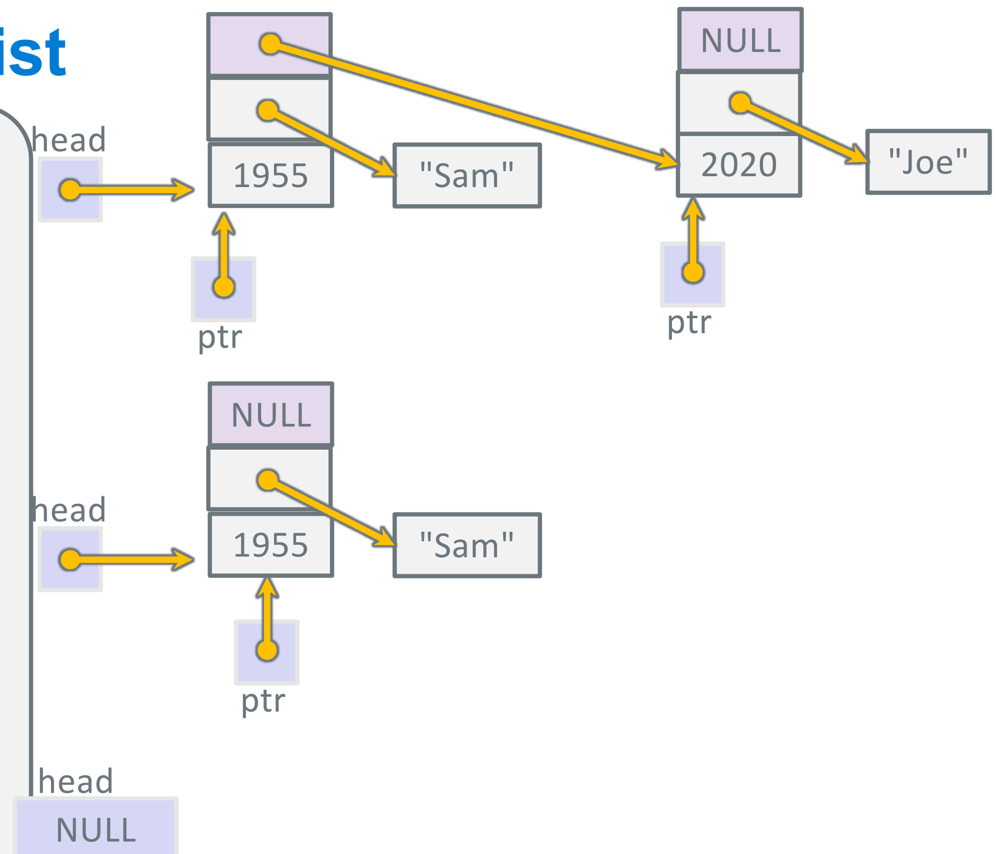
Returns pointer if found
NULL otherwise

```c
struct node *found;

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


x

Deleting a Node in a Linked List

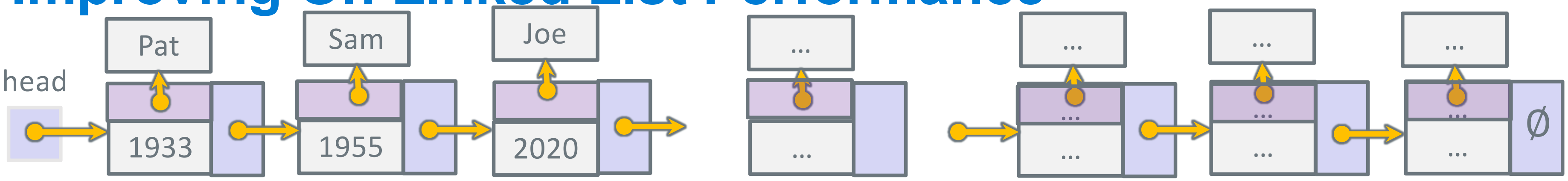
```c
// returns head pointer; may have changed…
struct node *deleteNode(int name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL;

    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;
}
```




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

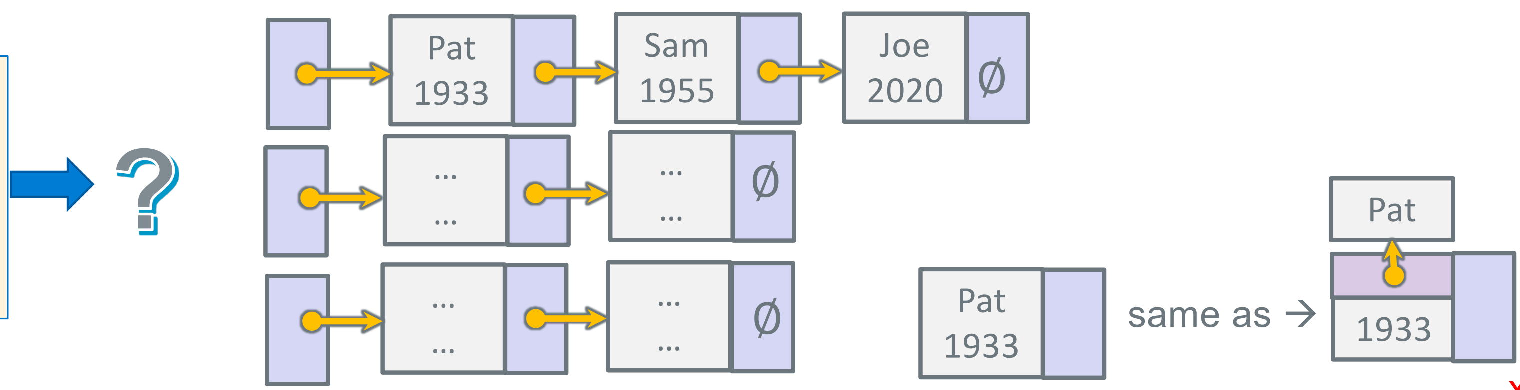

x

Improving On Linked List Performance



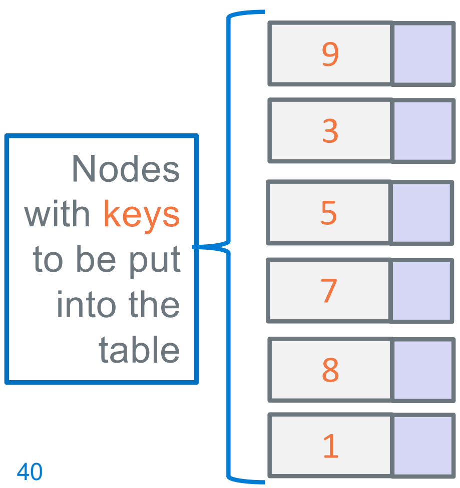
- When linked lists get long, the cost of finding an entry continues to increase O(n)
- How to improve search time?
- Break the single linked list into multiple **shorter length** linked lists
  - Shorter lists are faster to search
- **Requires a function** that takes a lookup key and selects just one of the shortened lists

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

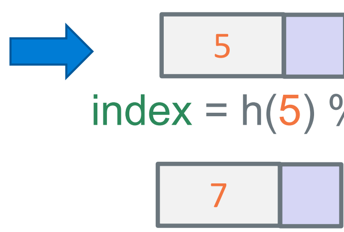


Hashing

- 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



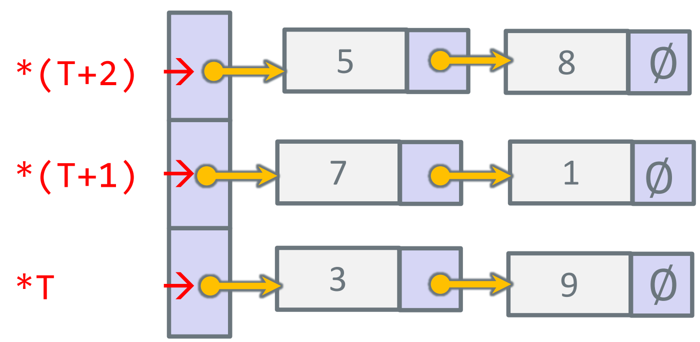
Using a Simple hash
h(k) = k // this is has value
index = k % size // map to a chain



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

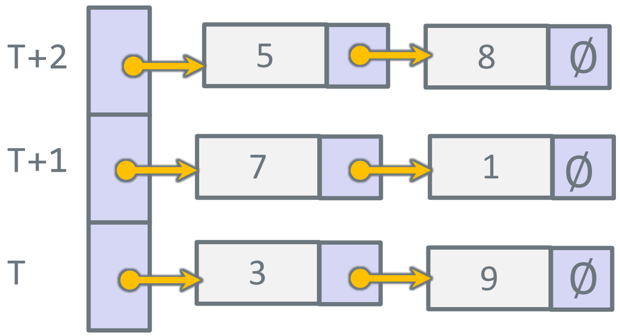
index = h(7) % 3 = 1 -> T+1

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



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**)*

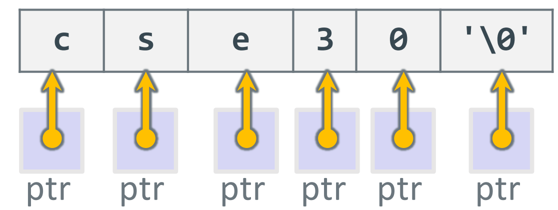
- **Hash arrays** need an **index number to select a chain**, so if we have a string, we must first convert to a number

x

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

```
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)

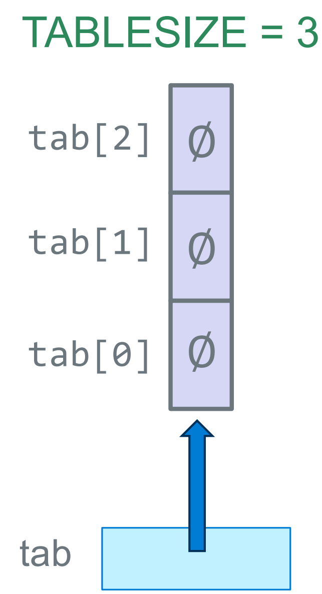


- Many different algorithms for string hash function (Dan Berman's djb2 above)
  - << is the **left** bit shift operator (later in course)
- **Fast to compute**, has a reasonable key distribution for short 8-bit ASCII strings into 32-bit unsigned ints

x

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

```
#define TBSZ 3
int main(void)
{
    struct node *ptr;
    struct node **tab; // pointer to hashtable
    uint32_t index;

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

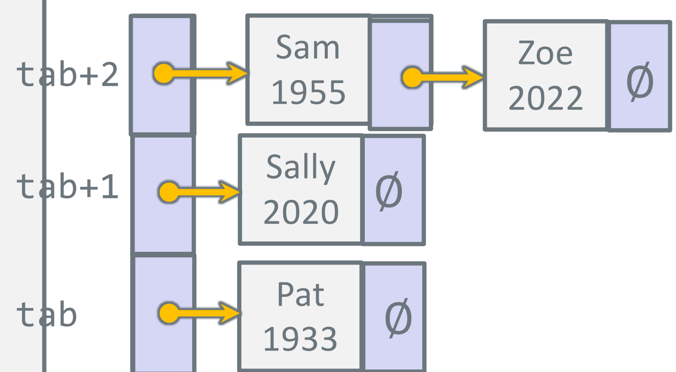


x

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;
```

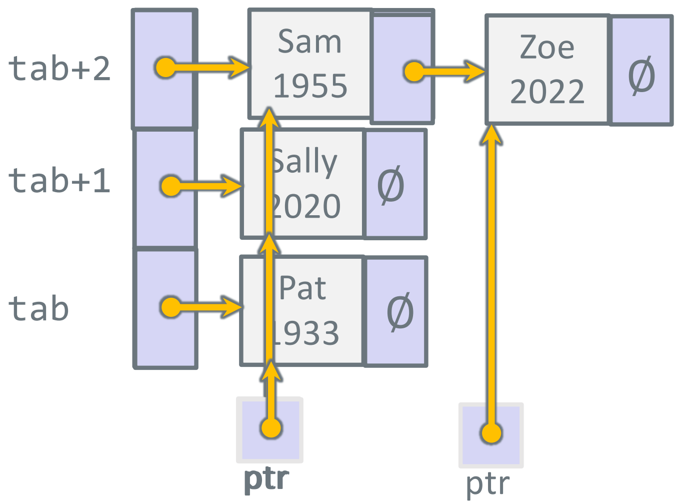
```
struct node {
  int year;
  char *name;
  struct node *next;
};
```



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

x

"Dumping" the Hash Table (traversing all Nodes)

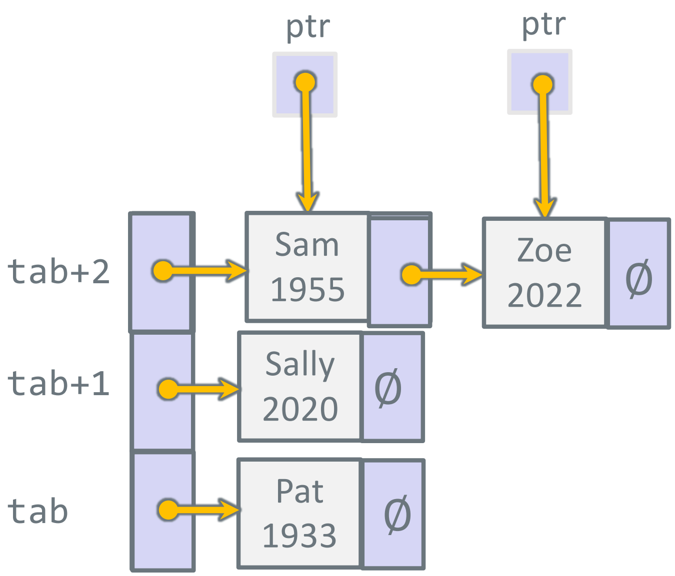



```
Dumping All Data
chain: 0
Year: 1933 Name: Pat
chain: 1
Year: 2020 Name: Sally
chain: 2
Year: 1955 Name: Sam
Year: 2022 Name: Zoe
```

```
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




```
// 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;
}
```

```
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

Extra Slides

-

Sizing Struct Members

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

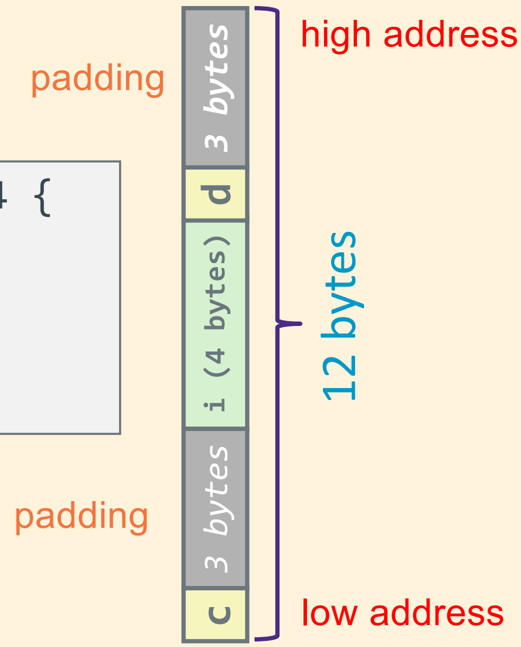
```
struct S4 {
  char c;     // byte aligned
  int i;      // 4 byte aligned
  char d;     // byte aligned
};
```

- 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

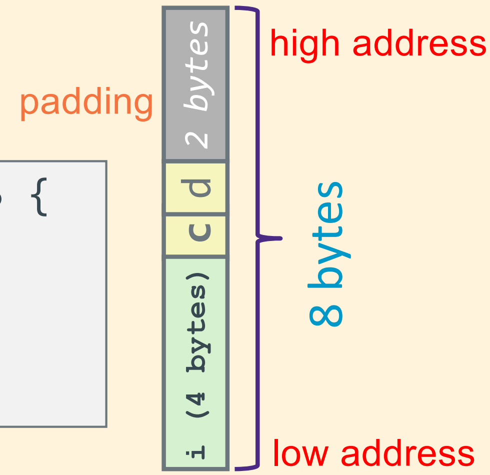
Re-Sizing Struct Members

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



Reduce size by being aware of member sizes

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



- 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