Version 1.00

# UCSD CSE 30

## Computer Organization and Systems Programming

## C Programming Part 4

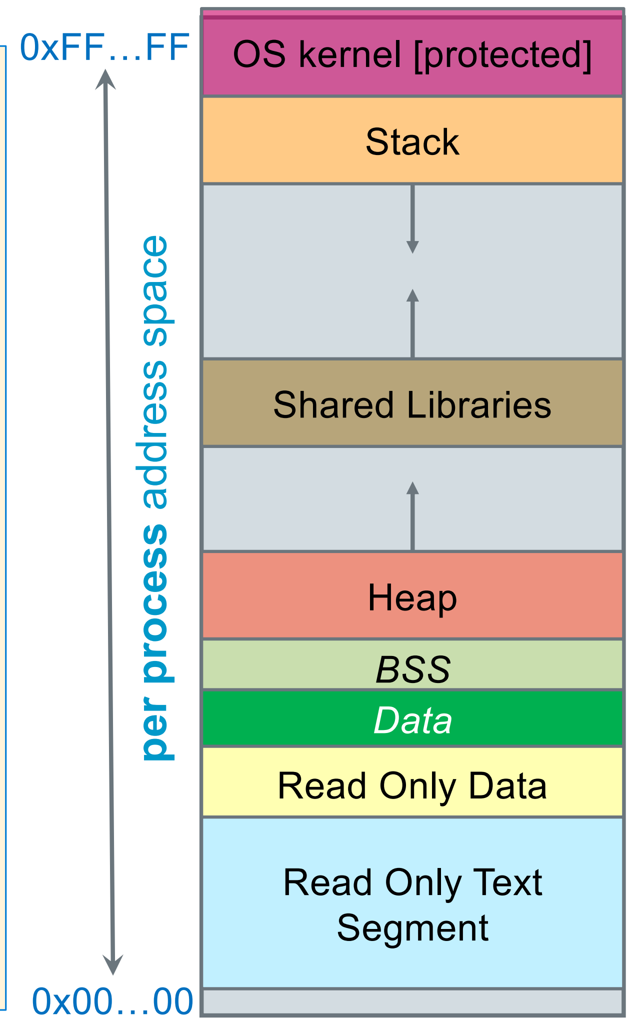
## Lecture 10 – Oct 24, 2022

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






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



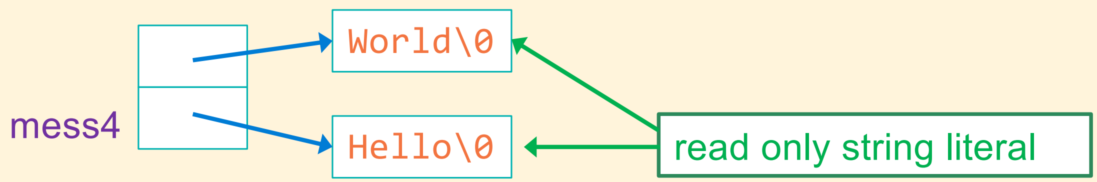
x

# String Literals, Mutable and Immutable arrays - 2

- mess4 is an array of pointers to immutable arrays

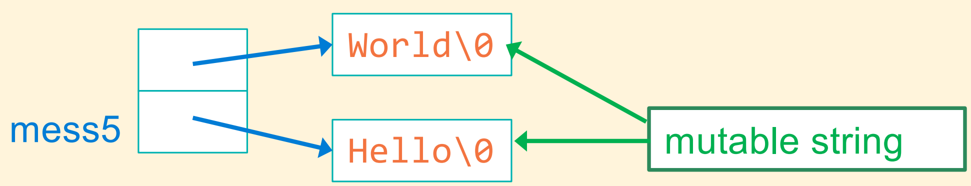
```c
char *mess4[] = {"Hello","World"};  // immutable string
*(*mess4 + 1)  = '\0';              // bus error
```

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




- mess5 is an array of pointers to mutable arrays

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

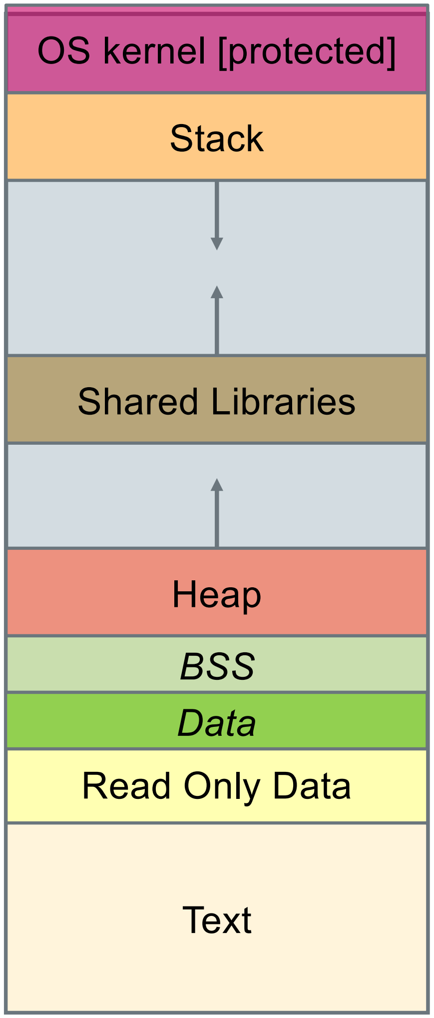





x

# 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 |
|---|---|---|
| `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 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

```
#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 passes the pointer in bytes to anything */
    return;
}
```

- Your program is responsible for cleaning up any memory it allocates but no longer needs
  - 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
  - Make sure you free memory when you no longer need it
- 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     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



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

```c
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

```c
char *str = strdup("Hello, world!");
str[0] = '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



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" struct type includes the keyword struct and the tagname

Easy to forget semicolon!

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

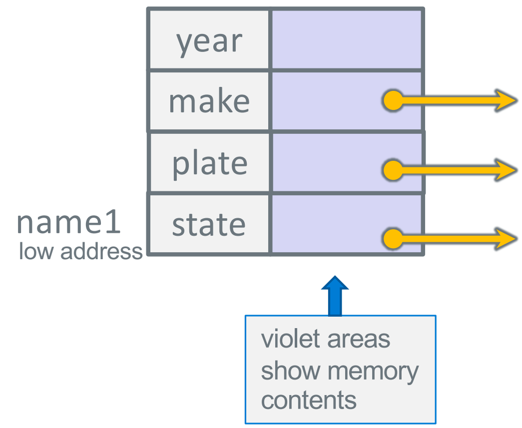
```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
};
```



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 which *"selects"* the requested field or member

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

| | |
|---|---|
| day | |
| month | |

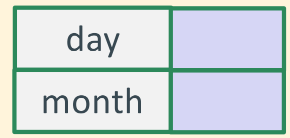
```
struct date bday; // struct instance
bday.month = 1;
bday.day = 24;

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

struct date bday

| | |
|---|---|
| day | 24 |
| month | 1 |

# Accessing members of a struct with pointers

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




- 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* |
|---|---|---|
| ()<br>[]<br>.<br>-><br>++ -- | Parentheses or function call<br>Brackets or array subscript<br>Dot or Member selection operator<br>Arrow operator<br>Postfix increment/decrement | left to right |
| ++ --<br>+ -<br>! ~<br>**(type)**<br>*<br>&<br>**sizeof** | Prefix increment/decrement<br>Unary plus and minus<br>not operator and bitwise complement<br>type cast<br>Indirection or dereference operator<br>Address of operator<br>Determine size in bytes | right to left |



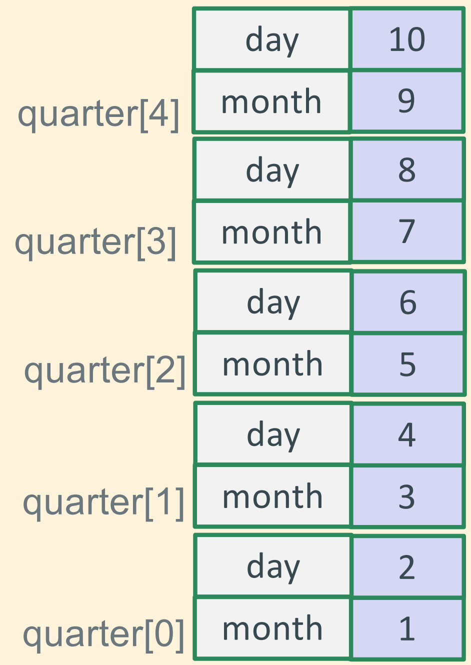
X

# 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(arry)/sizeof(*arry);  // cnt = 5
```

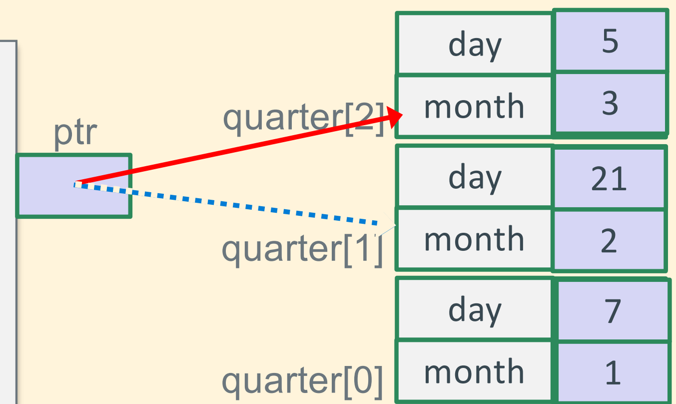





x

# 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;
```




| Operator | Description | Associativity |
| --- | --- | --- |
| ()<br>[]<br>.<br>-><br>++ -- | Parentheses or function call<br>Brackets or array subscript<br>Dot or Member selection operator<br>Arrow operator<br>Postfix increment/decrement | left to right |
| ++ --<br>+ -<br>! ~<br>(type)<br>*<br>&<br>sizeof | Prefix increment/decrement<br>Unary plus and minus<br>not operator and bitwise complement<br>type cast<br>Indirection or dereference operator<br>Address of operator<br>Determine size in bytes | right to left |

# 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;
```

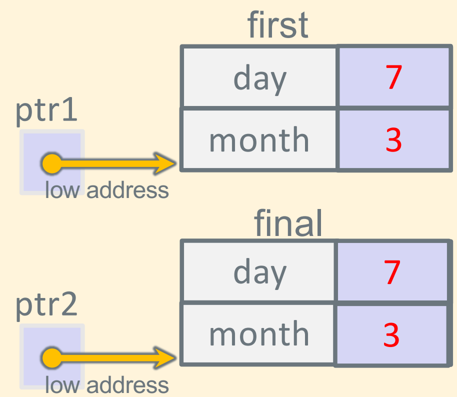


X

# 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
```






x

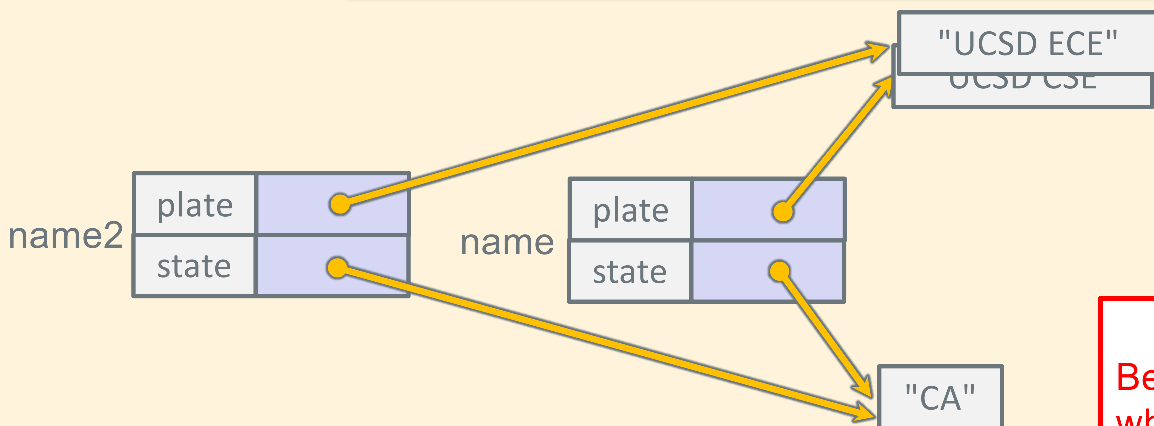
# Struct: Copy and Member Pointers

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

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


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

```c
name2 = name;   // copies members Only
```




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

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

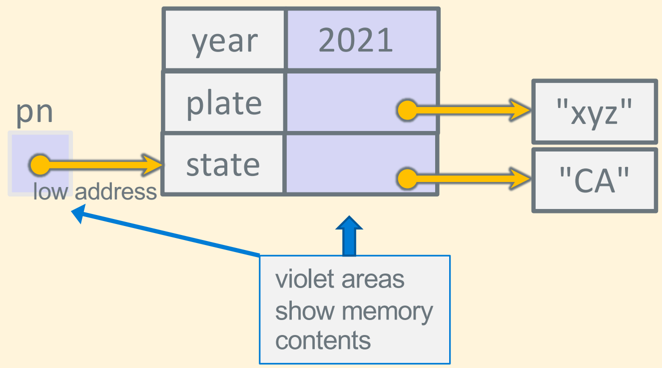


x

# 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)

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




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



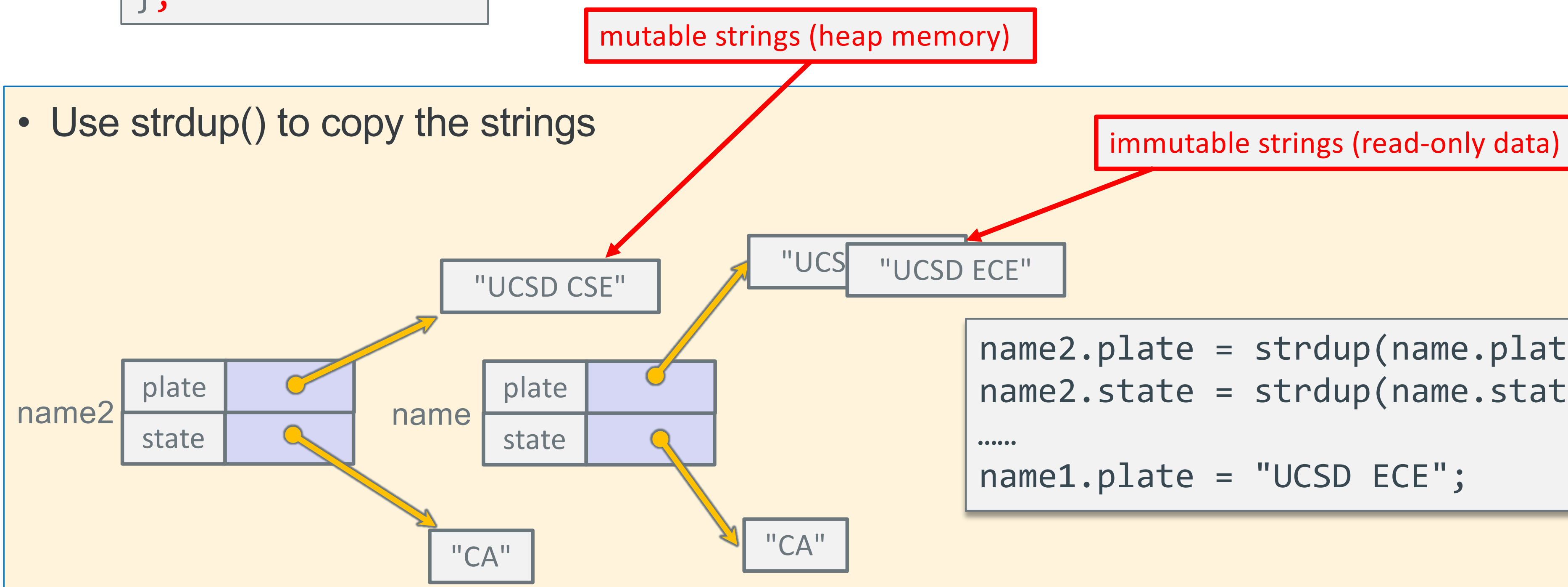
x

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

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

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

- Use strdup() to copy the strings




```c
name2.plate = strdup(name.plate);
name2.state = strdup(name.state);
......
name1.plate = "UCSD ECE";
```

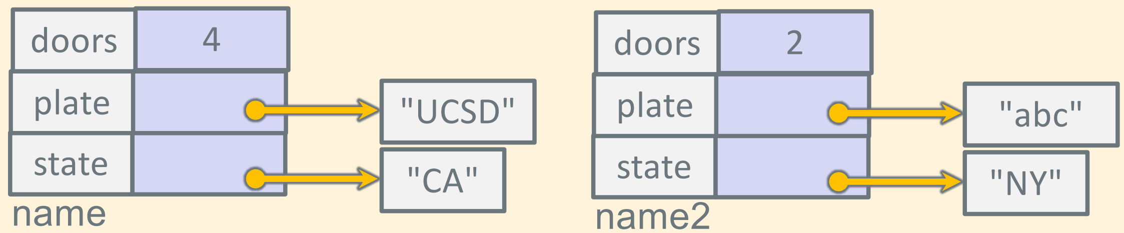


x

# 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");
}
```



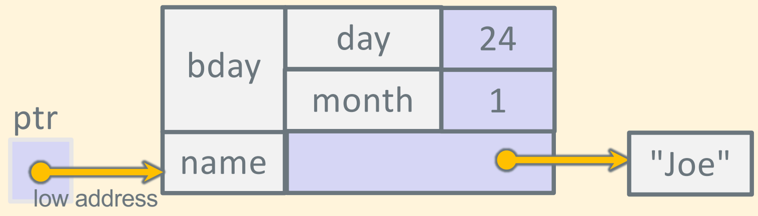
x

# 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;

ptr->bday.month = 1;
ptr->bday.day = 24;
```



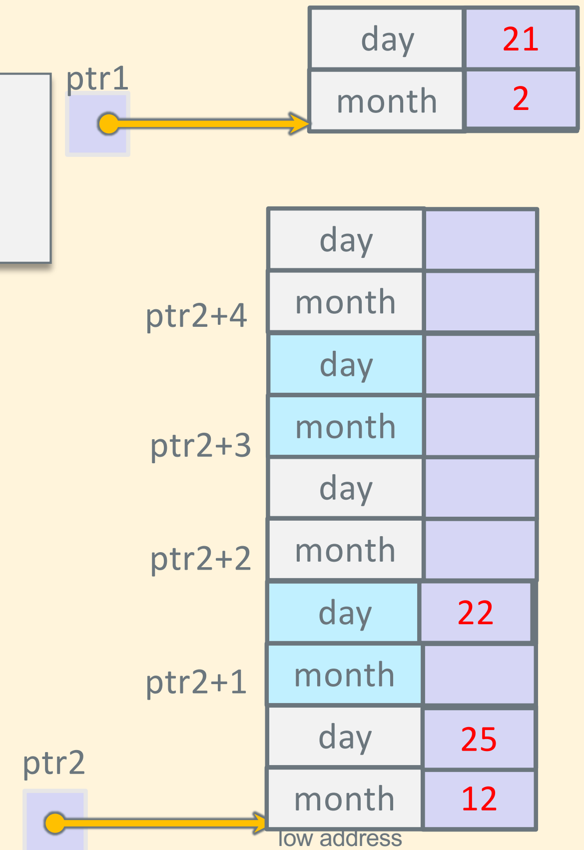
x

# 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;
```






x

# Struct As A Parameter to Functions

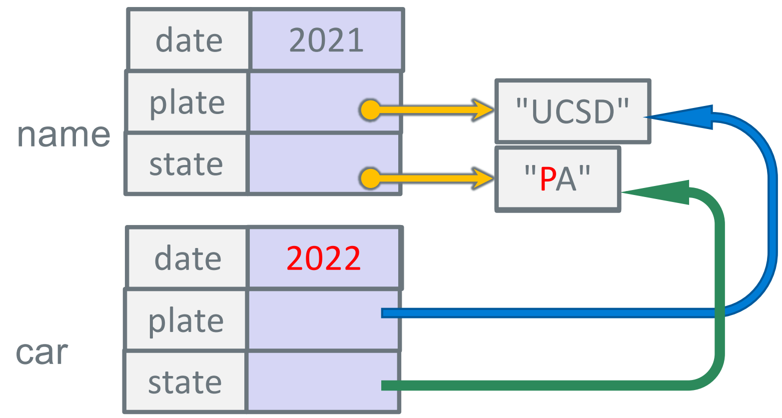
- WARNING: When you pass a struct, you pass a copy of the entire struct
  - this can be very expensive at runtime!
- More often code will pass the pointer to a struct to avoid the copy costs
  - Be careful and not modify what the pointer points to (unless it is an output parameter)
- Tradeoffs:
  - Passing 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…



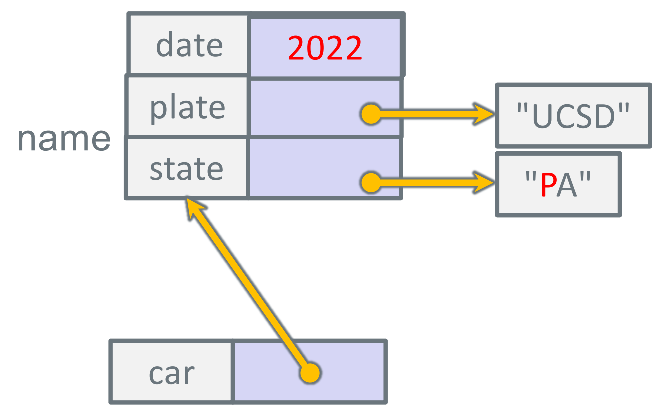
x

# Struct as a Parameter to Functions – Be Careful….

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




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



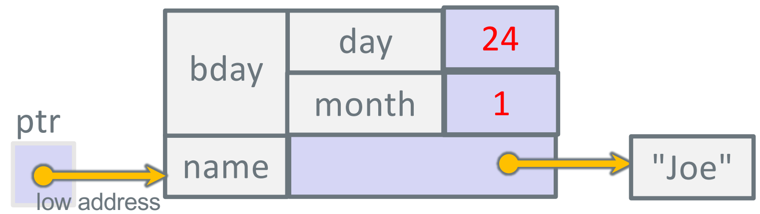



x

# Struct as an Output Parameter

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

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




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



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
};
```

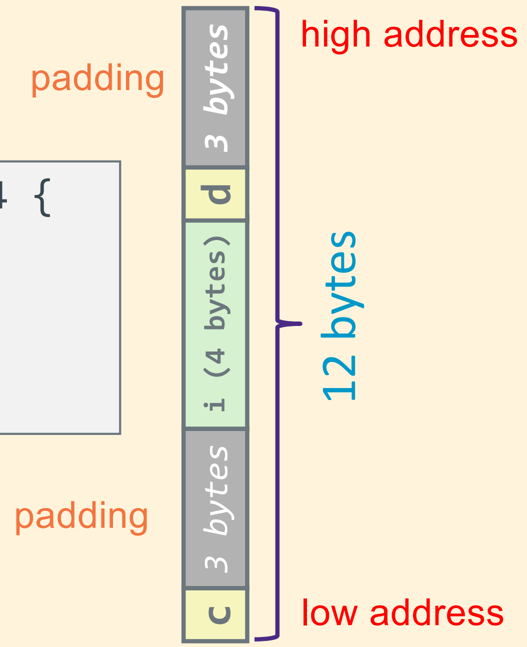
- Structs uses contiguously-allocated region of memory,
- compilers **are required to follow member order**, and HW alignment requirements
  1. not allowed to re-arrange member order in memory
  2. struct starting address: aligned to the requirements of largest member
  3. Add memory space between members (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

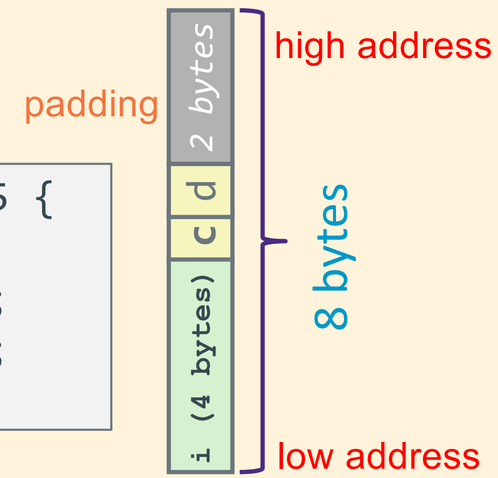
# Re-Sizing Struct Members

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




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

# 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



x

# 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

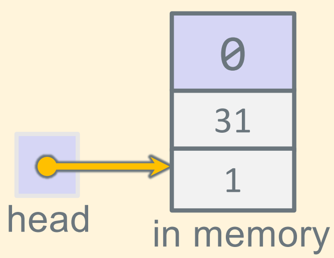
- A **self-referential struct** is a struct that has one or more members that are **pointers** to a **struct variable of the same type**
- Self-referential member
  - points to same type – itself

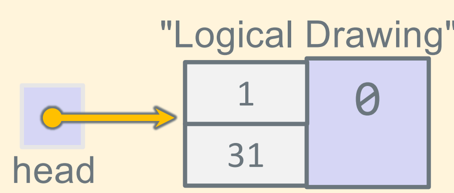
```
struct node {
  int data;
  struct node *next;
};
```


- There can be multiple struct members that make up the payload

```
struct node {
  int month;
  int day;
  struct node *next;
} x;
x.month = 1;
x.day = 31;
x.next = NULL;
```

```
struct node *head;   // head pointer
head = &x;
```





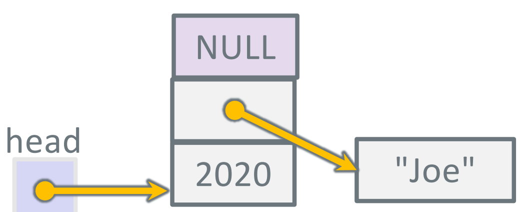

x

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

```
// create node; insert at front when passed head
struct node *
creatNode(int date, 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->date = date;
        ptr->next = link;
    }
    return ptr;
}
```

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

```
// 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;
}
```

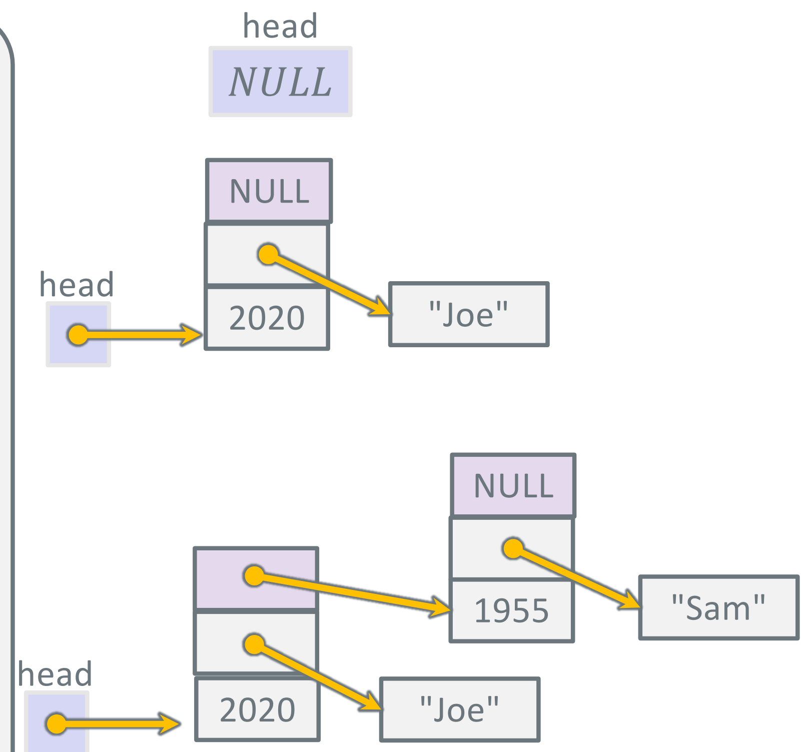




x

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

```
struct node *
insertEnd(int year, char *name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = head;
    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;
```

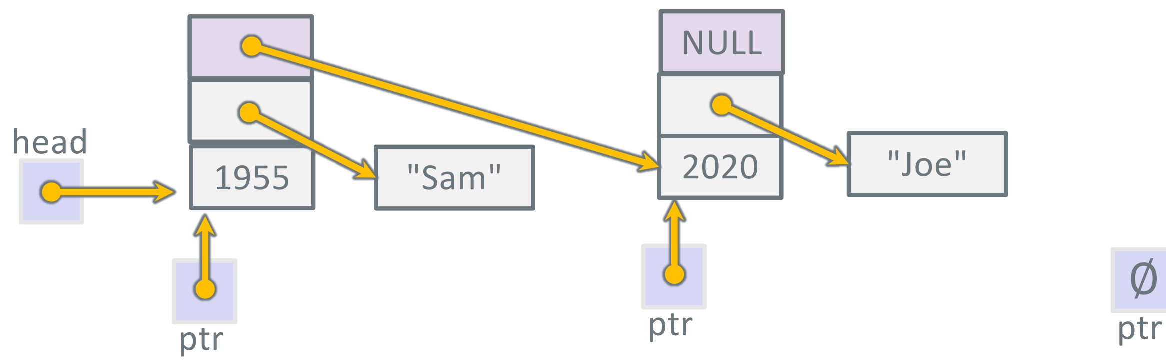


X

# "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->data1, ptr->data2);
    ptr = ptr->next;
}
```

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



x

# Extra Slides