Version 2.18

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 10

Keith Muller

Vax 11/780 1980

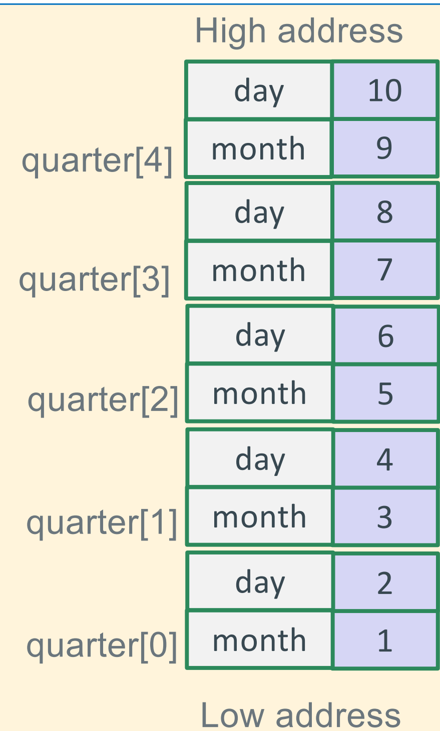
Thursday office hours moved to
Section B ZOOM Midterm Review
Thursday 7:30 PM – 8:30 PM
(My office hours zoom #)
(see canvas for number)

Accessing members of a struct

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

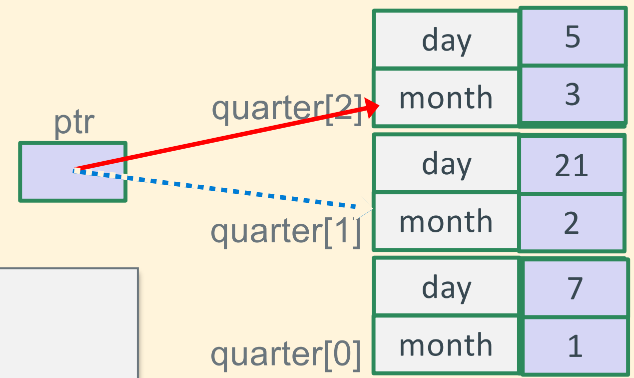
- You can create an array of structs and initialize them

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

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



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;
```


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;
```

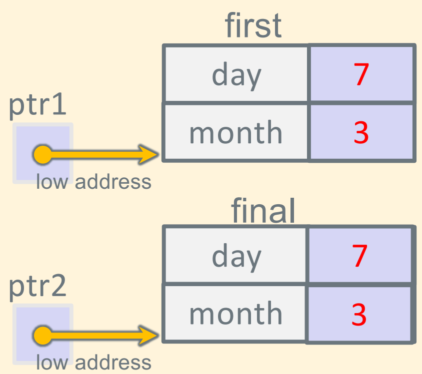

X

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

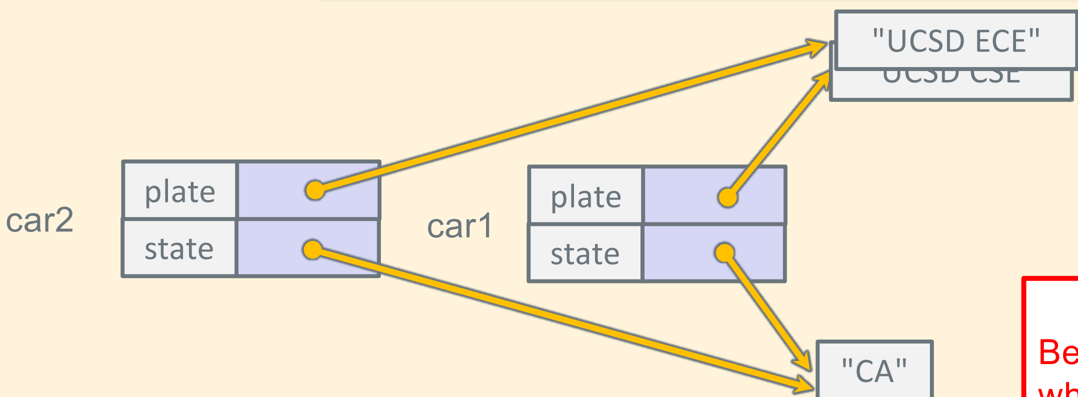
Caution: Assignment is a Shallow Copy of struct members

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

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

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

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




```
car2.plate = "UCSD ECE";
```

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

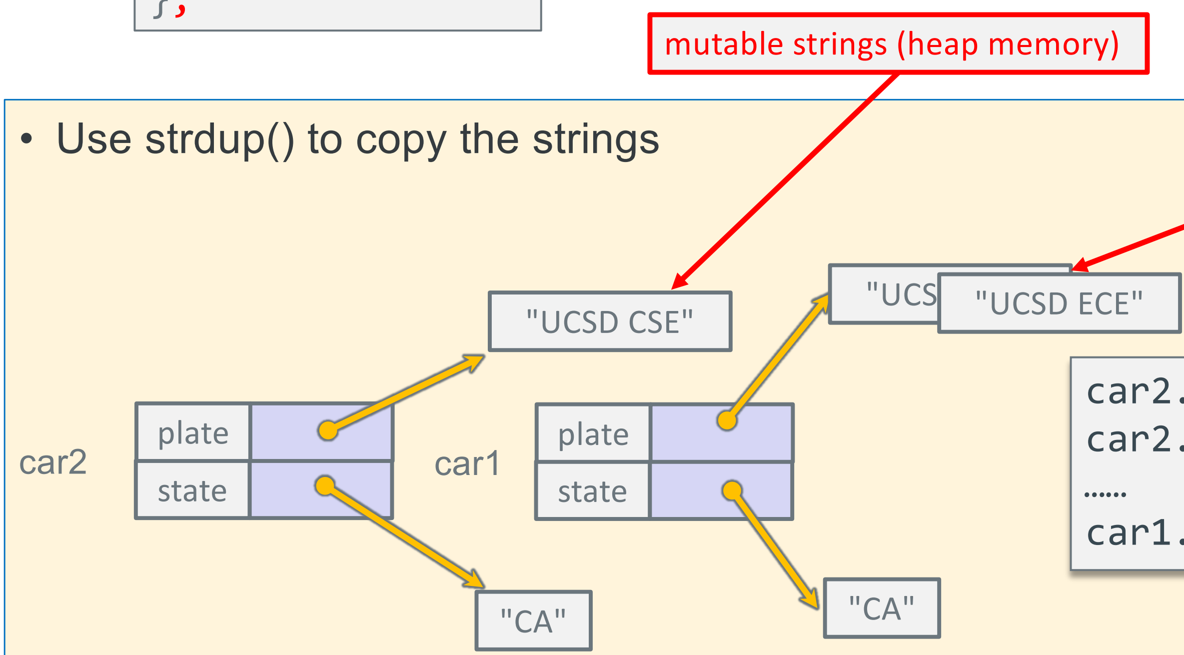
x

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



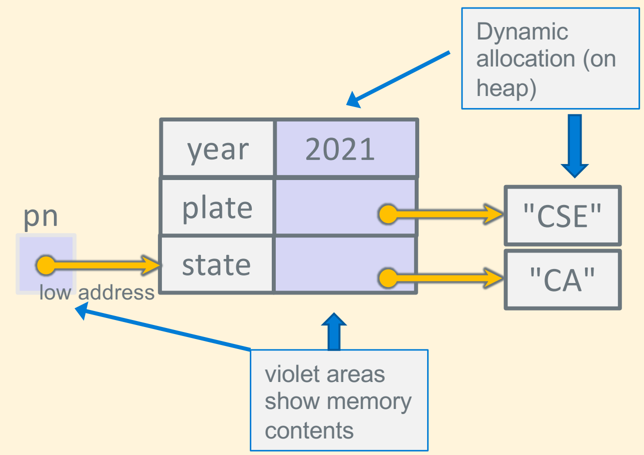

```c
car2.plate = strdup(car1.plate);
car2.state = strdup(car1.state);
……
car1.plate = "UCSD ECE";
```


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;
```

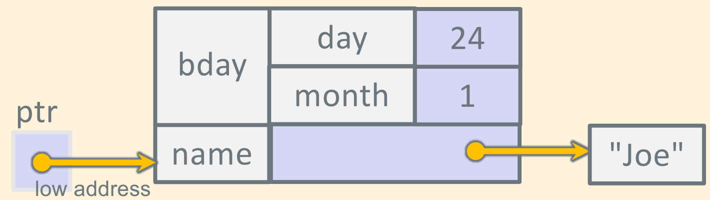

x

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;
};
```



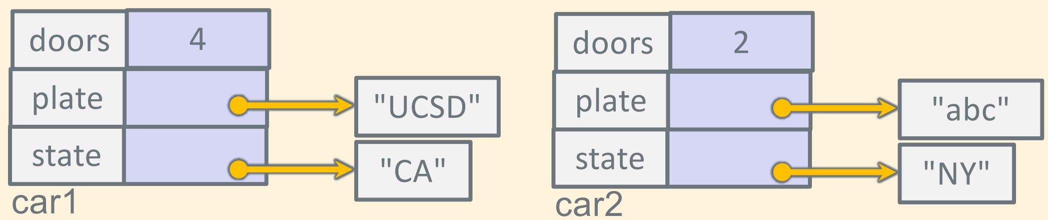

```
struct person first;
struct person *ptr;
ptr = &first;

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

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

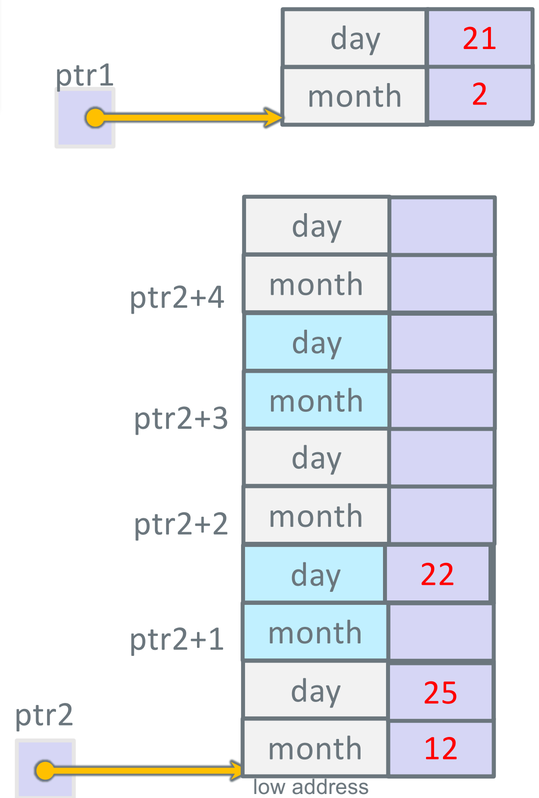



```
struct vehicle car1  = {"CA", "UCSD", 4};
struct vehicle car2 = { (char []) {"NY"}, (char []) {"abc"}, 2};
```

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

Struct Arrays: Dynamic Allocation

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




```
(*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!

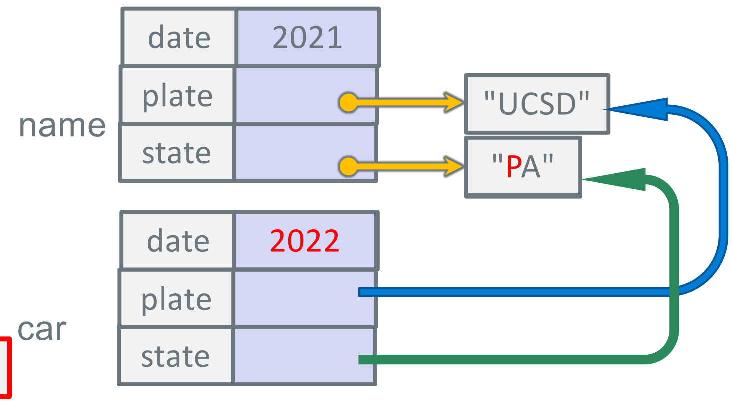
- WARNING: When you pass a struct, **you pass a copy** of all the struct members
  - This is a shallow copy (shallow copy – so if you have members that are pointers watch out)
- More often code will pass the pointer to a struct to avoid the copy costs
  - Be careful and not modify what the pointer points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

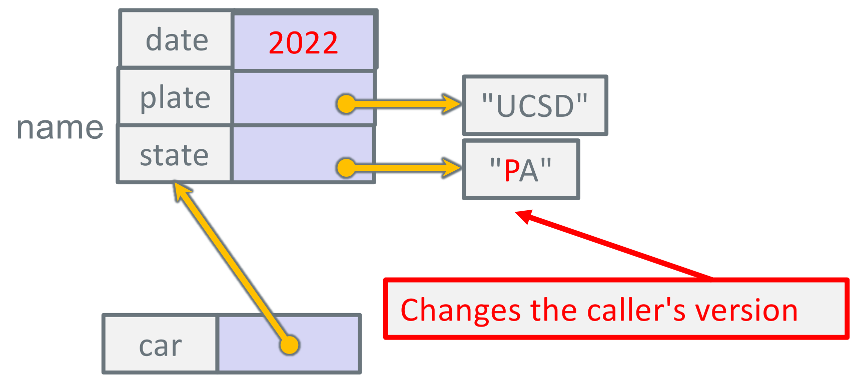
Struct Function Parameters – Be Careful it is not like arrays

```
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);
```

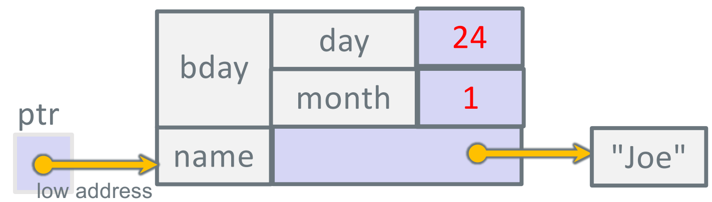


x

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

x

Review: Singly Linked Linked List - 2

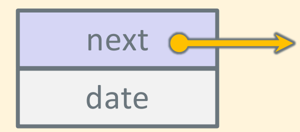
- Memory for each node is typically allocated dynamically at execution time (*i.e.,* using *heap memory – malloc() etc.*) when a new node is added to the list
- Memory for each node may be freed at execution time, using free() when a node is removed from the list
- Unlike arrays, linked list nodes are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

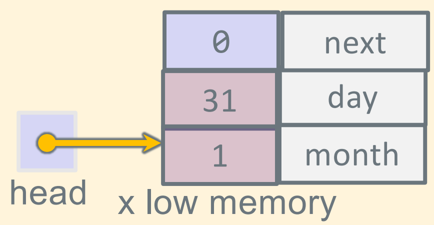
```
struct node {
  int date;
  struct node *next;
};
```



- 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

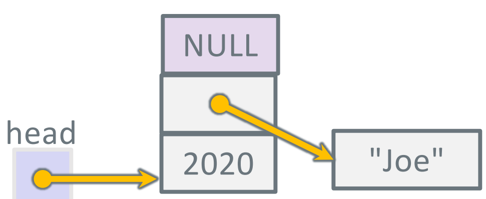
Creating a Node & Inserting it at the **Front** of the List

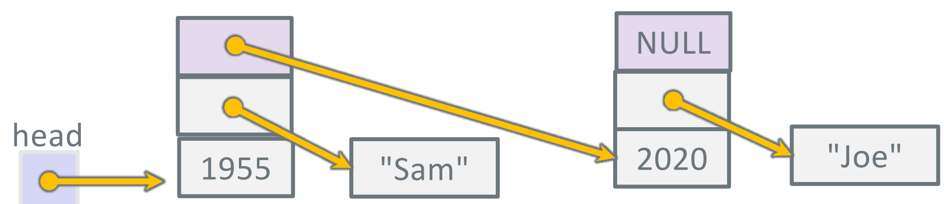
```
// create node; insert at front when passed head
struct node *
creatNode(int year, char *name, struct node *link)
{
    struct node *ptr = malloc(sizeof(*ptr));
    if (ptr != NULL) {
        if ((ptr->name = strdup(name)) == NULL) {
            free(ptr);
            return NULL;
        }
    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

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

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

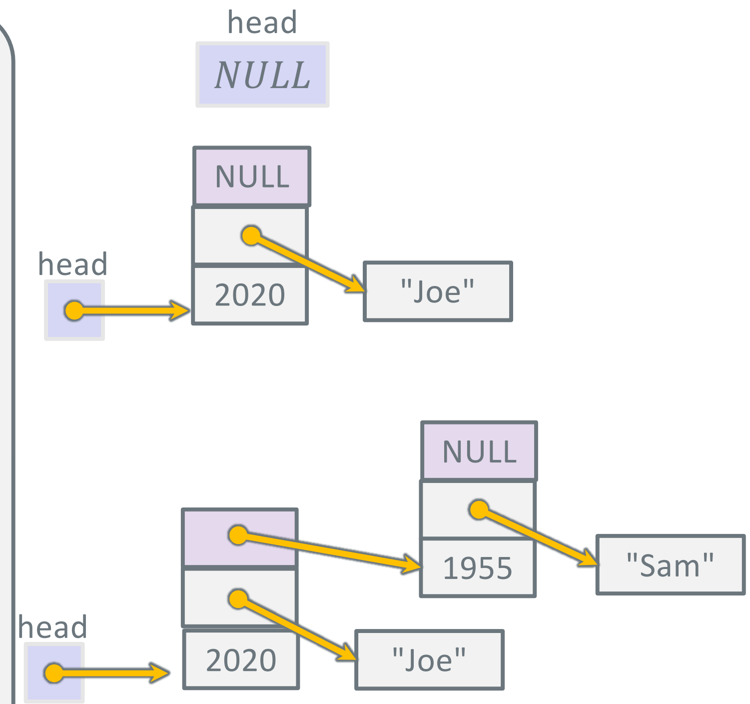
```
// calling function body
struct node *head = NULL; // insert at front
struct node *ptr;

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
}
```





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;
}
```



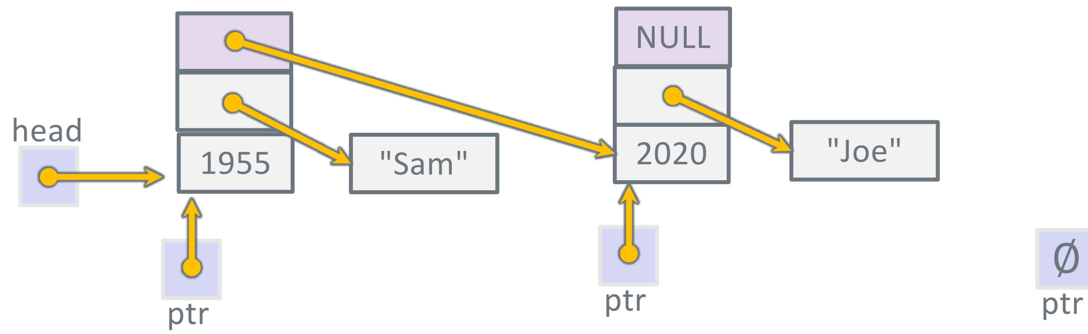

```
struct node *head = NULL;  // insert at end
struct node *ptr;
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)
    head = ptr;
```


x

"Dumping" the Linked List

"walk the list from head to tail"

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

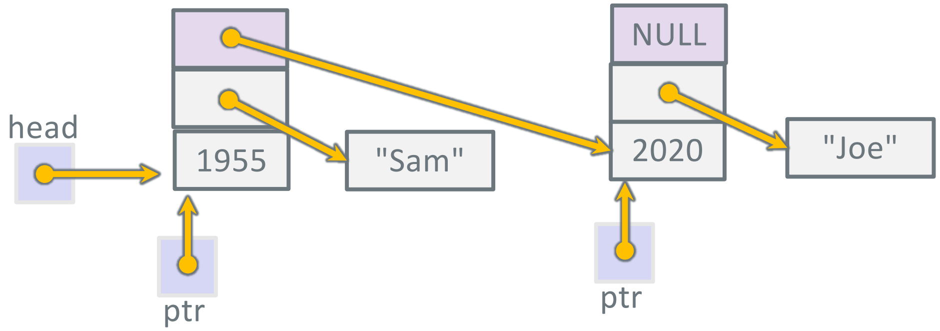



```
struct node *head;
struct node *ptr;
…
printf("\nDumping All Data\n");
ptr = head;
while (ptr != NULL) {
    printf("year: %d name: %s\n", ptr->year, ptr->name);
    ptr = ptr->next;
}
```

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


x

Finding A Node Containing a Specific Payload Value




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

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

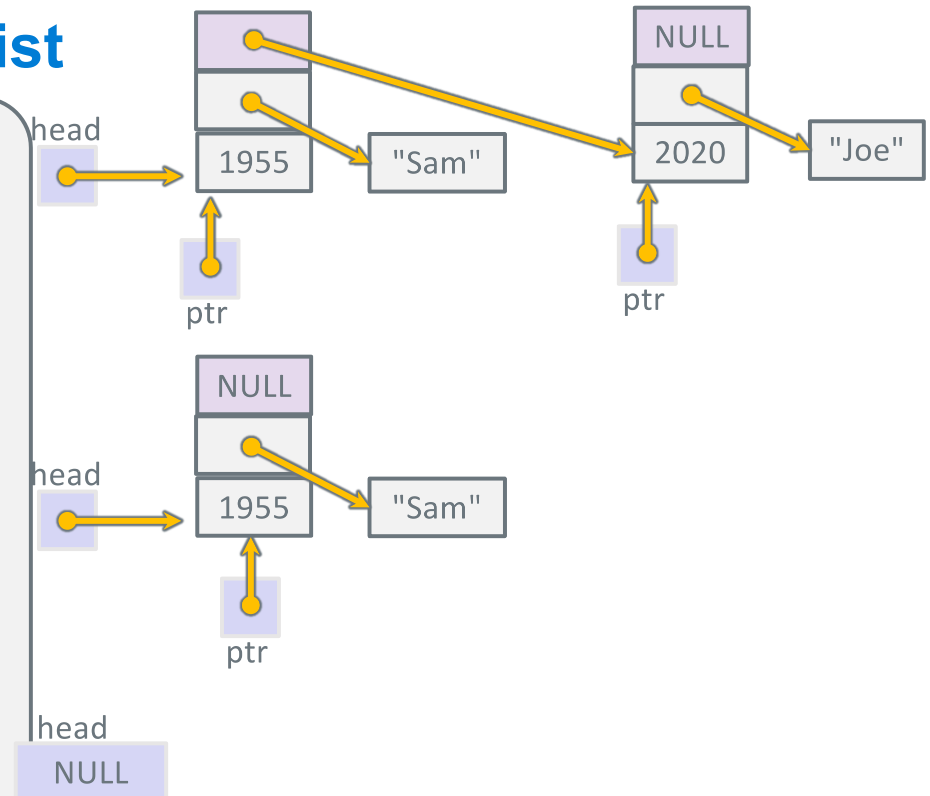
Returns pointer if found
NULL otherwise

```
struct node *found;

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


X

Deleting a Node in a Linked List

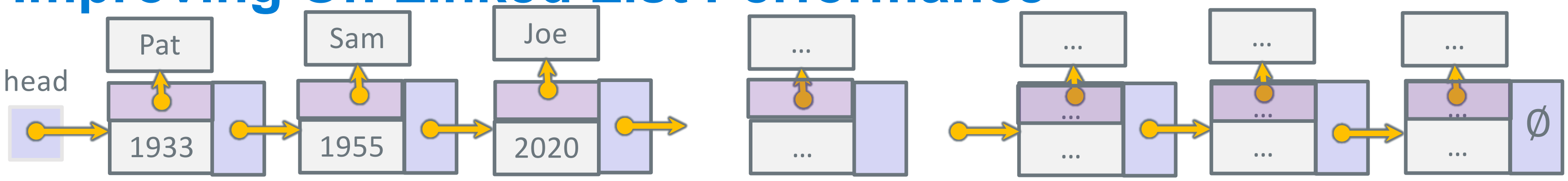
```
// returns head pointer; may have changed…
struct node *deleteNode(int name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL;

    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;
}
```




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

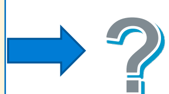
x

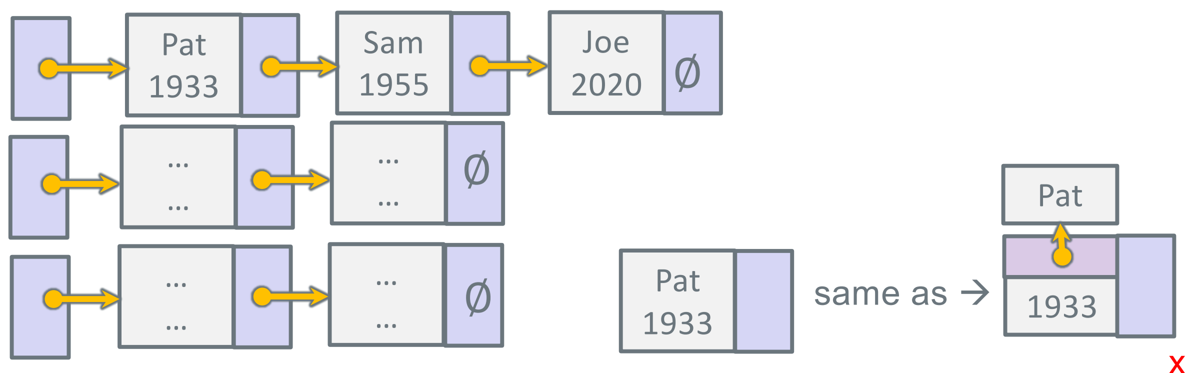
Improving On Linked List Performance



- 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

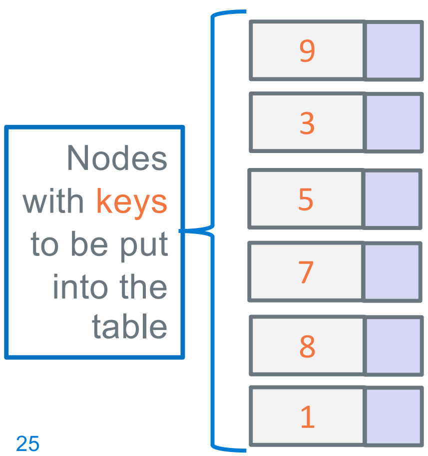
How do you determine on which linked list an entry is stored?



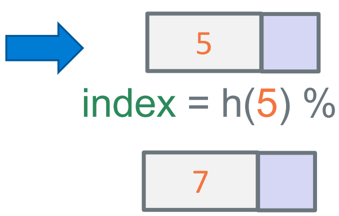


Hashing

- 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



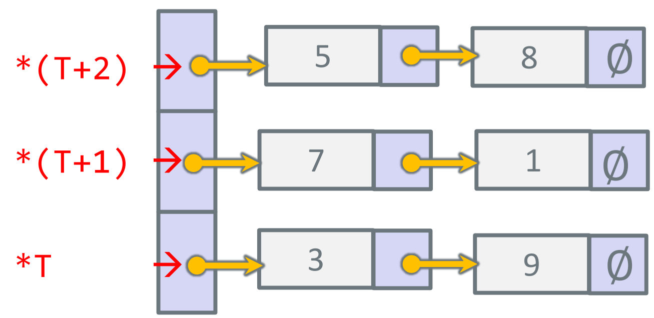
Using a Simple hash
h(k) = k // this is has value
index = k % size // map to a chain



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

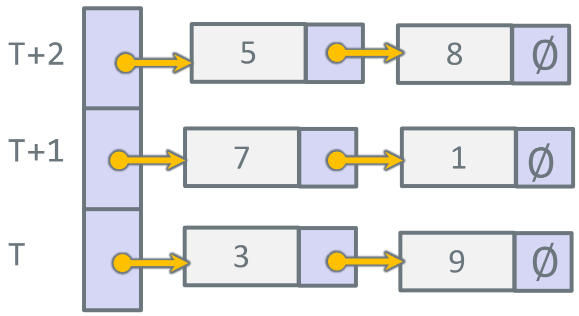
index = h(7) % 3 = 1 -> T+1

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**)*

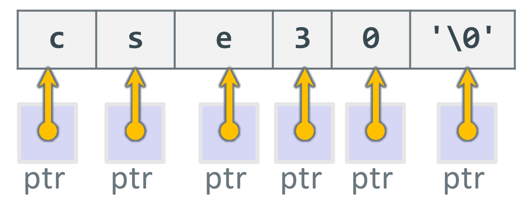
- **Hash arrays** need an **index number to select a chain**, so if we have a string, we must first convert to a number

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)

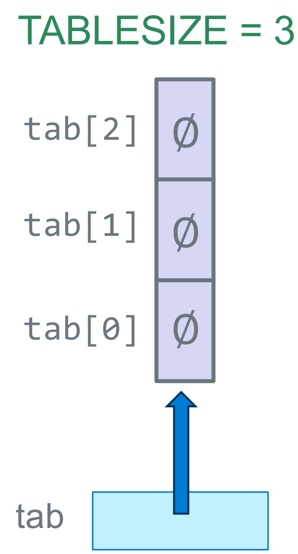


- Many different algorithms for string hash function (Dan Berman's djb2 above)
  - << is the **left** bit shift operator (later in course)
- **Fast to compute**, has a reasonable key distribution for short 8-bit ASCII strings into 32-bit unsigned ints

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
```

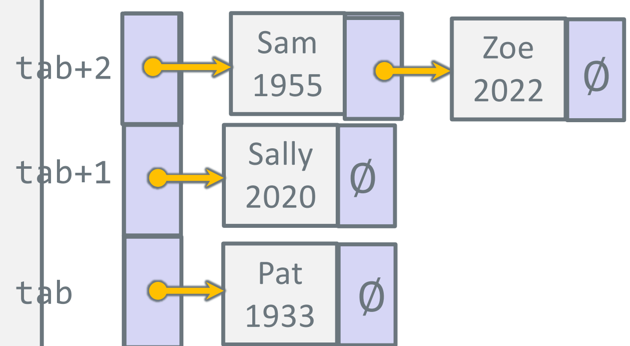


x

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;
};
```

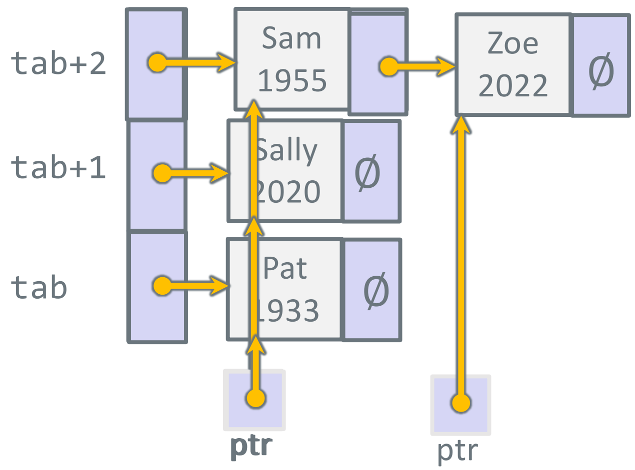


Notice

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)

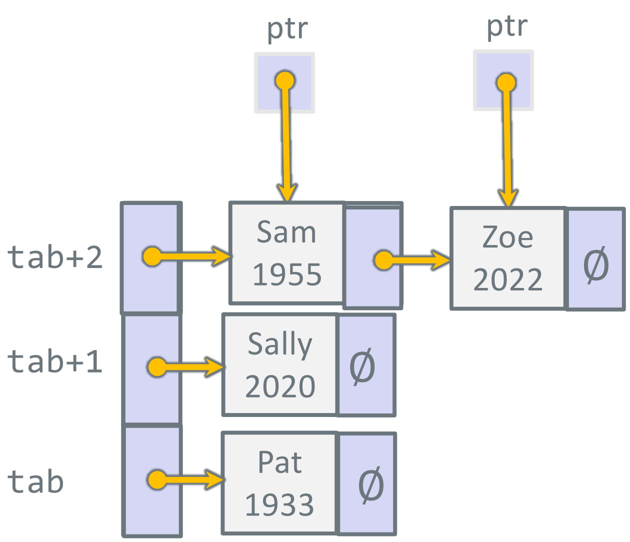



```
Dumping All Data
chain: 0
Year: 1933 Name: Pat
chain: 1
Year: 2020 Name: Sally
chain: 2
Year: 1955 Name: Sam
Year: 2022 Name: Zoe
```

```
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

Positive Number (unsigned) in 4 bits

- Real hardware has a fixed number of bits to store numbers (pi-cluster is 32 bits)
- There are only 2^n distinct values in n bits
- This limits the range of positive number to be 0 (unsigned min) to 2^n - 1 (unsigned max)

Hex digit	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111

umin

Hex digit	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
Decimal value	8	9	10	11	12	13	14	15
Binary value	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111

umax

x

Unsigned Integers (positive numbers) with Fixed # of Bits

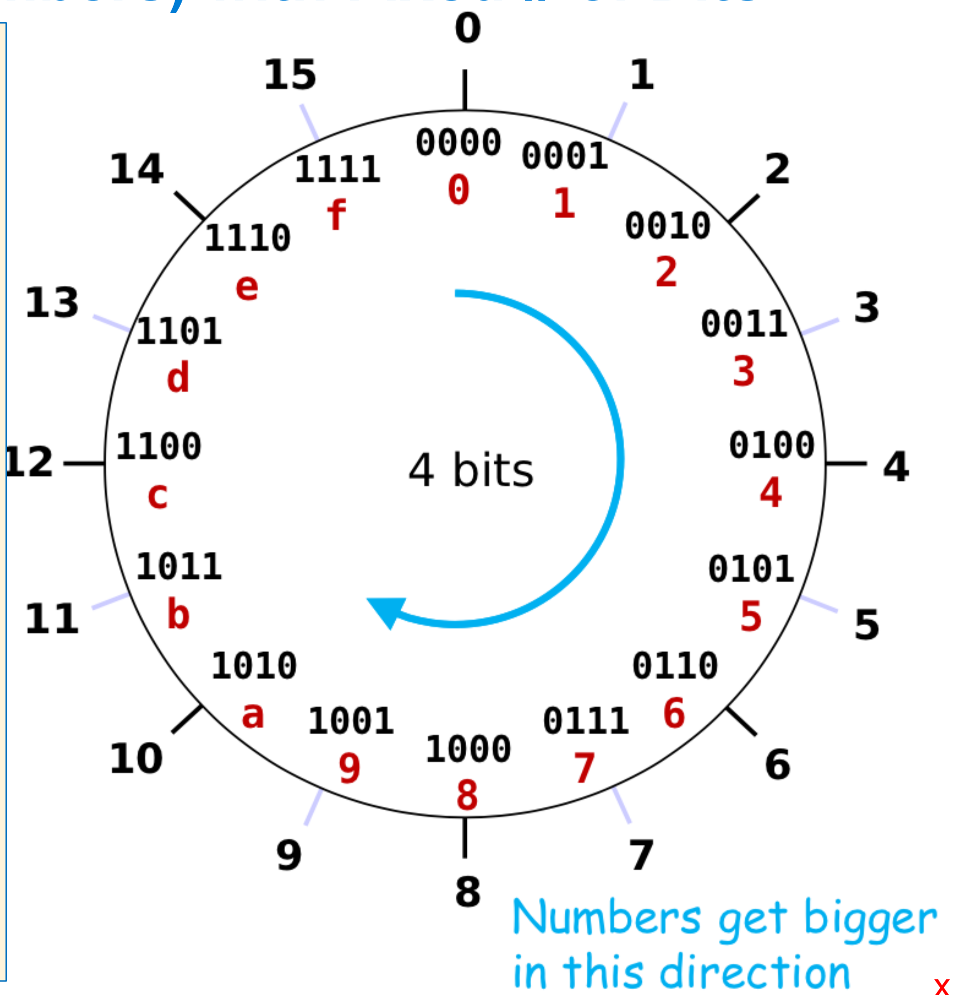
- 4 bits is 2^4 = ONLY 16 distinct values
- **Modular** (C operator: %) or clock math
  - Numbers start at 0 and "wrap around" after 15 and go back to 0
- Keep adding 1

  wraps (clockwise)

  0000 -> 0001 … -> 1111 -> 0000
- Keep subtracting 1

  wraps (counter-clockwise)

  1111 -> 1110 … -> 0000 -> 1111
- Addition and subtraction use normal "carry" and "borrow" rules, just operate in binary



x

Unsigned Binary Number: Addition in 4 bits

Be Aware in Binary

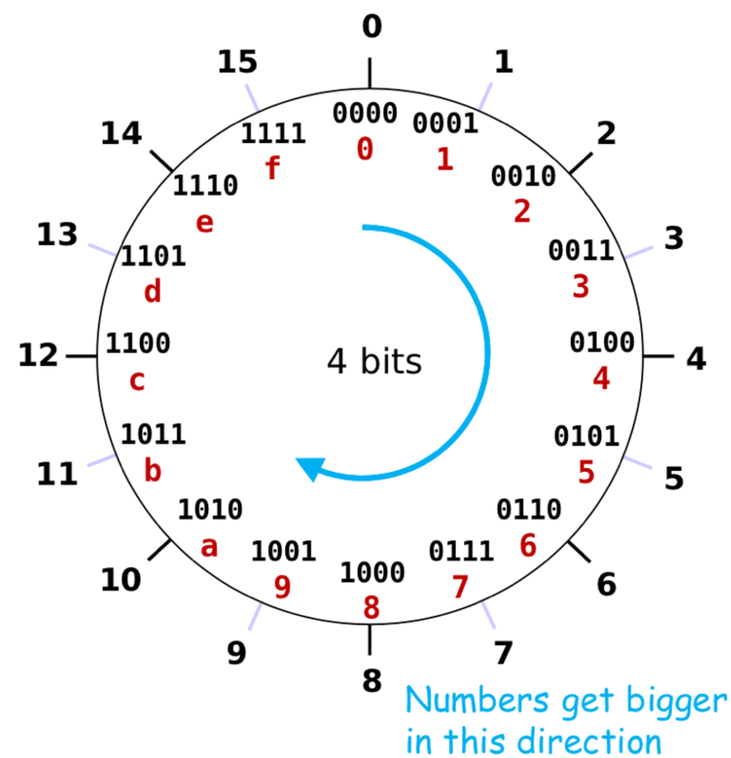
1 + 1 = 10 base 10: (1 + 1 = 2)

1 + 1 + 1 = 11 base10: (1 + 1 + 1 = 3)

Carry Bit

carries 0 1 1 1

carries	0	1	1	1		
		0	0	0	1	1
+		0	1	1	1	7
sum		1	0	0	0	= 8



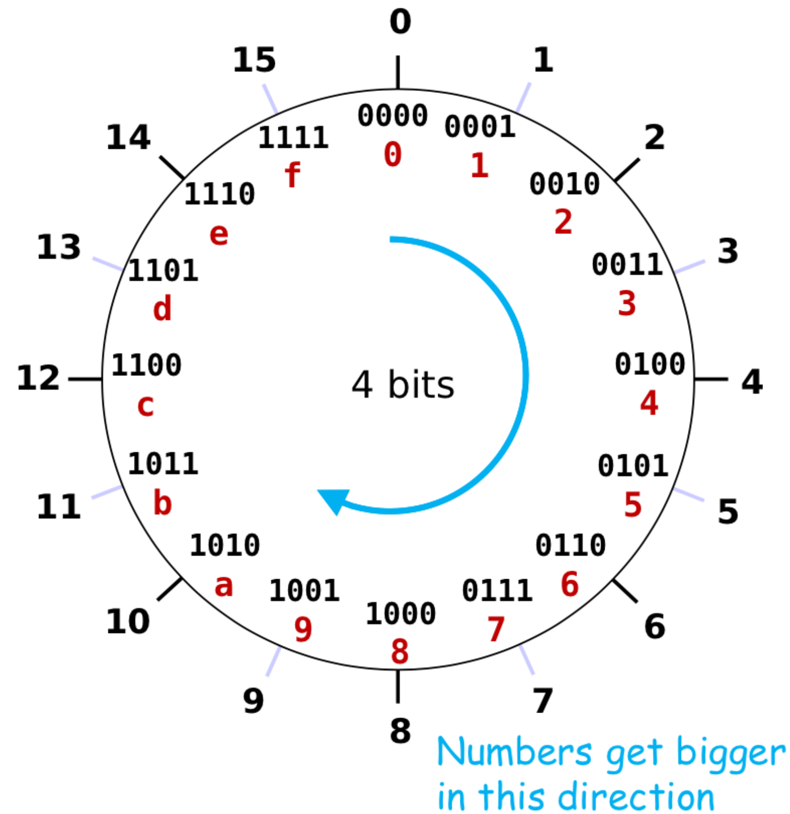
x

Unsigned Binary Number: Subtraction in 4 bits

Be Aware in Binary
1 - 1 = 0 base 10: (1 - 1 = 0)
10 - 1 = 1 base10: (2 – 1 = 1)

Borrows

			1			
	0	~~1~~	0	1		5
-	0	0	1	1		- 3
sum	0	0	1	0	=	2



x

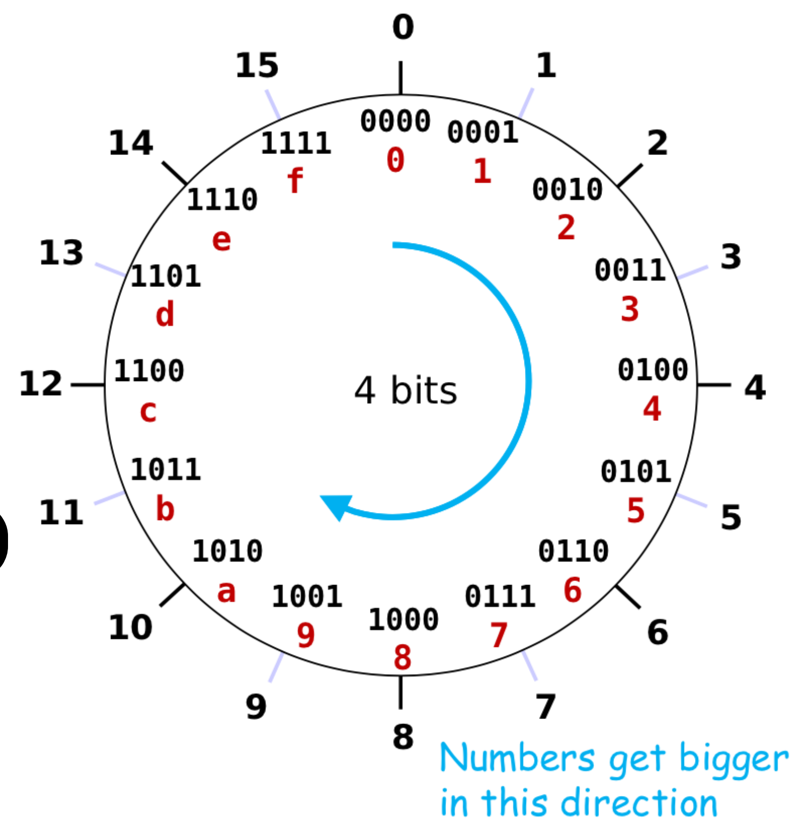
Unsigned Binary Number: Addition in 4 bits – Overflow!

Be Aware in Binary

1 + 1 = 10	base 10: (1 + 1 = 2)
1 + 1 + 1 = 11	base10: (1 + 1 + 1 = 3)

Carry Bit

carries	1	1	1			
		1	0	1	0	10
+		0	1	1	1	7
sum		0	0	0	1	≠ 17



x

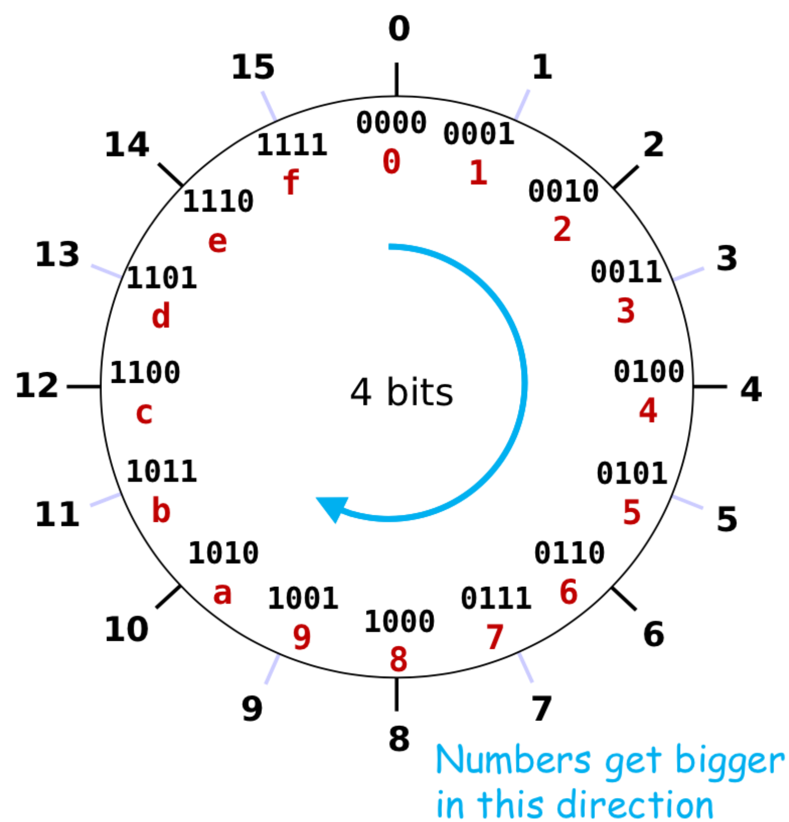
Unsigned Binary Number: Subtraction in 4 bits – Overflow!

Be Aware in Binary

1 - 1 = 0	base 10: (1 - 1 = 0)
10 - 1 = 1	base10: (2 – 1 = 1)

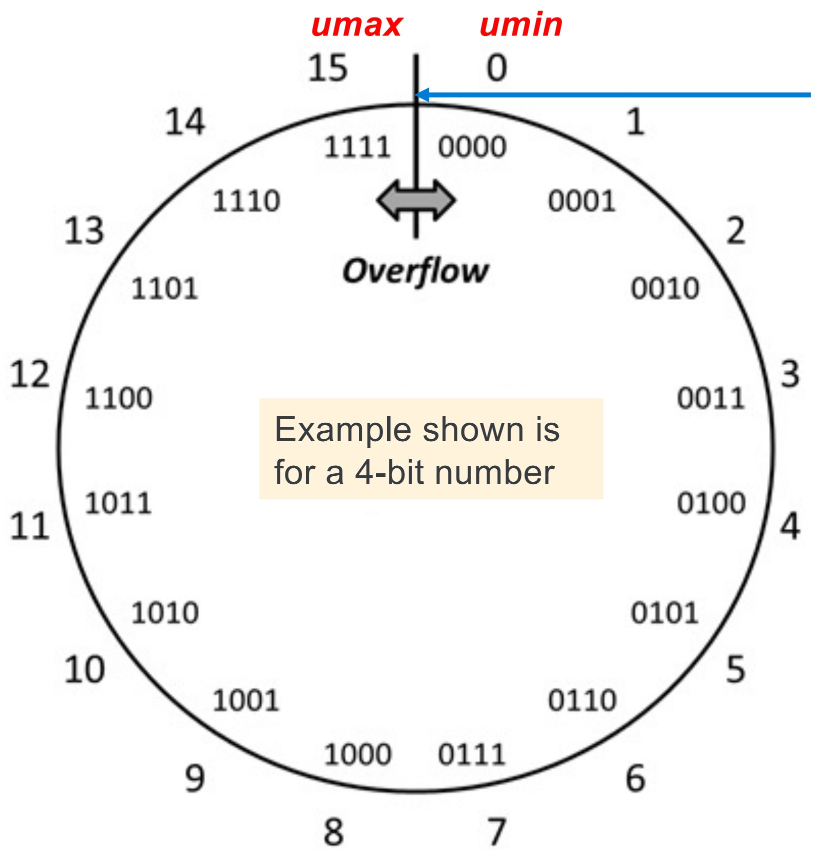
Borrows

$$
\begin{array}{rcccccl}
 & \cancel{1}\,1 & 10 & & 1 & & \\
 & \cancel{0} & \cancel{1} & 0 & 1 & & 5 \\
- & 0 & 1 & 1 & 1 & & 7 \\
\hline
\text{sum} & 1 & 1 & 1 & 0 & \neq & -2
\end{array}
$$



x

Overflow: Going Past the Boundary Between umax and umin



Overflow with unsigned numbers:
Occurs when an arithmetic result (from addition or subtraction for example) is is more than **min** or **max** limits

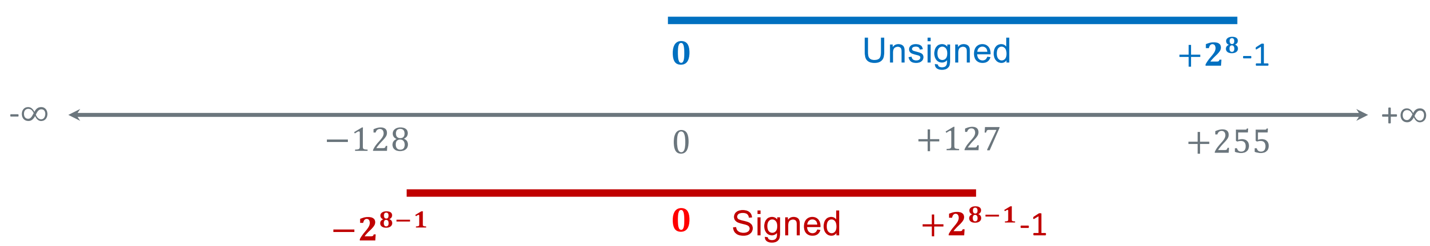
C (and Java) ignore overflow exceptions

- You end up with a bad value in your program and absolutely no warning or indication… happy debugging!….

x

Problem: How to Encode Both Positive and Negative Integers

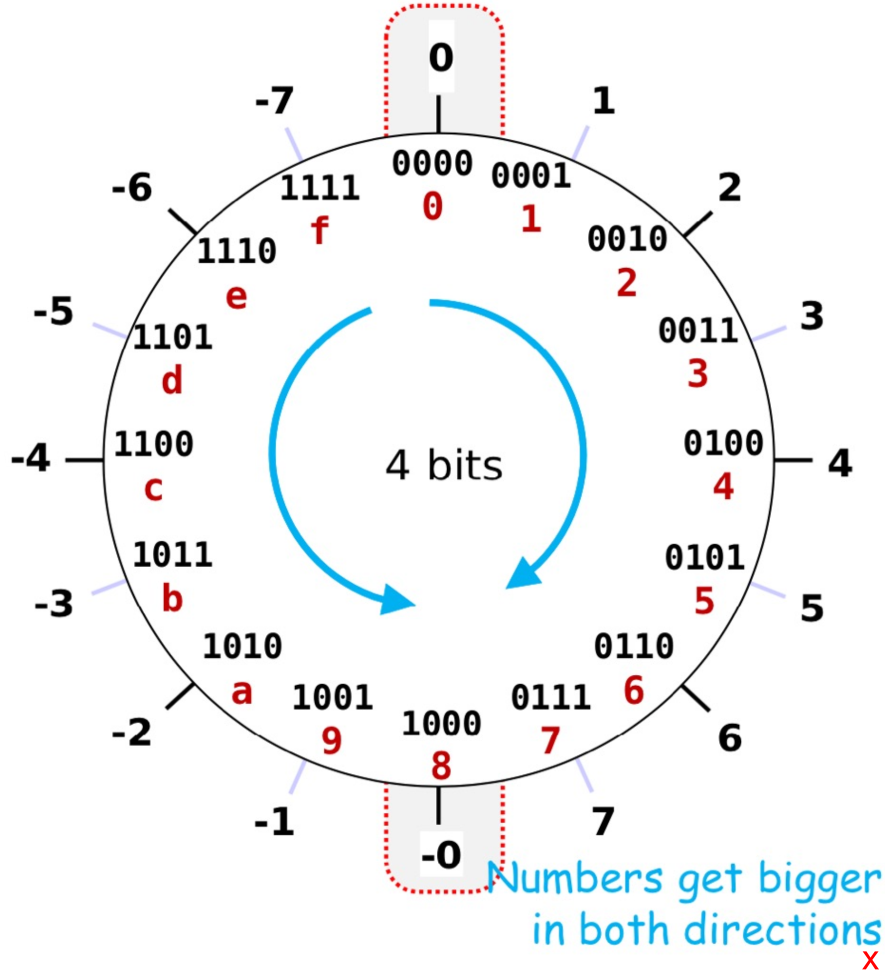
- How do we represent the negative numbers within a fixed number of bits?
  - Allocate some bit patterns to negative and others to positive numbers (and zero)
- 2^n distinct bit patterns to encode positive and negative values
- **Unsigned values:** $0 \ldots 2^n–1$ ← -1 comes from counting 0 as a "positive" number
- **Signed values:** $-2^{n-1} \ldots 2^{n-1}–1$ (dividing the range in ~ half including 0)
- **On a number line (below):** 8-bit integers – signed and unsigned (*e.g.*, `char` **in C**)



Same "width" (same number of encodings), just shifted in value

Negative Integer Numbers: Sign + Magnitude Method

- Use the Most Significant Bit as a sign bit
  - 0 as the MSB represents positive numbers
  - 1 as the MSB represents negative numbers
- Two (oops) representations for zero: 0000, 1000
- Tricky Math (must handle sign bit independently)
  - Positive and Negatives "*increment*" (+1) in the opposite directions!



Signed Magnitude Examples (Sign bit is always MSB)

$$\underbrace{0}_{\text{positive}}\ \underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\ \underbrace{011}_{3}$$

Examples (4 bits):

1 000 = -0	0 000 = 0
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

$$\underbrace{0}_{\text{positive}}\ \underbrace{0000000}_{0}$$

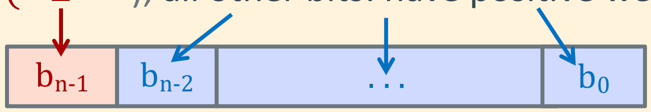
$$\underbrace{1}_{\text{negative}}\ \underbrace{0001100}_{12}$$

x

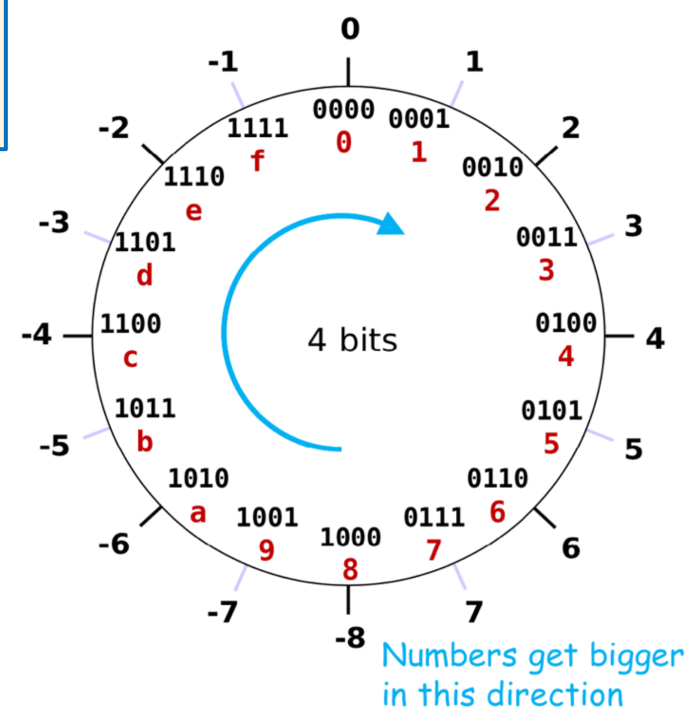
Two's Complement: The MSB Has a *Negative Weight*

$$2's\ Comp = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0$$

b_{n-1} weight is (-2^{n-1}), all other bits: have positive weights $(+2^i)$



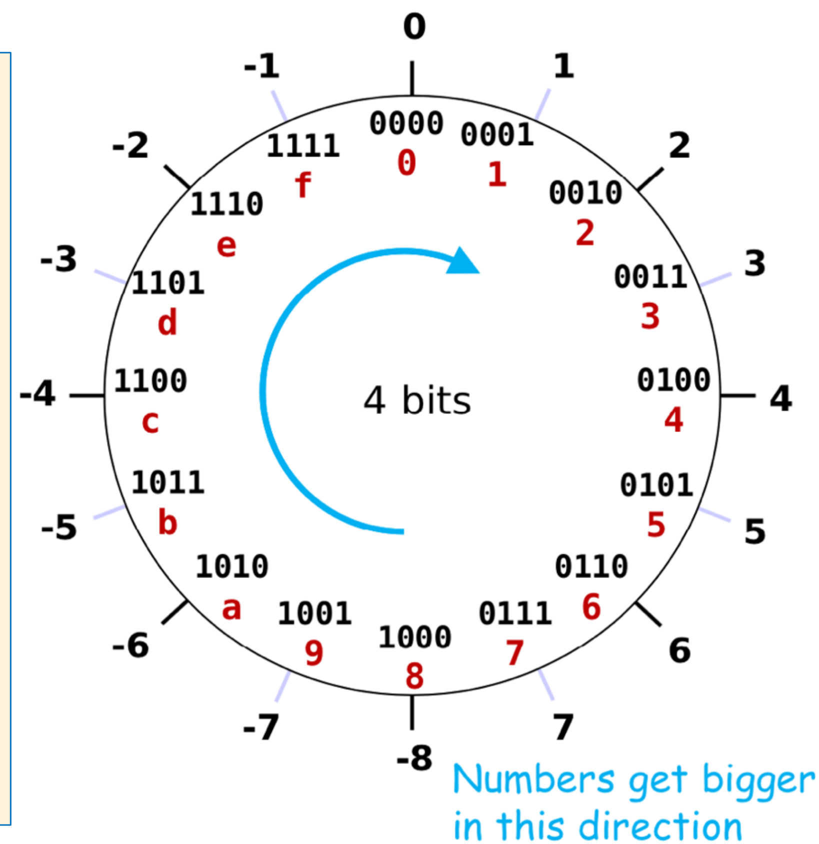
- 4-bit (w = 4) weight = -2^{4-1} = -2^3 = -8
  - 1010_2 **unsigned**:
    $1\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0 =$ **10**
  - 1010_2 **two's complement**:
    $-1\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0 = -8 + 2 =$ **–6**
  - -8 in **two's complement:**
    $1000_2 = -2^3 + 0 = -8$
  - -1 in **two's complement:**
    $1111_2 = -2^3 + (2^3 - 1) = -8 + 7 =$ **-1**



x

2's Complement Signed Integer Method

- Positive numbers encoded same as unsigned numbers
- All negative values have a one in the leftmost bit
- All positive values have a zero in the leftmost bit
  - This implies that 0 is a positive value
- Only one zero
- **For n bits, Number range is $-(2^{n-1})$ to $+(2^{n-1}$ - 1)**
  - Negative values "go 1 further" than the positive values
- Example: the range for 8 bits:
  -128, -127, .. 0, .. 126, **+127**
- Example the range for 32 bits:
  -2147483648 .. 0, .. **+2147483647**
- *Arithmetic is the same as with unsigned binary!*



x

Summary: Min, Max Values: Unsigned and Two's Complement

Two's Complement → Unsigned for n bits

- **Unsigned Value Range**

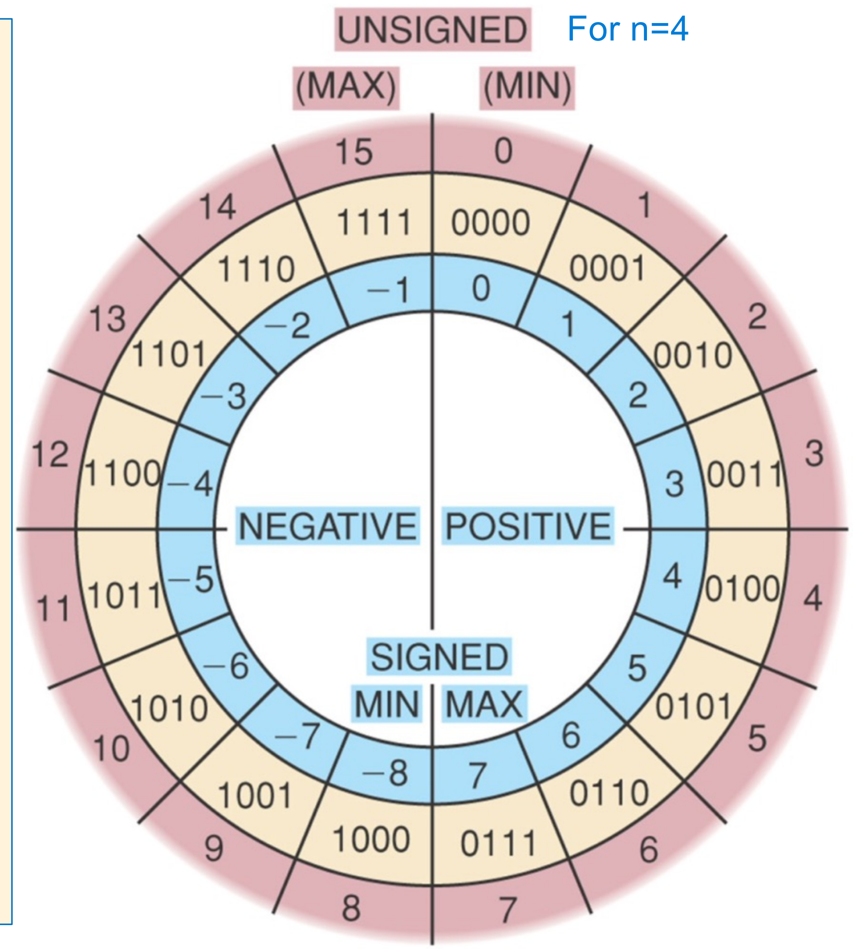
  UMin = 0b00…00
  = 0

  UMax = 0b11…11
  = $2^n - 1$

- **Two's Complement Range**

  SMin = 0b10…00
  = -2^{n-1}

  SMax = 0b01…11
  = $2^{n-1} - 1$



x

Negation Of a Two's Complement Number (Method 1)

```
     7 = 0111

invert = 1000
add 1  +    1
    -7   1001
```

```
    -7 = 1001

invert = 0110
add 1  +    1
     7   0111
```

```
-x == ~x + 1;
```

```
 7 =           0111
-7 =     +     1001
(discard carry) 0000
```

```
     1 = 0001

invert = 1110
add 1  +    1
    -1   1111
```

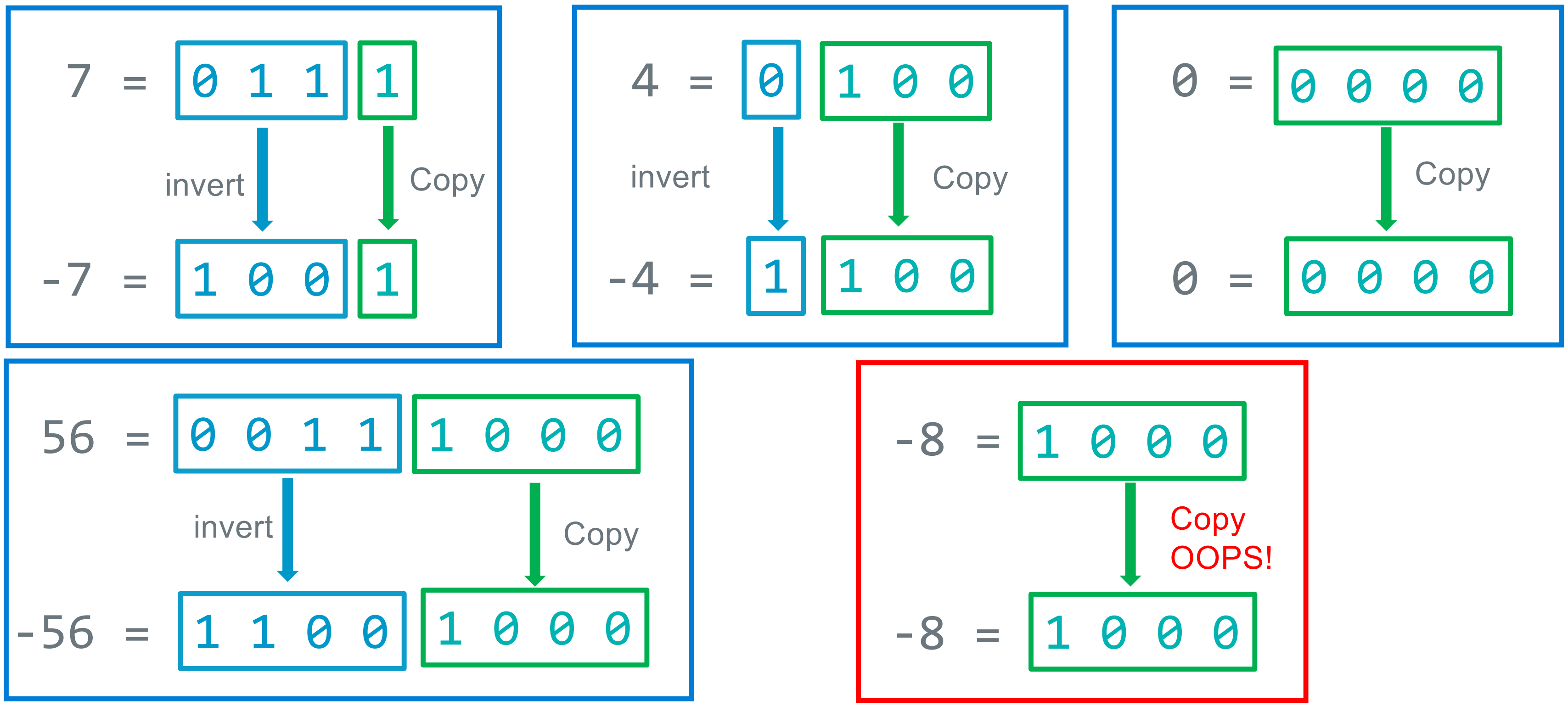
```
    -1 = 1111

invert = 0000
add 1  +    1
     1   0001
```

```
    -8 = 1000

invert = 0111
add 1  +    1
    -8   1000  oops!
```


x

Negation of a Two’s Complement Number (Method 2)

1. **copy unchanged** right most bit containing a 1 and all the 0's to its right
2. Invert all the bits to the left of the right-most 1



x