CS354: Machine Organization and Programming

Lecture 6
Wednesday the September 16th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

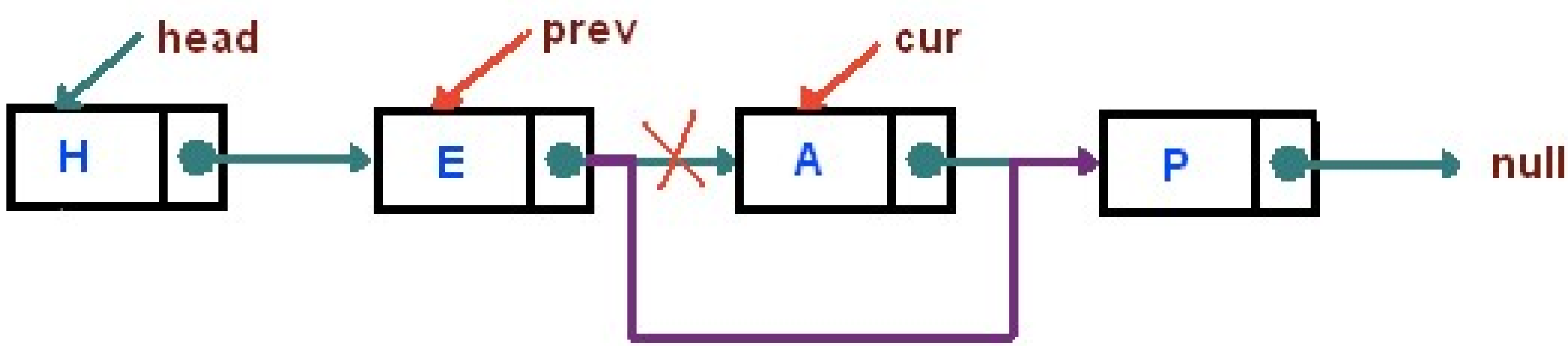
1. How many of you attended the WACM tutorial and found it useful ?
2. **Assignment 1 released - due before 9AM on Sep 30 .**
   - **You can find partners using Piazza too.**
   - **Start Early! Much much harder than Assign 0!**
3. Make sure you don't change your files to add very small changes like formatting, comments etc. after deadline. You get points deducted even if it is a small change.

Lecture Overview

1. Doubly Linked Lists
2. Data Representation (Unsigned , 2's complement)
3. Signed <-> Unsigned Conversions
4. Integer Arithmetic (Addition)

Example C Program on Singly Linked List

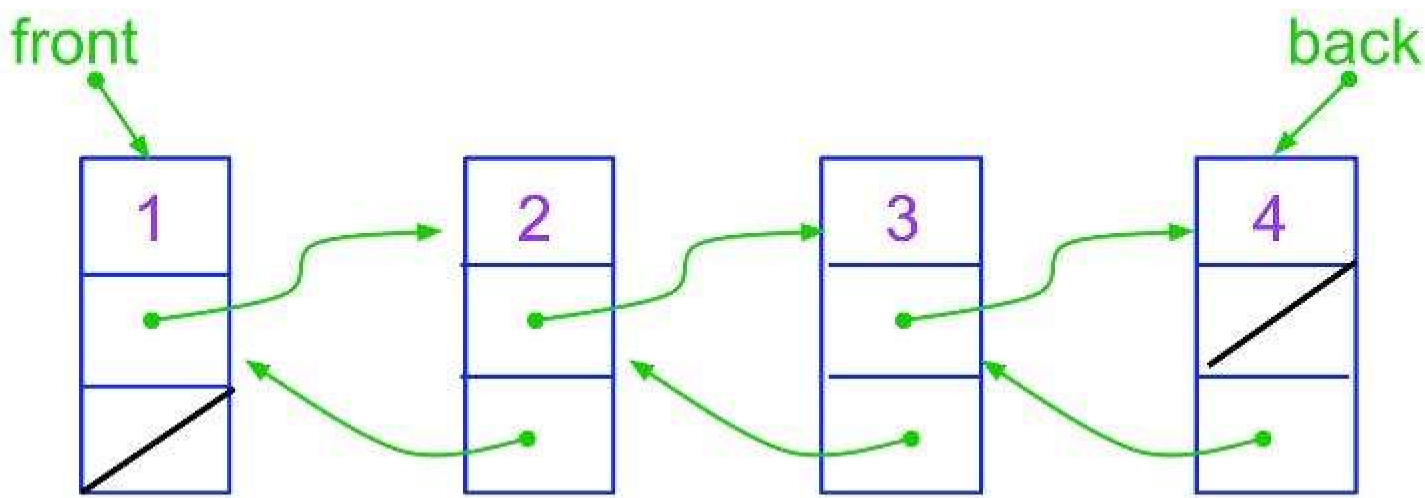
Deleting a node in a Singly Linked List without copying



Doubly Linked List

1. In order to delete a node in a singly linked list without copying values, a pointer to the previous node is also needed.
2. Doubly linked lists allow inserts and deletes in constant number of operations with only the node's address.
3. Doubly linked lists are easier to manipulate they allow fast and easy sequential access to the list in both directions.

```
struct node {
    int theint;
    struct node *next;
    struct node *previous;
};
```



DOUBLY LINKED

For convenience, name this user-defined type:

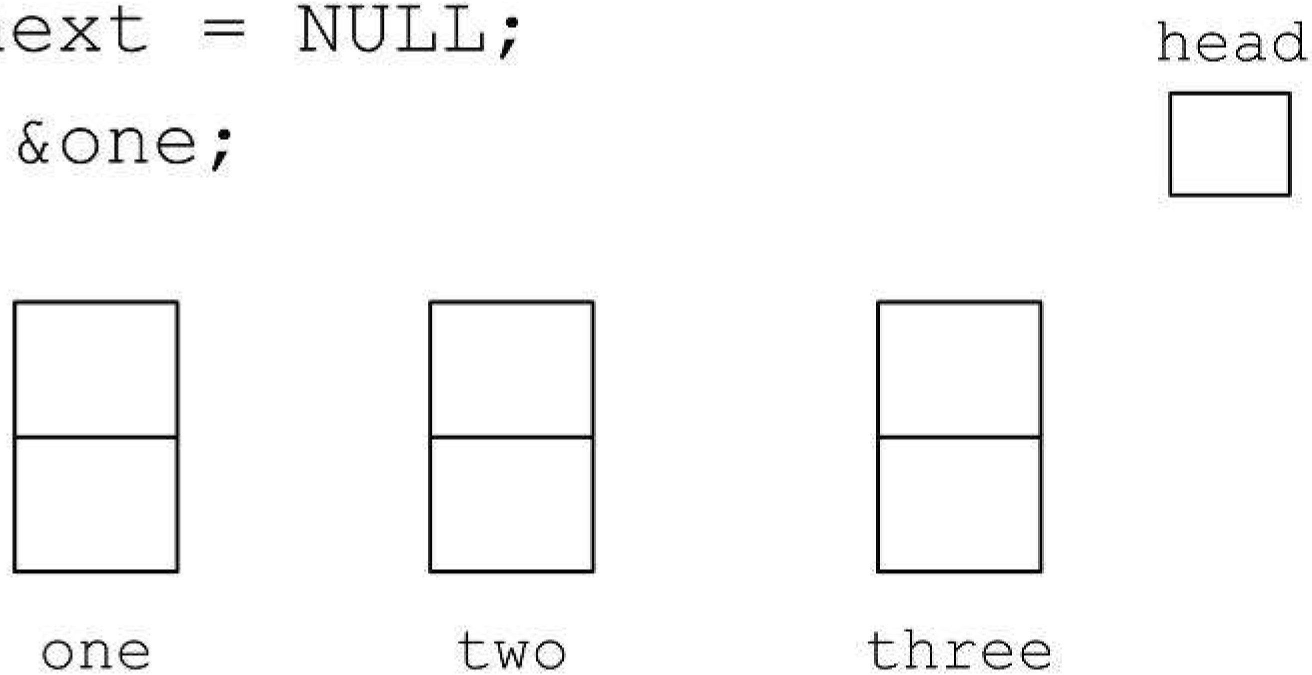
```
typedef struct node {
    int theint;
    struct node *next;
} Node;
```

Now, declarations have less (keyboard) typing:

```
Node one, two, three;
Node *head;
```

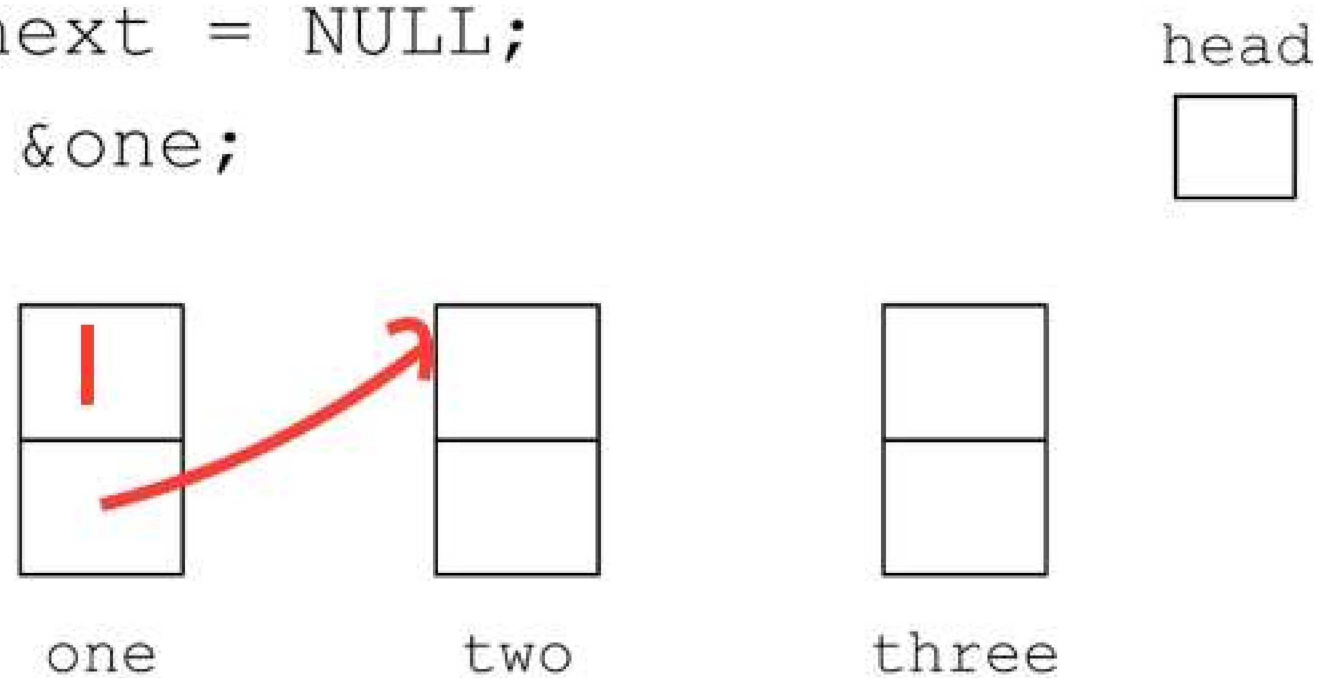
Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```



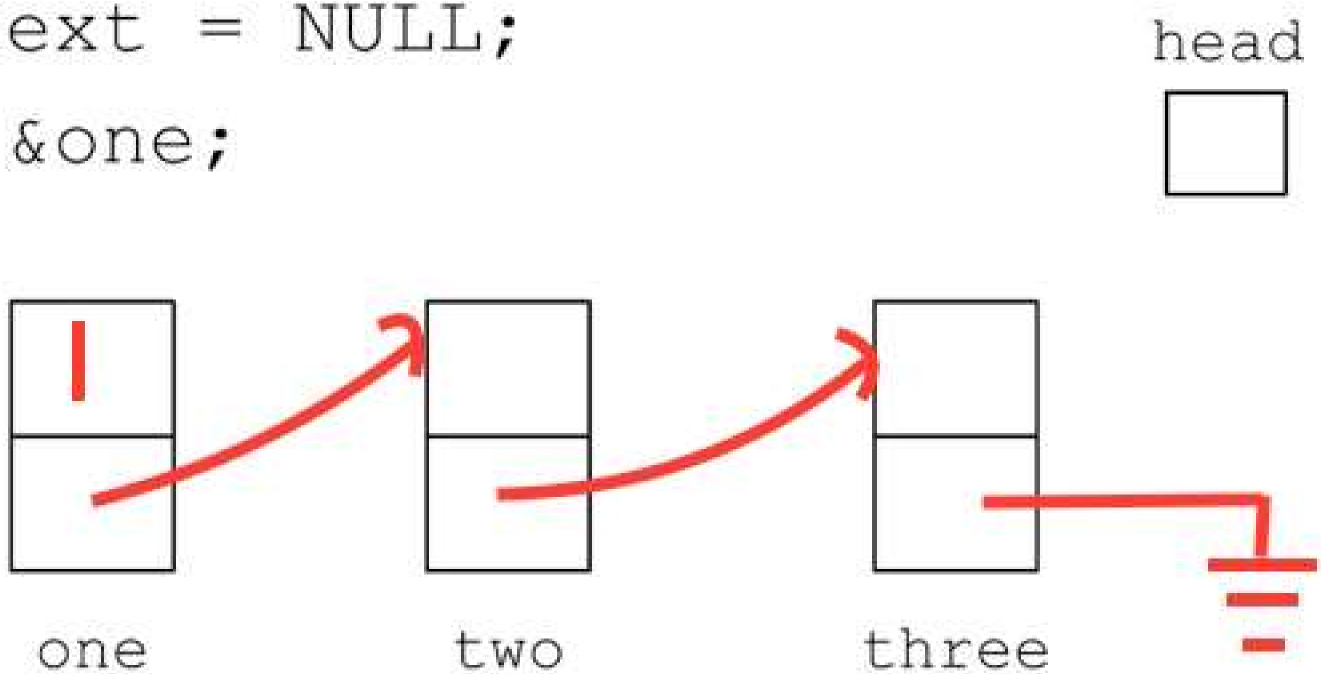
Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```



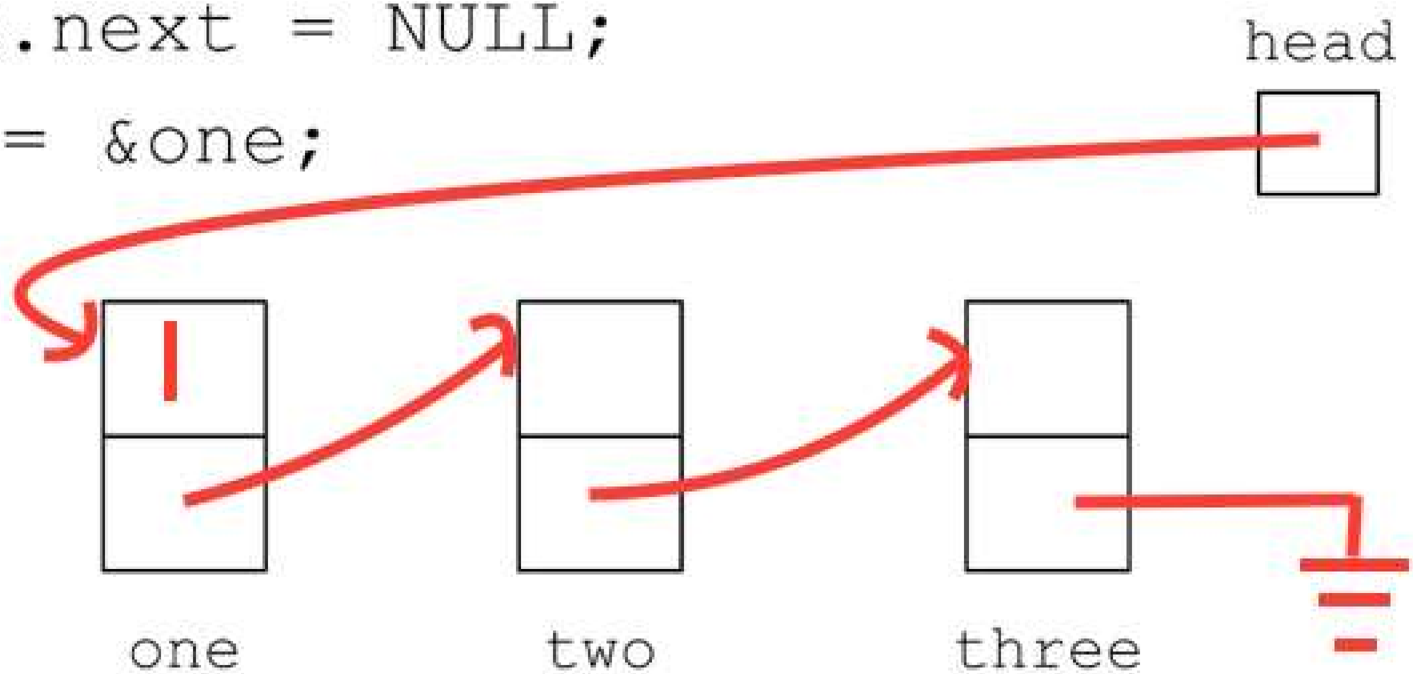
Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```



Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```

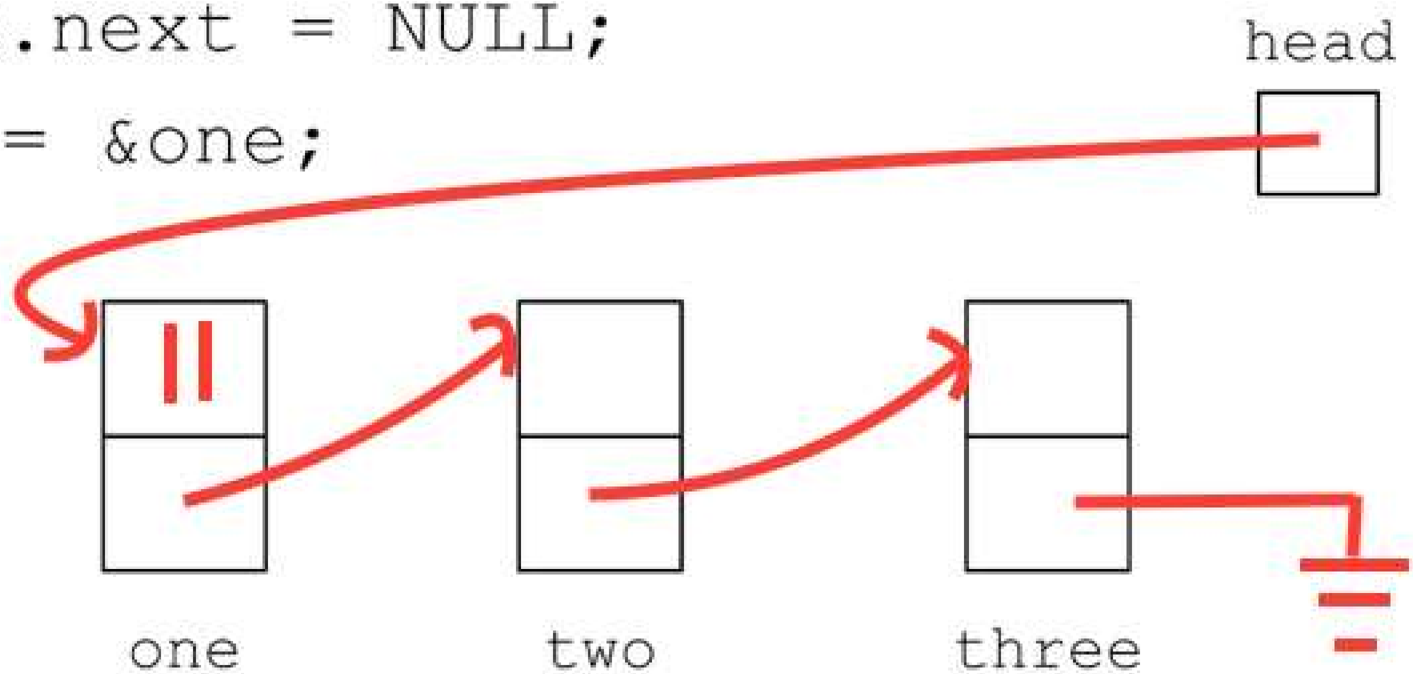


```
int  value = 1;
Node *ptr;

ptr = head;
while (ptr != NULL) {
    ptr->theint = value * 11;
    value++;
    ptr = ptr->next;
}
```

Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```

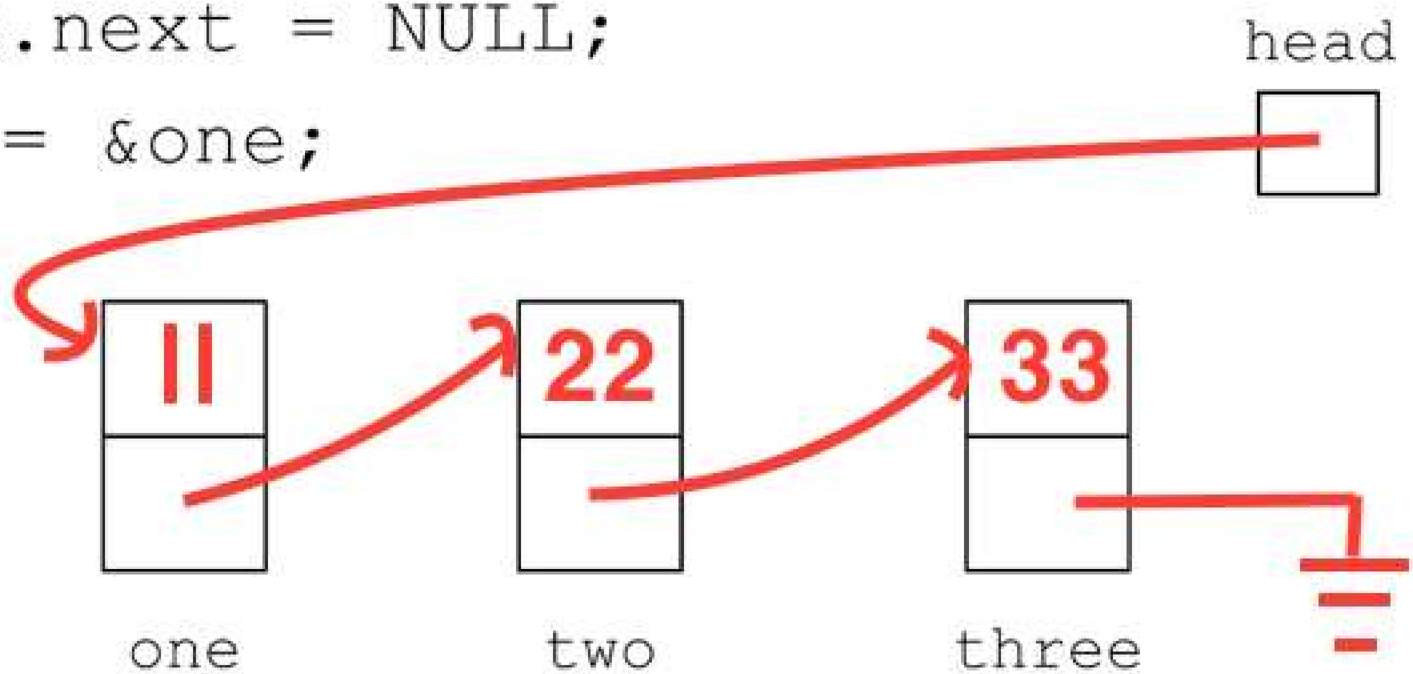


```
int  value = 1;
Node *ptr;

ptr = head;
while (ptr != NULL) {
    ptr->theint = value * 11;
    value++;
    ptr = ptr->next;
}
```

Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```

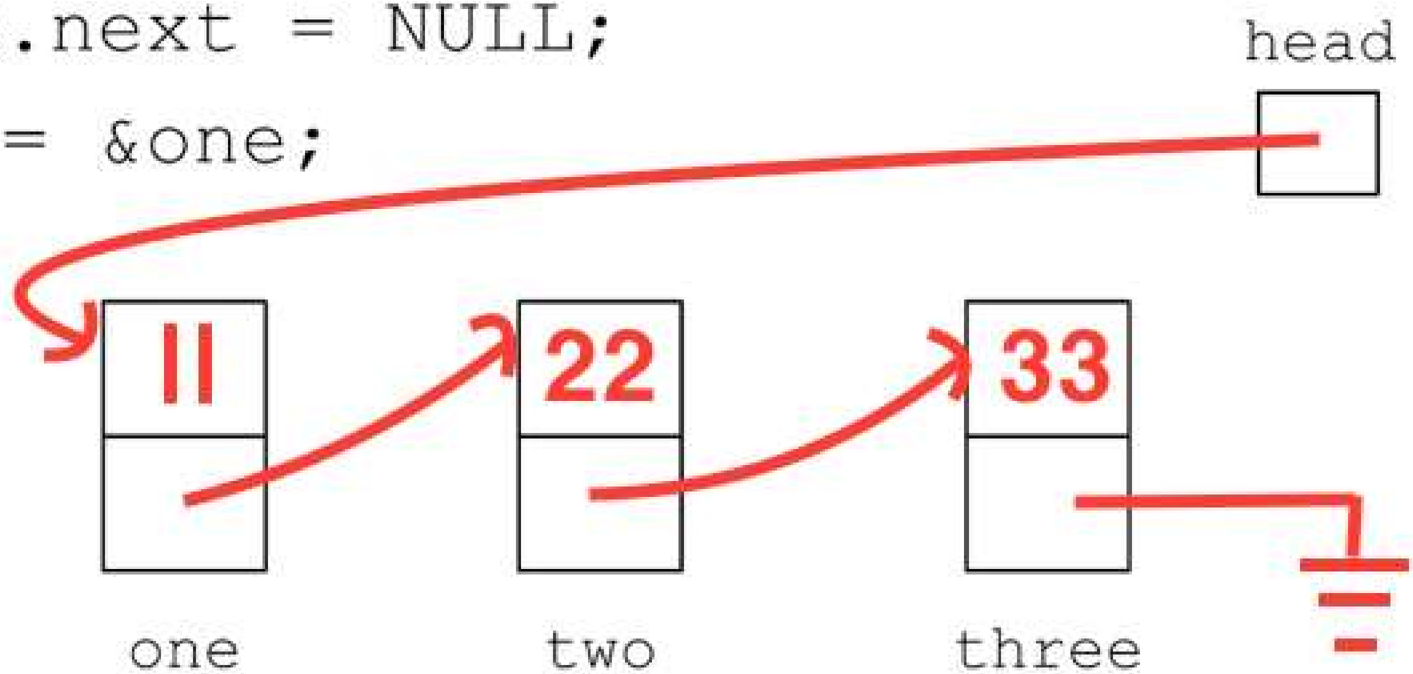


```
int  value = 1;
Node *ptr;

ptr = head;
while (ptr != NULL) {
    ptr->theint = value * 11;
    value++;
    ptr = ptr->next;
}
```

Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```



```
int  value = 1;
Node *ptr;

ptr = head;
while (ptr != NULL) {
    ptr->theint = value * 11;
    value++;
    ptr = ptr.next;
}
```

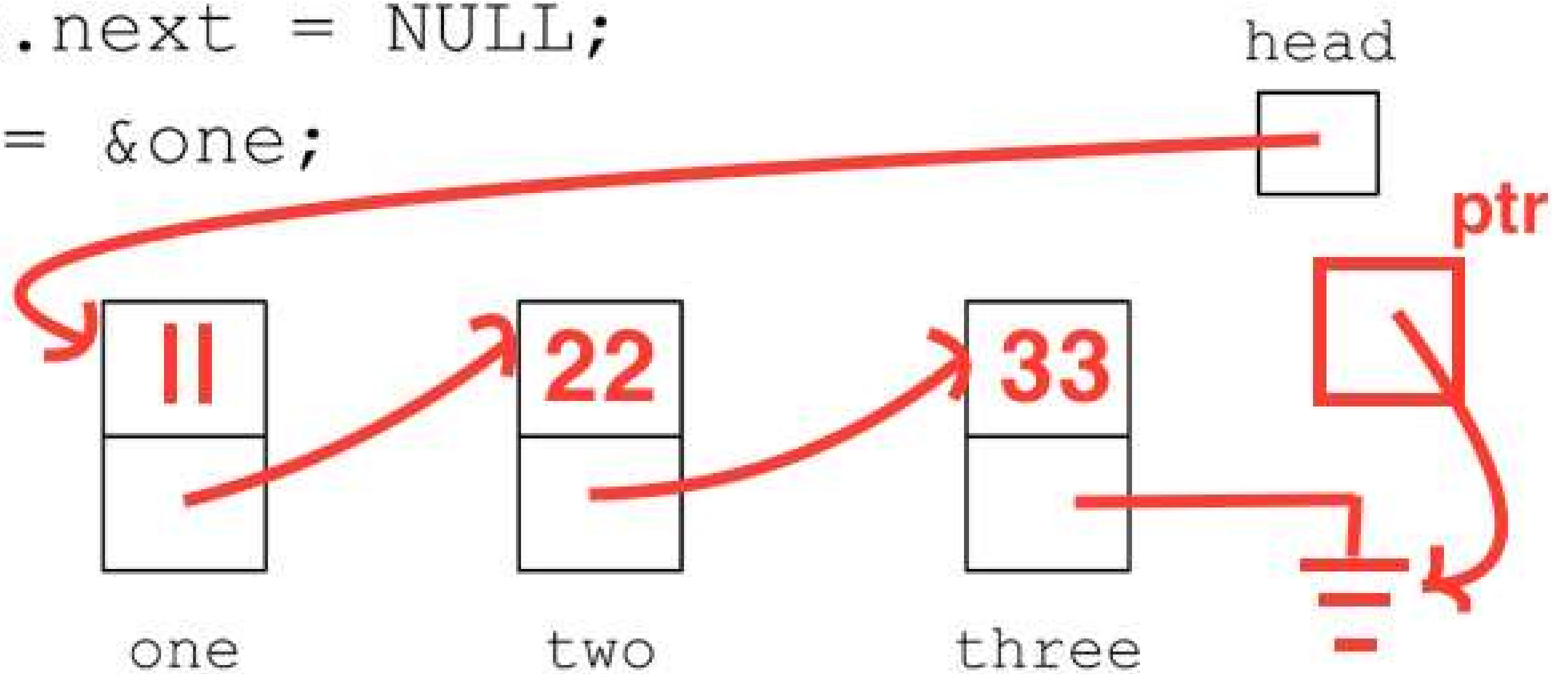
Why is this now incorrect?

With the correct code, what happens when this code is executed?

```
ptr = three.next;
ptr = ptr->next;
```

Some code, to show pointers and such. . .

```
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```



With the correct code, what happens when this code is executed?

```
ptr = three.next;
ptr = ptr->next;
```

Runtime error: NULL pointer dereference

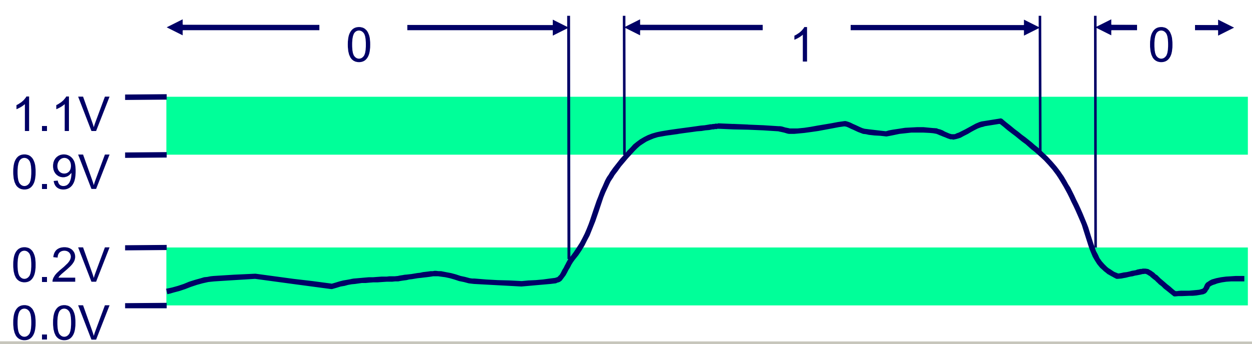
In Linux: Segmentation fault (core dumped)

Example C Program on Doubly Linked List

Detailed Example C Program on Singly Linked List

Bits, Nibbles, Bytes, Words

1. Bits represented using "high & low voltages", "magnetic domain oriented clockwise or anticlockwise" etc.
2. 4 Bits == Nibble ; 8 Bits == Byte ; 16/32/64 Bits == Word (depending on architecture);
3. Group of bits collected together with some *interpretation* is more useful than individual bits.



Word size

1. It is the nomimal size of integers and pointer data
2. Determines the maximum size of virtual address space
3. *w* bit word can address a virtual memory of size (2^w) ranging from 0 to 2^w-1.
4. Modern computers have 64 bit words. (Theoretically: 2^{64} = 16 Exabytes.)

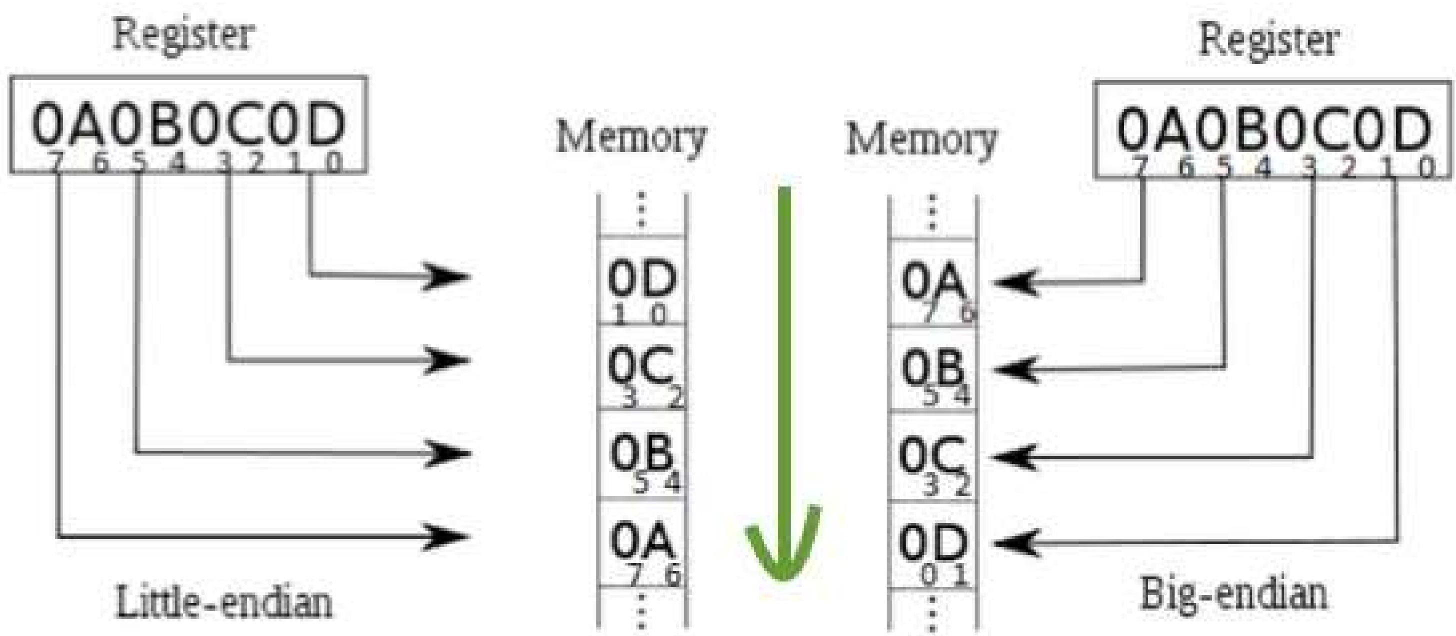
Byte encodings

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Byte Ordering/Endianness

1. Ordering of bytes within a word
2. Little endian – least significant byte comes first
3. Big endian – most significant byte comes first



Representations

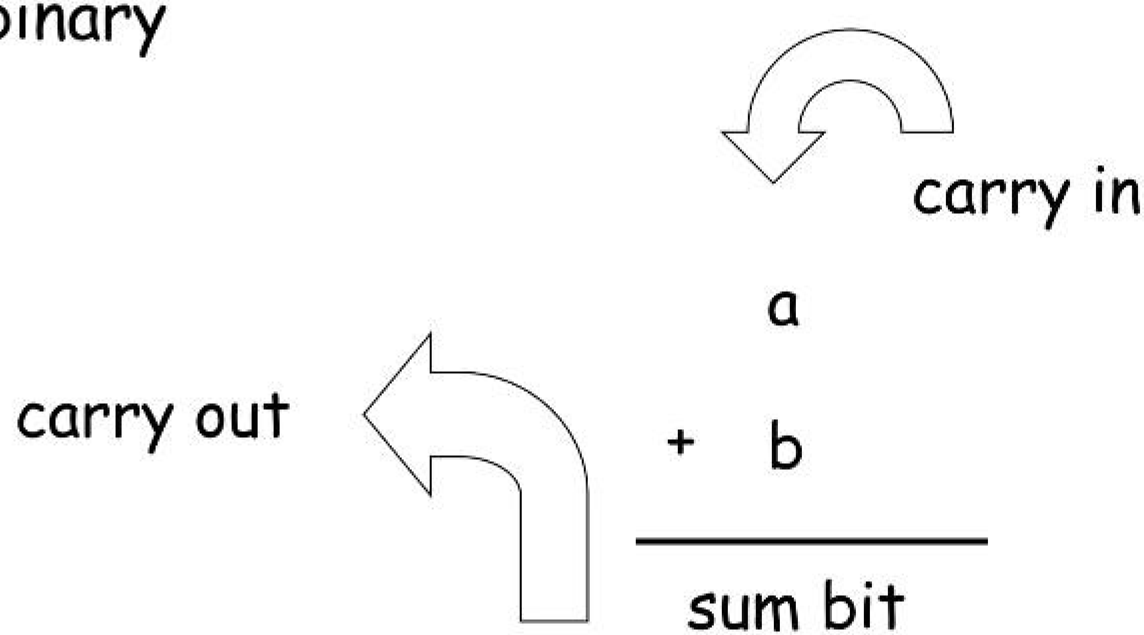
1. Unsigned encodings – positive integers
2. Two's complement – signed integers
3. Floating point – real numbers
4. Because of limited number of bits to encode a number, some operations can "overflow" when results are too large.

Arithmetic Operations

- Arithmetic Operations
  - addition
  - subtraction
  - multiplication
  - division
- Each of these operations on the integer representations:
  - unsigned
  - two's complement

Addition

One bit of binary addition



Addition Truth Table

Carry In	a	b	Carry Out	Sum Bit
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Unsigned Representation

$B2U_w(x_{vec}) = Sum_{i=0->w-1}\ x_i.2^i$

$B2U_4([0101]) = 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 5$

$B2U_w$ is a bijection:
- associates a unique value to each bit vector of length w
- each integer between **0 and 2^w-1** has a unique binary representation as a bit vector of length w

Unsigned Addition

Of two unsigned w bit values X & Y

X + Y equals:

- $X+Y$, if $(X+Y) < 2^w$
- $X+Y-2^w$, if $2^w <= (X+Y) < 2^{w+1}$

Addition

- Unsigned and 2's complement use *the same addition algorithm*
- Due to the fixed precision, throw away the carry out from the msb

```
  00010111
+ 10010010
----------
```

Addition

- Unsigned and 2's complement use *the same addition algorithm*
- Due to the fixed precision, throw away the carry out from the msb

```
  00010111
+ 10010010
----------
  10101001
```

Two's complement Representation

$B2T_w(x_{vec}) = -x_{w-1}2^{w-1} + \text{Sum}_{i=0->w-2}\, x_i 2^i$

$B2T_4([1011]) = -1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = -5$

$B2T_w$ is a bijection:
- associates a unique value to each bit vector of length w
- each integer between **-2^{w-1} and $2^{w-1}-1$** has a unique binary representation as a bit vector of length w

Range of Values for Unsigned and 2's Complement (16 bits)

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

#include <limits.h> declares constants, e.g., ULONG_MAX, LONG_MAX, LONG_MIN

(Values platform specific)

4-bit Unsigned and 2's complement Integers

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1