Lecture 3: Floats and Structs

CS 105

January 29, 2020

Representing Integers

- unsigned:

$$\mathrm{UnsignedValue}(x) = \sum_{j=0}^{w-1} x_j \cdot 2^j$$

- signed (two's complement):

$$\mathrm{SignedValue}(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{j=0}^{w-2} x_j \cdot 2^j$$

Note: to compute –x for a signed int x, flip all the bits, then add 1

$$x + \sim x = 11\ldots1 = -1, \text{ so } x + (\sim x + 1) = 0$$

Example: Three-bit integers

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

- The high-order bit is the *sign bit.*
- The largest unsigned value is $11\ldots1$, UMax.
- The signed value for -1 is always $11\ldots1$.
- Signed values range between `TMin` and TMax.

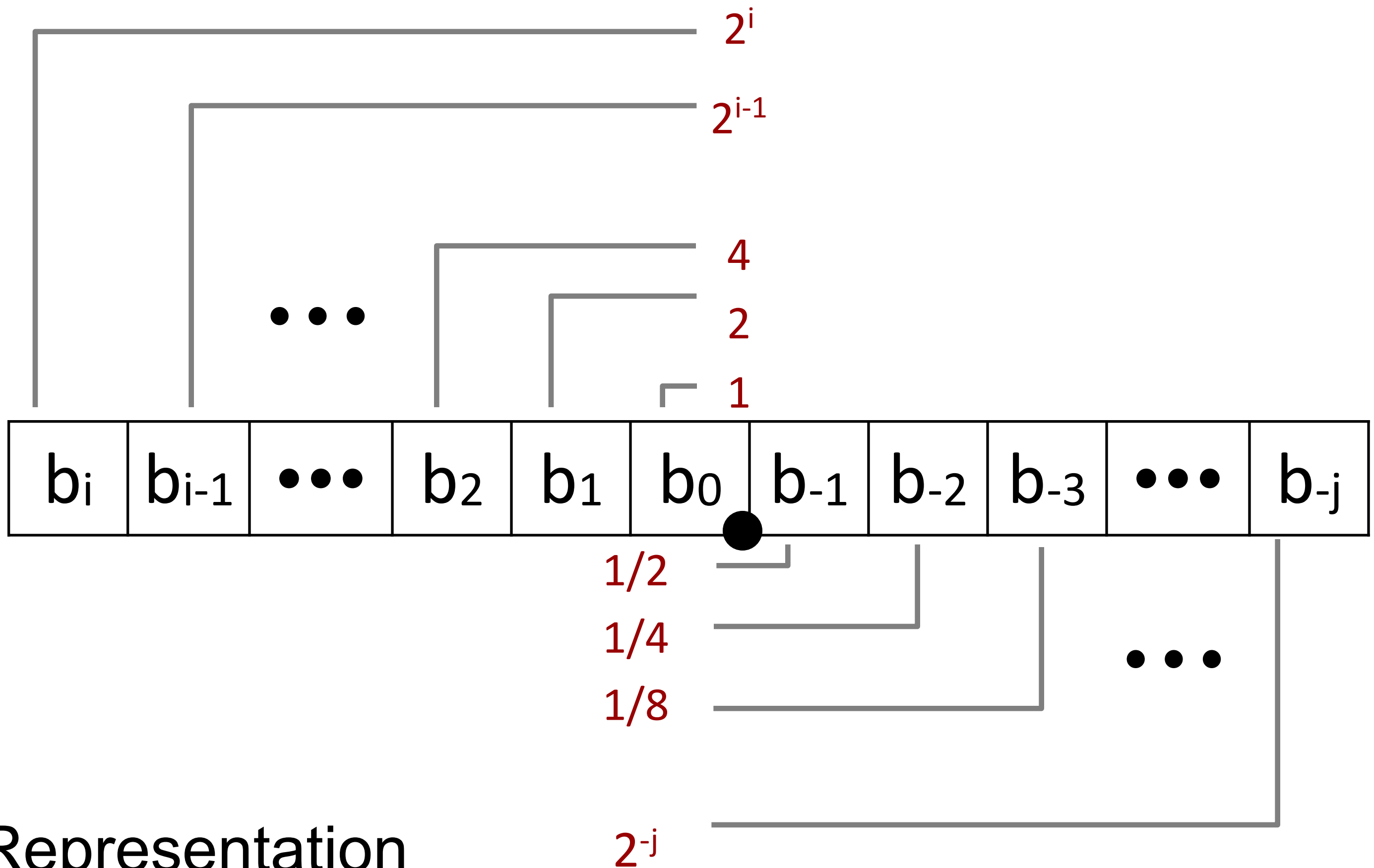
This representation of signed values is called *two's complement.*

FRACTIONAL NUMBERS

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i}(b_k \cdot 2^k)$

Exercise: Fractional Binary Numbers

- Translate the following fractional numbers to their binary representation
  - 5 3/4
  - 2 7/8
  - 1 7/16

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$

Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations
  - Value Representation
    - 1/3 `0.0101010101[01]`…2
    - 1/5 `0.001100110011[0011]`…2
    - 1/10 `0.0001100110011[0011]`…2

- Limitation #2
  - Just one setting of binary point within the *w* bits
  - Limited range of numbers (very small values? very large?)

Floating Point Representation

- Numerical Form: $(-1)^s \cdot M \cdot 2^E$
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0)
  - Exponent E weights value by power of two

- Encoding:

s	$\text{exp} = e_{k-1} \ldots e_1 e_0$	$\text{frac} = f_{n-1} \ldots f_1 f_0$

- s is sign bit s
- exp field encodes E (but is not equal to E)
  - normally $E = e_{k-1} \ldots e_1 e_0 - \boxed{(2^{k-1} - 1)}$ **bias**
- frac field encodes M (but is not equal to M)
  - normally $M = 1.f_{n-1} \ldots f_1 f_0$

Float (32 bits):
- k = 8, n = 23
- bias = 127

Double (64 bits)
- k=11, n = 52
- bias = 1023

Exercise: Floats

- What fractional number is represented by the bytes 0x0000c03e?

Normalized and Denormalized

s	exp	frac

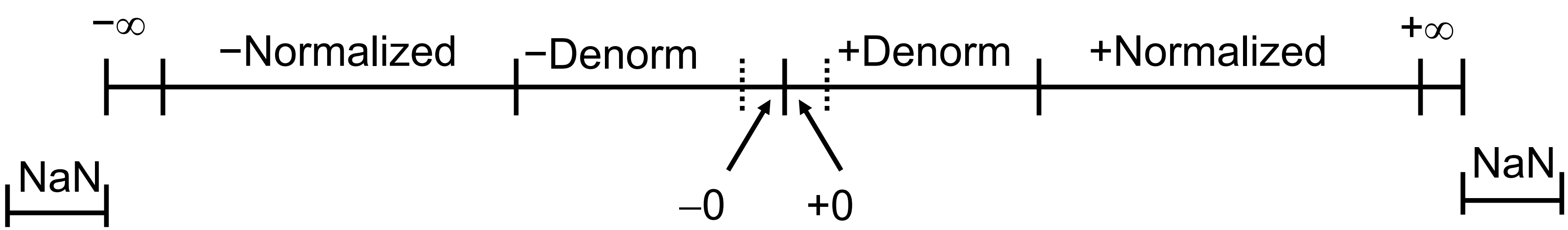
$$(-1)^s \cdot M \cdot 2^E$$

Normalized Values

- exp is neither all zeros nor all ones
- normal case
- exponent is defined as $\mathrm{E} = e_{k-1} \ldots e_1 e_0 - \mathrm{bias}$, where $\mathrm{bias} = 2^k - 1$ (e.g., 127 for float or 1023 for double)
- significand is defined as $M = 1.f_{n-1}f_{n-2} \ldots f_0$

- Denormalized Values
  - exp is either all zeros or all ones
  - if all zeros: $\mathrm{E} = 1 - \mathrm{bias}$ and $M = 0.f_{n-1}f_{n-2} \ldots f_0$
  - if all ones: infinity (if f is all zeros) or NaN

Visualization: Floating Point Encodings



Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision
  - **`double`** double precision
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round

STRUCTS

Structs

- Heterogeneous records, like Java objects
- Example:

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

- Usage:

```
struct rec c;
c.a[0] = 42;
c.next = NULL;
```

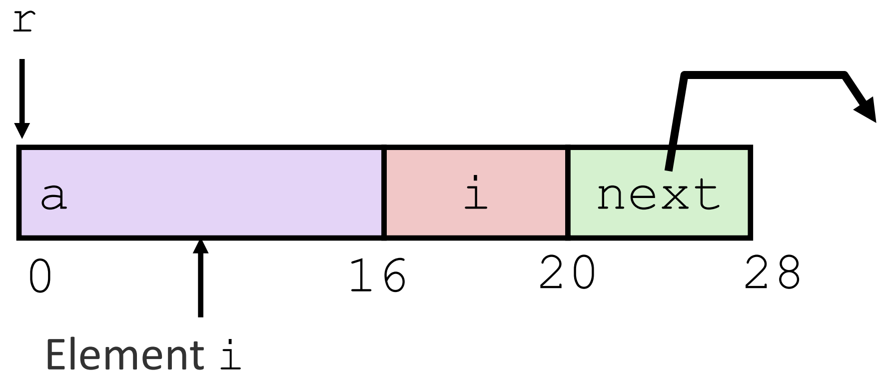
`p->next` is an bbreviation for `(*p).next`

- Pointers:

```
struct rec *p = malloc(sizeof(struct rec));
p->a[0] = 42;
p->next = NULL;
```

Following Linked List

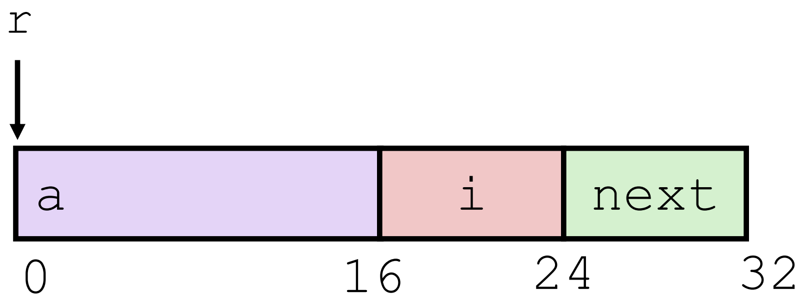
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```




```
void set_val(struct rec *r, int val){
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

Structure Representation

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

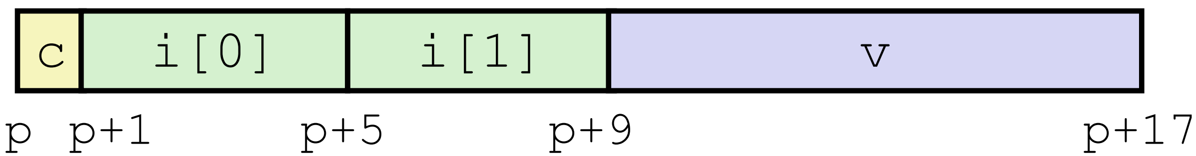


- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

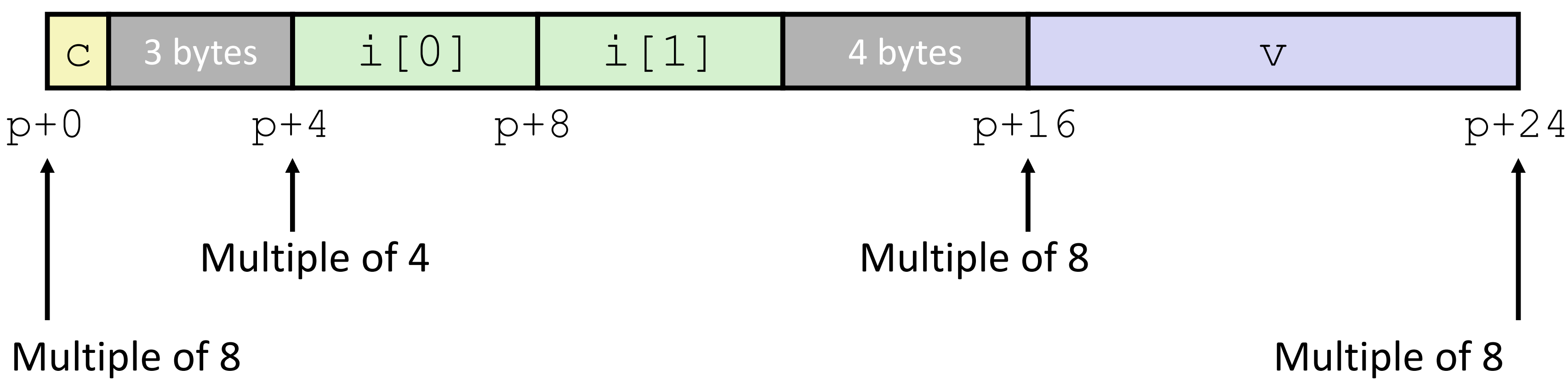
Structures & Alignment

```
struct S1 {
  char c;
  int i[2];
  double v;
};
```

- Unaligned Data



- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

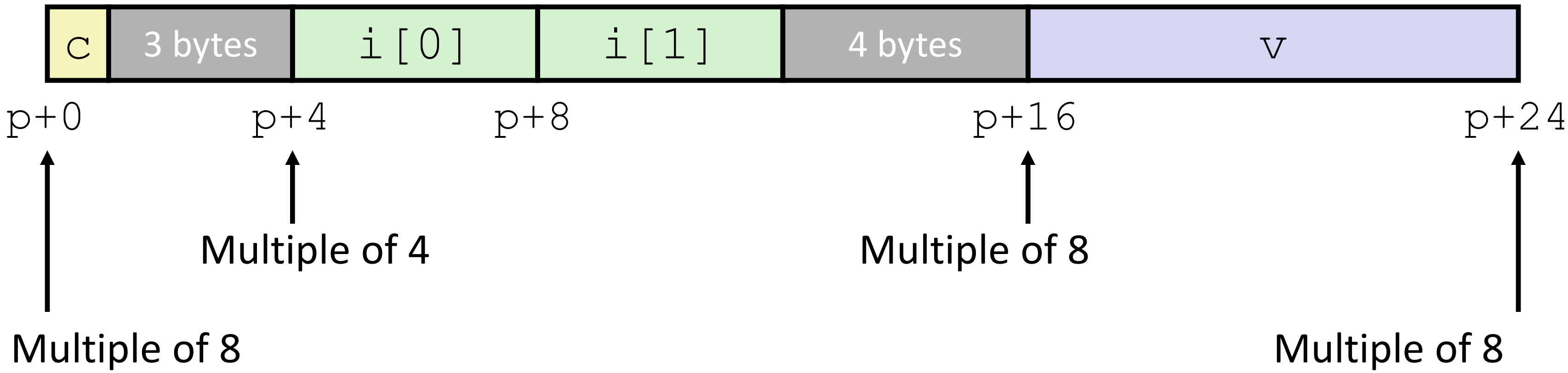
Specific Cases of Alignment (x86-64)

- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, `long`, **`char *`**, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: **`long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement `K`
    - `K` = Largest alignment of any element
  - Initial address & structure length must be multiples of `K`
- Example:
  - K = 8, due to **`double`** element

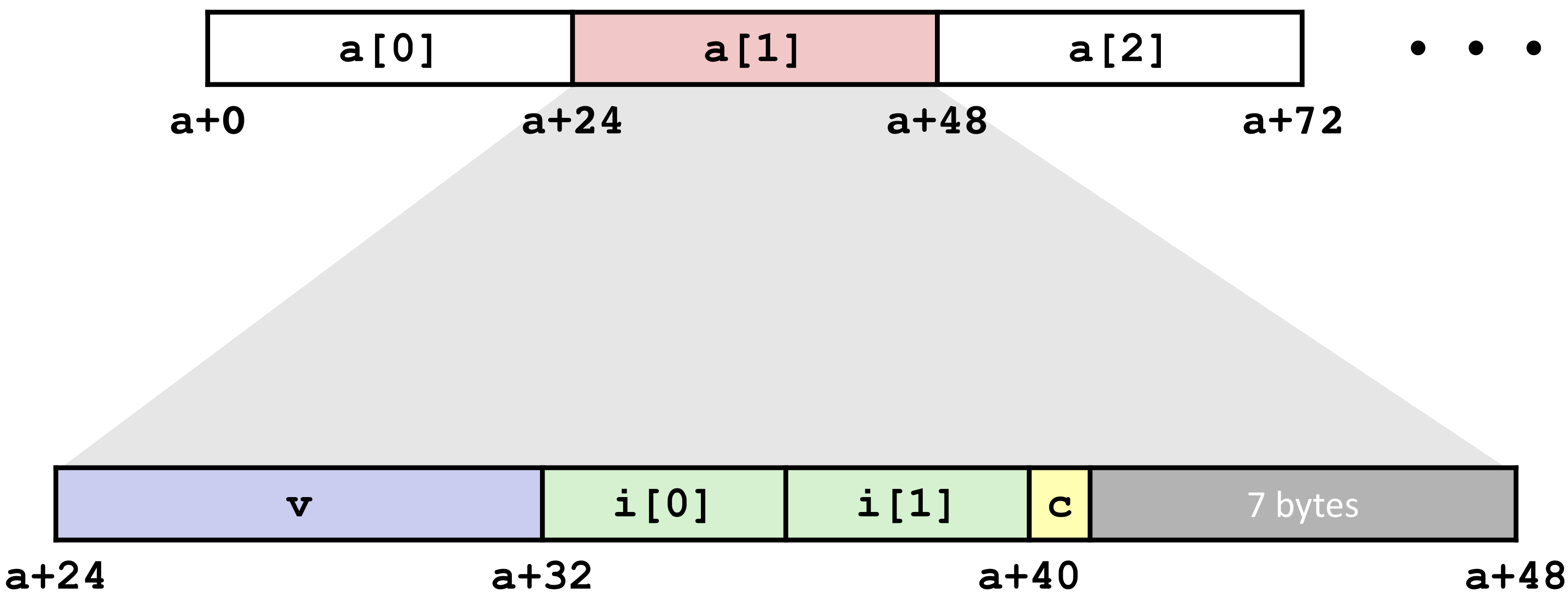
```
struct S1 {
  char c;
  int i[2];
  double v;
};
```



Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S1 {
  char c;
  int i[2];
  double v;
};
```



Saving Space

- Put large data types first

```c
struct S2 {
  char c;
  int i;
  char d;
};
```

```c
struct S3 {
  int i;
  char c;
  char d;
};
```

- Effect (K=4)