# CSE 410: Systems Programming

## Memory Representation

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

# Last Time

- The C programming language
- A bit about POSIX APIs

We spent a little bit of time on:

- Integer types
- Array types
- Pointers

This lecture will dive deeper on these issues.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Memory As Bits

Last lecture, I said “To the computer, memory is just bits.”

While this isn’t precisely true, it’s close enough to get started.

The computer doesn’t “know” about data types.
A modern processor can probably directly manipulate:

- Integers (maybe only of a single bit length!)
- Maybe floating point numbers
- …often, that’s all!

Everything else we create in software.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Memory as …Words?

It is probably more accurate to say memory is just words.

What is a word?

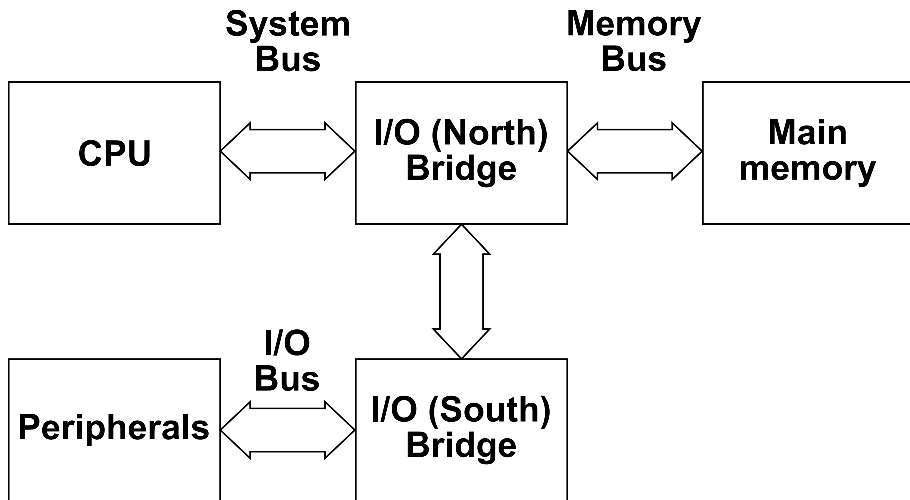
A word is the native integer size of a given platform.
For example, 64 bits on x86-64, or 32 bits on an ARM Cortex-A32.

A word can also (confusingly) be the width of the memory bus, if the processor's word size and its memory bus width are different.

- We will assume they are the same, at least for a while.

What is "native integer size"? What is the "width" of a memory bus?

# A Bit About Architecture





©2018 Ethan Blanton / CSE 410: Systems Programming

# Buses

A bus has a width, which is literally the number of wires it has.[1]

(This is a little less clear on a serial bus, where the width is a protocol convention.)

Each wire transmits one bit per transfer.

*Every bus transfer* is of that width, though some bits may be ignored.

Therefore, memory has a word size from the view of the CPU: the number of wires on that bus.

[1] This is an over-simplification, but it remains true from the point of view of the programmer's model of the processor.


©2018 Ethan Blanton / CSE 410: Systems Programming

# CPU ↔ Memory Transfer

The CPU fetches data from memory in words the width of the memory bus.

It places those words in registers the width of a cpu word.
This register width is the native integer size.[2]

These word widths may or may not be the same.
(On x86-64, they are.)

If they're not, a transfer may require:

- multiple registers, or
- multiple memory transfers.

[2]Some CPUs (including x86-64) can manipulate more than one size of integer in a single register.

# Imposing Structure on Memory

That said, programming languages expose things like:

- Booleans
- classes
- strings
- structures

How is that?

We impose meaning on words in memory by convention.

*E.g.*, as we saw before, a C string is a sequence of bytes that happen to be adjacent in memory.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Hexadecimal

A brief aside: we will be using hexadecimal ("hex") a *lot*.

Hex is the base 16 numbering system.
One hex digit ranges from 0 to 15.

Contrast this to decimal, or base 10 —
one decimal digit ranges from 0 to 9.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Hexadecimal

A brief aside: we will be using hexadecimal ("hex") a *lot*.

Hex is the base 16 numbering system.
One hex digit ranges from 0 to 15.

Contrast this to decimal, or base 10 —
one decimal digit ranges from 0 to 9.

In computing, hex digits are represented by 0-9 and then A-F.

| | |
|---|---|
| A = 10 | D = 13 |
| B = 11 | E = 14 |
| C = 12 | F = 15 |


©2018 Ethan Blanton / CSE 410: Systems Programming

# Why Hex?

Hexadecimal is used because one hex digit is four bits.

This means that two hex digits represents one 8-bit byte.

On machines with 8-bit-divisible words, this is *very convenient*.

| Hex | Bin | Hex | Bin |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | A | 1010 |
| 3 | 0011 | B | 1011 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | D | 1101 |
| 6 | 0110 | E | 1110 |
| 7 | 0111 | F | 1111 |


©2018 Ethan Blanton / CSE 410: Systems Programming

# The C Struct

A `struct` is a compound data type consisting of one or more primitive types.

```
struct IntList {
    int             value;
    struct IntList *next;
};
```

This struct contains an integer and a pointer.

The integer `value` comes before the pointer `next` in memory.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Padding and Alignment

In structures, as in most data on real systems, types are aligned.

This means that the address of a variable is evenly divisible by the size of its type.

Thus, if an `int` is 32 bits, its address is divisible by 4.
(32 bits / 8 bits per byte = 4 bytes, addressed in bytes.)

Padding is used between items to bring them into alignment.

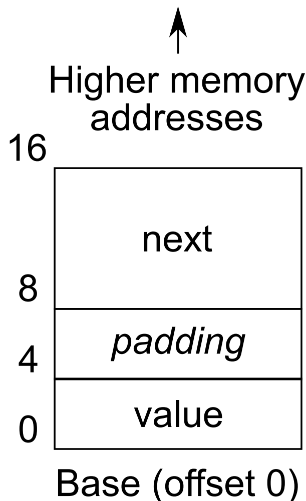
The address of a structure is typically divisible by the largest type it contains.

Padding and alignment rules vary widely by architecture and implementation!


©2018 Ethan Blanton / CSE 410: Systems Programming

# Struct Layout

```
struct IntList {
    int            value;
    struct IntList *next;
};
```





©2018 Ethan Blanton / CSE 410: Systems Programming

# Consequences of Alignment

On many systems, unaligned accesses result in error conditions.

An unaligned access is the attempt to manipulate a piece of data which has an address not evenly divisible by its size.

This often results in a bus error.

On other systems, unaligned accesses are merely slow.
(This is because multiple memory accesses are required, due to the fact that memory can only fetch aligned words.)

x86-64 is the latter type of system; it will “fix up” most unaligned accesses.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Declaring and Using Structures

We previously saw a structure declaration. The syntax is:

```
struct StructureTypeName {
    // Types in structure
    int whatever;
} instance; // semicolon required!
```

An instance of the structure may be created where the structure is declared, or using the type name later:

```
struct StructureTypeName instance;
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointers Review

Pointers are variables containing an address.

The data at that address can be manipulated through the pointer.

Pointers are:

- Declared with *
- Dereferenced with *, [], and ->
- Created from variables with &

```
int  *x   = &someint;
char *str = "A string constant";
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointers to Structures

We put off discussing -> before.

This operator is used for dereferencing pointers to structures:

```
struct Complex {
    double r;
    double i;
} complex;
struct Complex *pc = &complex;
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointers to Structures

We put off discussing -> before.

This operator is used for dereferencing pointers to structures:

```
struct Complex {
    double r;
    double i;
} complex;
struct Complex *pc = &complex;
```

These are equivalent:

```
(*pc).i;

pc->i;
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointers and Alignment

Pointers have alignment just like other types.

There are two alignments to be aware of:

- Alignment of the pointer itself
- Alignment of the object pointed to

The former is typically the pointer size.

The latter is the type's natural alignment:

- Bit width for integers
- Pointer width for pointers
- Width of the widest item in a structure
- *etc.*


©2018 Ethan Blanton / CSE 410: Systems Programming

# Arrays and Alignment

Arrays depart from pointer behavior with respect to alignment.

On many platforms, the first element of an array has a minimum alignment of a pointer word.

However, they often align wider for larger types.
(*E.g.*, with a 4 byte pointer, a double array would still align to 8 bytes.)

On some platforms, they may align narrower for narrower types.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointer Arithmetic

In addition to alignment, pointers have stride.

Stride is the distance between elements of the pointed-to type.

Pointer arithmetic operates in stride-sized chunks.
(This is why pointers can dereference like arrays!)

```
double *dptr = &somedouble;
```

If the value of `dptr` were `0`, `dptr + 1` would be eight, not one!

This is because a double is 8 bytes wide.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointer Types

A pointer's type tells the compiler what type to *treat it as*.

This is almost, but not entirely, different from what it actually is.

Nothing stops you from taking a valid pointer and doing invalid arithmetic.

Similarly, nothing stops you from pointing at nonsense.


©2018 Ethan Blanton / CSE 410: Systems Programming

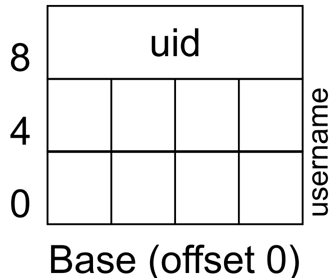
# Pointer Danger

```
struct Example {
    char username[8];
    int   uid;
} example;

char *name =
   &example.username;
```

What is username[8]?

What is name[8]?





©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointer Safety

The compiler tries to help you make safe pointer accesses.

```
struct Complex {
    double r;
    double i;
} complex;

double *d = &complex;
```

This results in a compiler error:

```
complex.c:6:10: warning: initialization from
   incompatible pointer type
   [-Wincompatible-pointer-types]
 double *d = &complex;
             ^
```

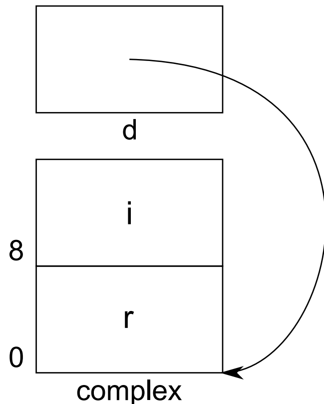

©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointer Safety

Is it right, though?

```
struct Complex {
    double r;
    double i;
} complex;

double *d = &complex;
```





©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointer Safety

Probably.

This pointer happens to be valid, but if the programmer meant it, she probably should have used:

```
double *d = &complex.r;
```

This practice:

- Tells the compiler you intend for d to point to r
- Tells the next programmer what you really meant
- Is somewhat more robust to changes in the structure


©2018 Ethan Blanton / CSE 410: Systems Programming

# Pointer Daring

Sometimes you really do want to create a dissimilar pointer.

In this case, you should cast your pointer type.

```
struct Complex {
    double r;
    double i;
} complex;

char *bytes= (char *)&complex;
```

This forces the compiler to allow the assignment.

The special type `void *` can be assigned to any pointer.


©2018 Ethan Blanton / CSE 410: Systems Programming

# The sizeof operator

There are several operators used to help with reflection in C.

One of these is the `sizeof` operator.

It returns the size in bytes of its operand, which can be:

- A variable
- An expression that is “like” a variable
- A type

(Expressions “like” a variable include, *e.g.*, members of structures.)


©2018 Ethan Blanton / CSE 410: Systems Programming

# Looking at sizeof

Examples:

```
char str[32];
int matrix[2][3];

sizeof(int);                        // yields 4
sizeof(str);                        // yields 32
sizeof(matrix);                     // yields 24
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Dynamic Allocation

The next thing to learn about pointers is dyamic allocation.
(We'll talk *a lot more* about this later.)

The header `stdlib.h` defines `malloc()` and `free()`:

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```

Malloc will allocate memory, while free will release it.


©2018 Ethan Blanton / CSE 410: Systems Programming

# malloc

```
void *malloc(size_t size);
```

Malloc returns a `void *` pointer, which can point to anything.

To allocate an array with 10 `int` entries dynamically, we:

- Determine the size of a single `int`
- Tell the system we want ten of those
- Assign the result to an appropriate pointer

```
int *array = malloc(10 * sizeof(int));
```

The variable `array` can now be used as a regular `int` array.


©2018 Ethan Blanton / CSE 410: Systems Programming

# free

```
void free(void *ptr);
```

Free accepts a `void *` pointer, which can point to anything.

Freed memory returns to the system to be allocated again later via `malloc()`.

```
free(array);
```

Note that free does not modify the value of its argument. Thus you cannot "tell" that a pointer has been freed!


©2018 Ethan Blanton / CSE 410: Systems Programming

# Structure Allocation

We can allocate, use, and free a structure thus:

```
struct Complex {
    double r;
    double i;
};

struct Complex *c = malloc(sizeof(struct Complex));

c->r = 1.0;
c->i = 0.0;

free(c);
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Exploring Representation

Let's use pointer casting to explore some in-memory data.

```
#include <stdio.h>

void dump_mem(const void *mem, size_t len) {
    const char *buffer = mem;   // Cast to char *
    size_t i;

    for (i = 0; i < len; i++) {
        if (i > 0 && i % 8 == 0) { printf("\n"); }

        printf("%02x ", buffer[i] & 0xff);
    }
    if (i > 1 && i % 8 != 1) { puts(""); }
}
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# dump_mem Details

What is this for?

```
const char *buffer = mem;
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.


©2018 Ethan Blanton / CSE 410: Systems Programming

# dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

It prints a newline after every 8th byte excepting the first.


©2018 Ethan Blanton / CSE 410: Systems Programming

# dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

It prints a newline after every 8th byte excepting the first.

Finally:

```
buffer[i] & 0xff
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# dump_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ printf("\n"); }
```

It prints a newline after every 8th byte excepting the first.

Finally:

```
buffer[i] & 0xff
```

This is necessary to avoid sign extension.
We’ll talk more about this later.


©2018 Ethan Blanton / CSE 410: Systems Programming

# A Simple Integer

First, a simple integer:

```
int x = 98303; // 0x17fff
dump_mem(&x, sizeof(x));
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# A Simple Integer

First, a simple integer:

```
int x = 98303; // 0x17fff
dump_mem(&x, sizeof(x));
```

Output:

```
ff 7f 01 00
```

Let's pull this apart.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Byte Ordering

Why is 98303, which is `0x17fff`, represented by `ff 7f 01 00`?


©2018 Ethan Blanton / CSE 410: Systems Programming

# Byte Ordering

Why is 98303, which is `0x17fff`, represented by `ff 7f 01 00`?

The answer is endianness.

Words are organized into bytes in memory — but in what order?

- Big Endian: The "big end" comes first.
  This is how we write numbers.
- Little Endian: The "little end" comes first.
  This is how x86 processors (and others) represent integers.

You cannot assume anything about byte order in C!


©2018 Ethan Blanton / CSE 410: Systems Programming

# Sign Extension

```
char c = 0x80;
int  i = c;

dump_mem(&i, sizeof(i));
```

# Sign Extension

```
char c = 0x80;
int  i = c;

dump_mem(&i, sizeof(i));
```

Output:

```
80 ff ff ff
```

`0xffffff80`? Where did all those one bits come from?!

# Positive Integers

A formal definition of a positive integer on a modern machine is:

Consider an integer of width $w$ as a vector of bits, $\vec{x}$:

$$\vec{x} = x_{w-1}, x_{w-2}, \dots, x_0$$

This vector $\vec{x}$ has the decimal value:

$$\vec{x} \doteq \sum_{i=0}^{w-1} x_i 2^i$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Calculating Integer Values

Consider the 8-bit binary integer `0010 1011`:

$$\begin{aligned} 0010\,1011\text{b} &= 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 0 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 \\ &= 32 + 8 + 2 + 1 \\ &= 43 \end{aligned}$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Negative Integers

Previously, the variable `c` was sign extended into `i`.

As previously discussed, integers may be signed or unsigned.

Since integers are just bits, the negative numbers must have different bits set than their positive counterparts.

There are several typical ways to represent this, the most common being:

- One's complement
- Two's complement


© 2018 Ethan Blanton / CSE 410: Systems Programming

# One's Complement

One's complement integers represent a negative by inverting the bit pattern.

Thus, a 32-bit 1:

```
00000000 00000000 00000000 00000001
```

And a 32-bit -1:

```
11111111 11111111 11111111 11111110
```

Formally, this is like a positive integer, except:

$$x_{w-1} \doteq -2^{w-1} + 1$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Decoding Negative One's Complement

Therefore, 4-bit `-1`: `1110`

$$\begin{aligned} 1110b &= 1 \cdot (-2^3 + 1) + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot -7 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= -7 + 4 + 2 \\ &= -1 \end{aligned}$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Decoding Negative One's Complement

Therefore, 4-bit −1: `1110`

$$\begin{aligned}
1110b &= 1 \cdot (-2^3 + 1) + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= 1 \cdot -7 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\
&= -7 + 4 + 2 \\
&= -1
\end{aligned}$$

This is fine, except there are two zeroes!:

$$\begin{aligned}
0000b &= 0 \cdot (-2^3 + 1) + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
1111b &= 1 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= -7 + 4 + 2 + 1
\end{aligned}$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Two's Complement

Most (modern) machines use two's complement.

Two's complement differs *slightly* from one's complement.
Its $w - 1$th bit is defined as:

$$x_{w-1} \doteq -2^{w-1}$$

(Recall that one's complement added 1 to this!)

This means there is only one zero — all 1s is -1!


©2018 Ethan Blanton / CSE 410: Systems Programming

# Decoding Two's Complement

Consider 1110 in two's complement:

$$
\begin{aligned}
1110\text{b} &= 1 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= -8 + 4 + 2 + 0 \\
&= -2
\end{aligned}
$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Decoding Two's Complement

Consider 1110 in two's complement:

$$
\begin{aligned}
1110\text{b} &= 1 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= -8 + 4 + 2 + 0 \\
&= -2
\end{aligned}
$$

$w$-bit Two's complement integers run from $-2^{w-1}$ to $2^{w-1} - 1$.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Negative Integer Bit Patterns

*In general*, the high-order bit of a negative integer is 1.

In our previous example:

```
char c = 0x80;
int  i = c;
```

`c` is signed, and thus equivalent to -128.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Negative Integer Bit Patterns

*In general*, the high-order bit of a negative integer is 1.

In our previous example:

```
char c = 0x80;
int  i = c;
```

`c` is signed, and thus equivalent to -128.

It is then sign extended into `i` by duplicating the high bit to the left.

This results in an `i` that also equals -128.

Why?


©2018 Ethan Blanton / CSE 410: Systems Programming

# Computing `c` and `i`

```c
char c = 0x80;
```

Here, `c` is -128 plus no other bits set.

```c
int i = c;
```

What is `i` if we sign extend?


©2018 Ethan Blanton / CSE 410: Systems Programming

# Computing `c` and `i`

```
char c = 0x80;
```

Here, `c` is -128 plus no other bits set.

```
int i = c;
```

What is `i` if we sign extend?

`11111111 11111111 11111111 10000000`

What is the value of that two's complement integer?


©2018 Ethan Blanton / CSE 410: Systems Programming

# Computing Sign Extension

```
11111111 11111111 11111111 10000000
```

Remember that the high 1 bit indicates $-2^{w-1}$, or $-2^{31}$, here.



©2018 Ethan Blanton / CSE 410: Systems Programming

# Computing Sign Extension

```
11111111 11111111 11111111 10000000
```

Remember that the high 1 bit indicates $-2^{w-1}$, or $-2^{31}$, here.

We then add in each of the other bits as positive values.

Every bit from $2^7$ through $2^{30}$ is set, and $2^0$ through $2^6$ are unset:

$$-2^{31} + 2^{30} + 2^{29} + \ldots + 2^8 + 2^7$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# Computing Sign Extension

```
11111111 11111111 11111111 10000000
```

Remember that the high 1 bit indicates $-2^{w-1}$, or $-2^{31}$, here.

We then add in each of the other bits as positive values.

Every bit from $2^7$ through $2^{30}$ is set, and $2^0$ through $2^6$ are unset:

$$-2^{31} + 2^{30} + 2^{29} + \ldots + 2^8 + 2^7$$

…this sums to -128!


©2018 Ethan Blanton / CSE 410: Systems Programming

# What is “Floating Point”?

A floating point number, such as a `float` or `double`, is a number with a variable number of digits before or after the decimal point

(On computers, a variable number of bits before or after the binary point!)

Examples:
3.14159
$6.022 \times 10^{23}$
$6.626 \times 10^{-34}$


©2018 Ethan Blanton / CSE 410: Systems Programming

# What is “Floating Point”?

A floating point number, such as a `float` or `double`, is a number with a variable number of digits before or after the decimal point

(On computers, a variable number of bits before or after the binary point!)

Examples:
3.14159
$6.022 \times 10^{23}$
$6.626 \times 10^{-34}$

It would take nearly 200 bits to represent all three of these numbers precisely.


©2018 Ethan Blanton / CSE 410: Systems Programming

# What is "Floating Point"?

In order to represent numbers of very small or very large magnitude, floating point allows the point to move.

The number of digits of precision is fixed.

Some (loose) terms:

- **Significand**: The meaningful digits of a number
- **Exponent**: The "distance" of those digits from zero in powers of the arithmetic base


©2018 Ethan Blanton / CSE 410: Systems Programming

# Floating Point Representation

In base 10, a floating point number is of the form $x \times 10^y$.

If we consider Avogadro's Number ($6.022 \times 10^{23}$):

- The significand $x$ is 6.022
- The exponent $y$ is 23.

This requires six digits to store, versus 24 digits for 602200000000000000000000.

In base 2, a floating point number is $x \times 2^y$.


©2018 Ethan Blanton / CSE 410: Systems Programming

# IEEE 754 Floating Point

IEEE Standard 754 defines a particular floating point format.

If a floating point number is $x \times 2^y$, in IEEE 754:

- A single precision number (`float`) has a 23-bit $x$ and 8-bit $y$
- A double precision number (`double`) is 52-bit $x$ and 11-bit $y$

Each has a one-bit sign.

# Storing IEEE 754 Components

However, $x$ and $y$ are not stored directly!

$x$ (the significand) is stored:

- Normalized to a value right of the binary point
- With an assumed leading 1 preceding the binary point

This means that a stored significand of 0 is $x = 1.0$

$y$ (the exponent) is stored as $y + 127$.

This means that an exponent of 0 is stored as 127.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Examining Floats

```
float f1 = 2.0f;
float f2 = 0.2f;

dump_mem(&f1, sizeof(f1));
dump_mem(&f2, sizeof(f2));
```


©2018 Ethan Blanton / CSE 410: Systems Programming

# Examining Floats

```
float f1 = 2.0f;
float f2 = 0.2f;

dump_mem(&f1, sizeof(f1));
dump_mem(&f2, sizeof(f2));
```

Output:

```
00 00 00 40
cd cc 4c 3e
```



©2018 Ethan Blanton / CSE 410: Systems Programming

# Deconstructing 2.0

Why is `2.0f` `0x40000000`?

0 10000000 0000000 00000000 00000000

Remembering our significand and exponent storage rules, this means:
$x = 1.0$
$y = 1$
Thus: $1.0 \times 2^1 = 2.0$

(We didn't use 1.0 because it's kind of a special case.)


©2018 Ethan Blanton / CSE 410: Systems Programming

# Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?


©2018 Ethan Blanton / CSE 410: Systems Programming

# Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

What just happened?


©2018 Ethan Blanton / CSE 410: Systems Programming

# Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

What just happened?

The significand isn't decimal!

It's after the binary point.

Fractions cleanly represented in decimal, like $^1/_5$, may not be clean in binary — sort of like $^1/_3$ in decimal.


©2018 Ethan Blanton / CSE 410: Systems Programming

# The Binary Point

Suppose we have a $b$-bit binary number with bits both before and after the binary point, such that:

- There are $w$ whole-number bits before the binary point
- There are $f$ fractional bits after the binary point
- The largest bit before the point is $b_{w-1}$
- The smallest bit before the point is $b_0$
- The largest bit after the point is $b_{-1}$
- The smallest bit after the point is $b_{-f}$


©2018 Ethan Blanton / CSE 410: Systems Programming

# A $w.b$-bit Binary Number

The $w$ whole-number bits are defined as in integers:

$$b_i, i \geq 0 \doteq b_i \cdot 2^i$$

The $f$ fractional-number bits are defined as follows:

$$b_i, i < 0 \doteq b_i \cdot 2^{-b_i}$$

Thus, its total value is:

$$\sum_{i=0}^{w-1} b_i \cdot 2^i + \sum_{j=1}^{f} b_i \cdot 2^{-j}$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# An Example Binary-Point Computation

Consider 11.101b:

$$
\begin{aligned}
11.101b &= 1 \cdot -2^2 + 1 \cdot 2^1 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\
&= 2 + 1 + {}^1/_2 + 0 + {}^1/_8 \\
&= 3\,{}^5/_8 \\
&= 3.625
\end{aligned}
$$


©2018 Ethan Blanton / CSE 410: Systems Programming

# More Floating Point

IEEE 754 is more complicated than we covered here.
(You'll read more about it in the text.)

We have covered the big ideas, however.

Some important implications to consider:

- Very large (either positive or negative) floating point numbers become imprecise because of that $\times 2^y$ factor.
- Very small (close to zero) floating point numbers become imprecise for the same reason.
- Double precision numbers can still be quite large and precise!
- The possible floating point values are unevenly spaced.[3]

[3] See "Denormalized Values" in your text for a caveat.


©2018 Ethan Blanton / CSE 410: Systems Programming

# Summary

- Machines use words for memory and register access
- Hexadecimal is convenient for representing words on modern systems
- C structures are C datatypes laid out adjacent in memory
- Word sizes have alignment implications on memory layout
- Integer representation has complications!
- Floating point representations have different complications!


©2018 Ethan Blanton / CSE 410: Systems Programming

# References I

## Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 1: 1.4; Chapter 2: Intro, 2.1-2.4; Chapter 3: Intro, 3.2, 3.3. Pearson, 2016.


©2018 Ethan Blanton / CSE 410: Systems Programming

# License

Copyright 2018 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.