Week 1 - Friday

COMP 2400

Last time

- What did we talk about last time?
- More C basics
- C compilation model
- History of Unix and Linux

Questions?

Project 1

Quotes

ANSI C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

Kernighan and Ritchie

from *The C Programming Language*, 2nd Edition

C Literals

Literals

- By default, every integer is assumed to be a signed **`int`**
- If you want to mark a literal as **`long`**, put an **`L`** or an **`l`** at the end
  - **`long value = 2L;`**
  - Don't use **`l`**, it looks too much like **`1`**
  - There's no way to mark a literal as a **`short`**
- If you want to mark it unsigned, you can use a **`U`** or a **`u`**
  - **`unsigned int x = 500u;`**
- Every value with a decimal point is assumed to be **`double`**
- If you want to mark it as a **`float`**, put an **`f`** or an **`F`** at the end
  - **`float z = 1.0f;`**

Integers in other bases

- You can also write a literal in hexadecimal or octal
- A hexadecimal literal begins with **`0x`**
  - **`int a = 0xDEADBEEF;`**
  - Hexadecimal digits are **`0`** – **`9`** and **`A`** – **`F`** (upper or lower case)
- An octal literal begins with **`0`**
  - **`int b = 0765;`**
  - Octal digits are **`0`** – **`7`**
  - Be careful not to prepend other numbers with **`0`**, because they will be in octal!
- Remember, this changes only how you write the literal, not how it's stored in the computer
- Can't write binary literals

Printing in other bases

- The **`printf()`** function provides flags for printing out integers in:
  - **`%d`** Decimal
  - **`%x`** Hexadecimal (**`%X`** will print **`A-F`** in uppercase)
  - **`%o`** Octal

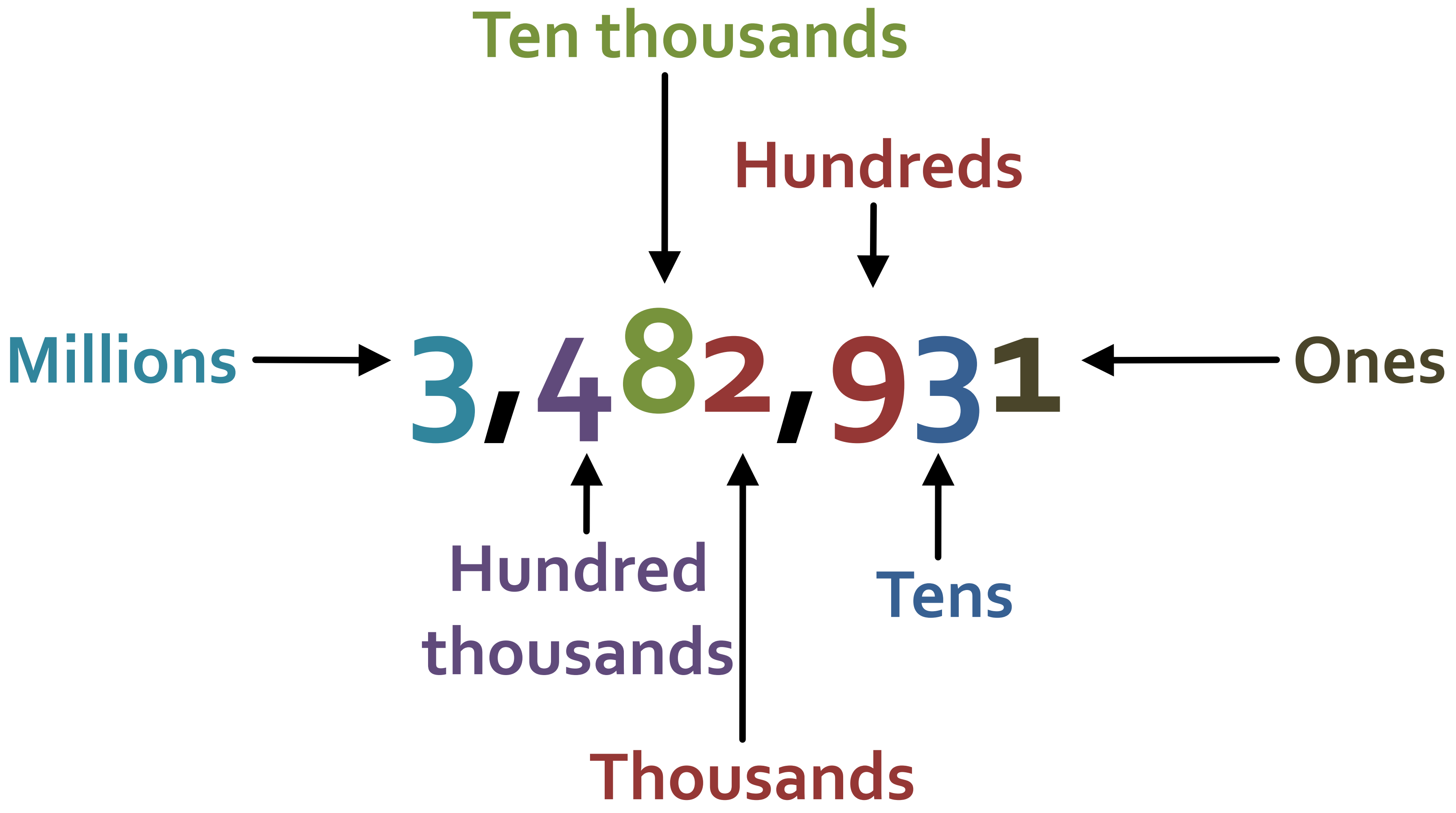
```
printf("%d", 1050); // Prints 1050
printf("%x", 1050); // Prints 41a
printf("%X", 1050); // Prints 41A
printf("%o", 1050); // Prints 2032
```

Data Representation

Base 10 (decimal) numbers

- Our normal number system is base 10
- This means that our digits are: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9
- Base 10 means that you need 2 digits to represent ten, namely 1 and 0
- Each place in the number as you move left corresponds to an increase by a factor of 10

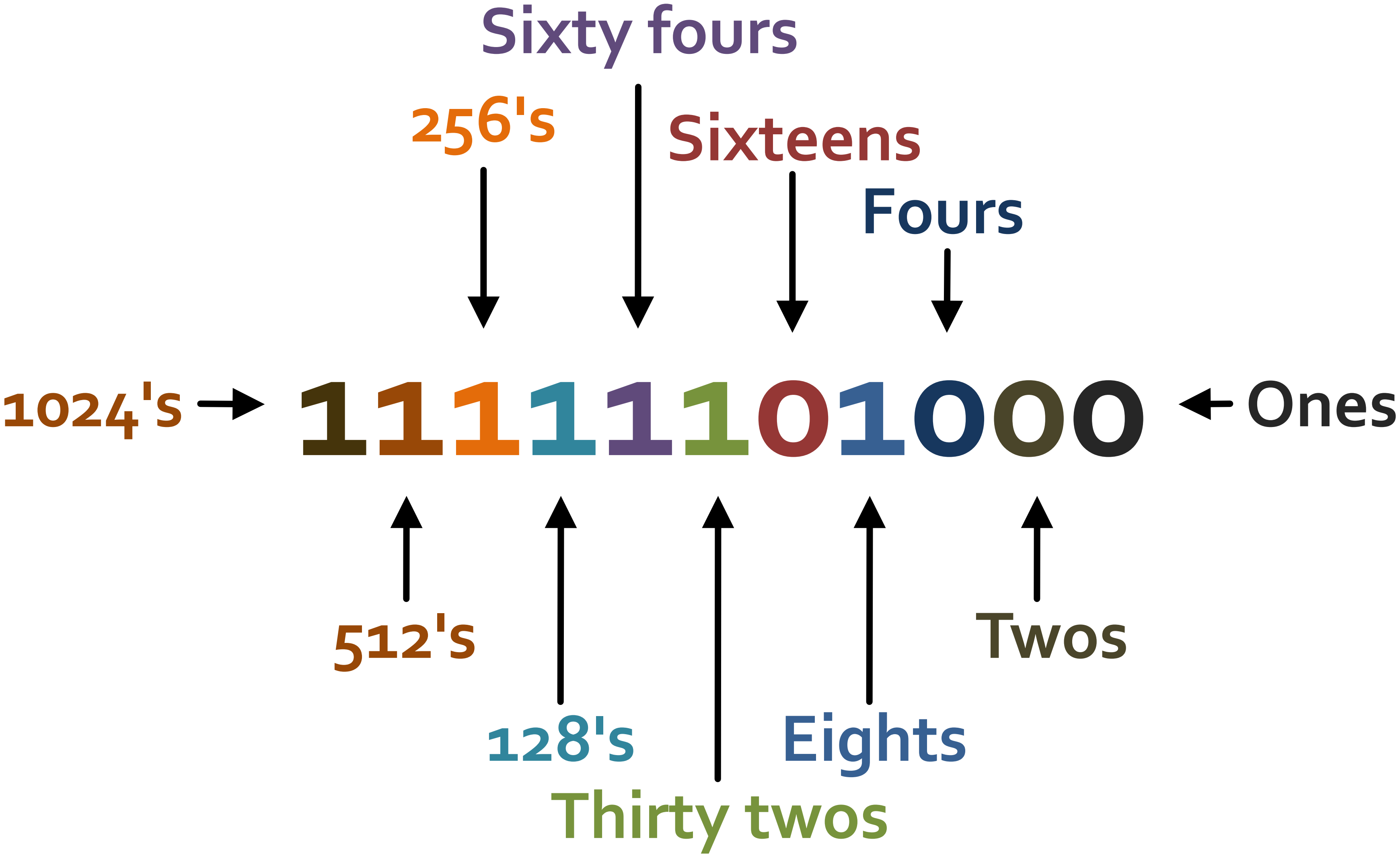
Base 10 Example



Base 2 (binary) numbers

- The binary number system is base 2
- This means that its digits are: **0** and **1**
- Base 2 means that you need 2 digits to represent two, namely 1 and 0
- Each place in the number as you move left corresponds to an increase by a factor of **2** instead of **10**

Base 2 Example



Binary representation

- This system works fine for unsigned integer values
  - However many bits you've got, take the pattern of 1's and 0's and convert to decimal
- What about signed integers that are negative?
  - Most modern hardware (and consequently C and Java) use **two's complement** representation

Two's complement

- Two's complement only makes sense for a representation with a fixed number of bits
  - But we can use it for **any** fixed number
- If the **most significant bit** (**MSB**) is a 1, the number is negative
  - Otherwise, it's positive
- Unfortunately, it's *not* as simple as flipping the MSB to change signs

Negative integer in two's complement

- Let's say you have a positive number ***n*** and want the representation of –***n*** in two's complement with ***k*** bits

1. Figure out the pattern of ***k*** 0's and 1's for ***n***
2. Flip every single bit in that pattern (changing all 0's to 1's and all 1's to 0's)
   - This is called one's complement
3. Then, add 1 to the final representation as if it were positive, carrying the value if needed

Example

- For simplicity, let's use 4-bit, two's complement
- Find -6

1. 6 is **`0110`**
2. Flipped is **`1001`**
3. Adding 1 gives **`1010`**

Two's complement to negative integer

- Let's say you have a ***k*** bits representation of a negative number and want to know what it is

1. Subtract 1 from the representation, borrowing if needed
2. Flip every single bit in that pattern (changing all 0's to 1's and all 1's to 0's)
3. Determine the final integer value

Example

- For simplicity, let's use 4-bit, two's complement
- Given **`1110`**

1. Subtracting 1 **`1101`**
2. Flipped is **`0010`**
3. Which is 2, meaning that the value is -2

All four bit numbers

Binary	Decimal	Binary	Decimal
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

But why?!

- Using the flipping system makes it so that adding negative and positive numbers can be done without any conversion
  - Example 5 + -3 = **0101** + **1101** = **0010** = 2
  - Overflow doesn't matter
- Two's complement (adding the 1 to the representation) is needed for this to work
  - It preserves parity for negative numbers
  - It keeps us with a single representation for zero
  - We end up with one extra negative number than positive number

Floating point representation

- Okay, how do we represent floating point numbers?
- A completely different system!
  - IEEE-754 standard
  - One bit is the sign bit
  - Then some bits are for the exponent (8 bits for float, 11 bits for double)
  - Then some bits are for the mantissa (23 bits for float, 52 bits for double)

More complexity

- They want floating point values to be unique
- So, the mantissa leaves off the first 1
- To allow for positive and negative exponents, you subtract 127 (for **`float`**, or 1023 for **`double`**) from the written exponent
- The final number is:
  - $(-1)^{\textit{sign bit}} \times 2^{(\textit{exponent} - 127)} \times 1.\textit{mantissa}$

Except even that isn't enough!

- How would you represent zero?
  - If all the bits are zero, the number is 0.0
- There are other special cases
  - If every bit of the exponent is set (but all of the mantissa is zeroes), the value is positive or negative infinity
  - If every bit of the exponent is set (and some of the mantissa bits are set), the value is positive or negative NaN (not a number)

Number	Representation
`0.0`	`0x00000000`
`1.0`	`0x3F800000`
`0.5`	`0x3F000000`
`3.0`	`0x40400000`
+Infinity	`0x7F800000`
-Infinity	`0xFF800000`
+NaN	`0x7FC00000` and others

One little endian

- For both integers and floating-point values, the **most significant bit** determines the sign
  - But is that bit on the rightmost side or the leftmost side?
  - What does left or right even mean inside a computer?
- The property is the **endianness** of a computer
- Some computers store the most significant bit first in the representation of a number
  - These are called **big-endian** machines
- Others store the least significant bit first
  - These are called **little-endian** machines

Why does it matter?

- Usually, it doesn't!
- It's all internally consistent
  - C uses the appropriate endianness of the machine
- With pointers, you can look at each byte inside of an **`int`** (or other type) in order
  - When doing that, endianness affects the byte ordering
- The term is also applied to things outside of memory addresses
- Mixed-endian is rare for memory, but possible in other cases:

Math Library

Math library

Function	Result	Function	Result
`cos(double theta)`	Cosine of **theta**	`exp(double x)`	e^x
`sin(double theta)`	Sine of **theta**	`log(double x)`	Natural logarithm of **x**
`tan(double theta)`	Tangent of **theta**	`log10(double x)`	Common logarithm of **x**
`acos(double x)`	Arc cosine of **x**	`pow(double base, double exponent)`	Raise **base** to power **exponent**
`asin(double x)`	Arc sine of **x**	`sqrt(double x)`	Square root of **x**
`atan(double x)`	Arc tangent of **x**	`ceil(double x)`	Round up value of x
`atan2(double y, double x)`	Arc tangent of **y/x**	`floor(double x)`	Round down value of **x**
`fabs(double x)`	Absolute value of **x**	`fmod(double value, double divisor)`	Remainder of dividing **value** by **divisor**

Math library in action

- You must **`#include <math.h>`** to use math functions

```
#include <math.h>
#include <stdio.h>

int main()
{
	double a = 3.0;
	double b = 4.0;
	double c = sqrt(a*a + b*b);
	printf("Hypotenuse: %f\n", c);
	return 0;
}
```

It doesn't work!

- Just using **`#include`** gives the headers for math functions, not the actual code
- You must link the math library with flag **`-lm`**

```
> gcc hypotenuse.c -o hypotenuse -lm
```

- Now, how are you supposed to know that?

```
> man 3 sqrt
```

My main man

- Man (manual) pages give you more information about commands and functions, in 8 areas:
  1. General commands
  2. System calls
  3. Library functions (C library, especially)
  4. Special files and devices
  5. File formats
  6. Miscellaneous stuff
  7. System administration
- Try by typing **`man topic`** for something you're interested in
- If it lists topics in different sections, specify the section

```
> man 3 sqrt
```

- For more information:

```
> man man
```

Upcoming

Next time…

- Preprocessor directives
- Single character I/O

Reminders

- Keep reading K&R Chapter 1
- Keep working on Project 1
- **No class Monday!**