More On Memory (mis)-Management

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Reminder on Strings...

- Reminder: Strings are just like any other C array...
  - You have a pointer to the start and no way of knowing the length
  - But you have an in-band "end of string" signal with the '\0' (0-byte) character
- Since you can have multiple pointers point to the same thing...
  - `char *a, *b;` ...
    `a = b;` ...
    `b[4] = 'x';` /* This will update a as well, since they are pointing to the same thing */
- So how do you copy a string?
  - Find the length (strlen), allocate a new array, and then call strcpy...
  - `a = malloc(sizeof(char) * (strlen(b) + 1) );`
    /* Forget the +1 at your own peril */
  - `strcpy(a, b)` or `strncpy(a, b, strlen(b) + 1);`
    - `strcpy` doesn't know the length of the destination, so it can be very unsafe
    - `strncpy` copies only n character for safety, but if its too short it ***will not copy the null terminator!***

And Constant Strings…

- Anything you put explicitly in quotes becomes a ***constant*** string
  - `char *foo = "this is a constant";`
- For efficiency, these strings are stored as ***read only*** global variables
  - So if you also have `char *bar = "this is a constant";` it is the same string
- It is, guess what, undefined behavior to write to a constant string
  - But fortunately it is usually an immediate crash.

String & Character Functions

- getc/getchar
  - Read single characters... Note return type!
- gets/fgets
  - Read strings up to a linefeed...
  - Note danger of gets(): it will write however much it wants to!
- printf/fprintf
  - Formatted printing functions
- scanf/fscanf
  - Formatted data input functions: Need to take pointers as argument
  - e.g.

```
int i;
scanf("%i", &i);
```

Pointer Ninjitsu: Pointers to Functions

- You have a function definition
  - `char *foo(char *a, int b){ … }`
- Can create a pointer of that type…
  - `char *(*f)(char *, int);`
    - Declares f as a function taking a char * and an int and returning a char *
- Can assign to it
  - `f = &foo`
    - Create a reference to function foo
- And can then call it...
  - `printf("%s\n", (*f)("cat", 3))`
- Necessary if you want to write generic code in C:
  E.g. a hashtable that can handle pointers of any type

C unions

- We've seen how structs can hold multiple elements addressed by name…
  - But what if you want to hold different types in the same location?
-
```
union fubar {
    int a;
    char *b;
    void c;
} Fubar;
```
- Accessed just like a struct, but…
  -
```
Fubar *f = (Fubar *) malloc(sizeof(union fubar))…
f->a = 1312;
f->b = "baz"
```
- They are actually the same memory! It is just treated differently by the compiler!

How to Use Unions…

- Well, you also have to know what the type is… Because C won't do it for you
- Common pattern
  -
```
enum FieldType {a_type, b_type, c_type};
union bar {
          char *a;
          int b;
          float c;};
struct foo {
    FieldType type;
    union bar data; };
…
   struct foo *f;
…
   switch(f->type){
      case a_type:
            printf("%s\n", f->data.a); break;
```

Structure Layout In Memory

- Everything in C is just buckets o bytes…
  - So how do we do structures? We lay out the structure starting at the 0th byte
-
```
struct foo {
    int a;
    char b;
    short c;
    char *d};
```
- It depends on the compiler and underlying architecture…

Alignment, Packing, & Structures…

- If the architecture did not ***not*** force alignment:
  - Just squish everything together (Sometimes seen on old exams)
  -
```
struct foo {
    int a; /* At 0 */
    char b; /* At 4 */
    short c; /* At 5 */
    char *d; /* At 7 */
    char e;}; /* At 11 */
```
- But we already mention that computers don't actually like this!
  - They want things aligned

Default Alignment Rules…

- These are the ***default*** alignment rules for the class
  - Centered around a "32b architecture":
    Integers and pointers are 32b values
- char: 1 byte, no alignment needed when stored in memory
- short: 2 bytes, 1/2 world aligned
  - So 0, 2, 4, 6…
- int: 4 bytes, word aligned
- pointers are the same size as ints
- Need to allow multiple instances of the same structure to be aligned!

So with alignment


- ```
  struct foo {
      int a; /* At 0 */
      char b; /* At 4 */
      short c; /* At 6 */
      char *d; /* At 8 */
      char e;}; /* At 13 */
  ```
- For the class we assume ***no reordering of fields***
- But `sizeof(struct foo)` == 16!
  - Need to add padding to the end as well, so that if we allocate two structures at the same time it is always aligned!
  ```

Pointer Ninjitsu: Pointers to arrays of structures


- ```
  typedef struct foo_struct
      { int x;
        char *z;
        char y;} foo;
  ```
  - So how big is a foo?
    - assume an aligned architecture, sizeof(int) == sizeof(void *) == 4:
    - 12... It needs to be padded
- Dynamically allocated a single element:
  - `foo *f = (foo *) malloc(sizeof(foo))`
- Dynamically allocate a 10 entry array of foos:
  - `foo *f = (foo *) malloc(sizeof(foo) * 10);`
  ```

Pointer Ninjitsu Continued: Accessing that array...

- Accessing the 5th element's string pointer:
  - `f[4].z = "fubar"`
    - Assigns the z pointer to point to the static string fubar
      - It is undefined behavior to then do `f[4].z[1] = 'X'`
      - If you want to modify the string pointed to by `z` you are going to have to do a string copy
- What does it look like "under the hood"?
  - The address written to in `f[4].z = "fubar"` is `(f + 4 * 12 + 4)`:
    - Note: This math is the 'under the hood' math: if you actually tried this in C it would not work right! But it is what the compiler produces in the assembly language
    - The 5th element of type `foo` is offset (4*12) from `f`
      - Since we want all elements in the array to have the same alignment this is why we had the padding
    - The field z is offset 4 from the start of a foo object

Pointer Ninjitsu Advanced: How C++ works...

- C++ is "Object Oriented C"
  - AKA "portable PDP8 assembly language with delusions of grandeur"
- C++ objects are C structures with an extra pointer at the beginning
  - The "vtable" pointer:
    Pointing to an array of pointers to functions
- For inherited ("virtual") functions...
  - To call that function, the compiler writes code that follows the vtable, gets the pointer to function, and calls that

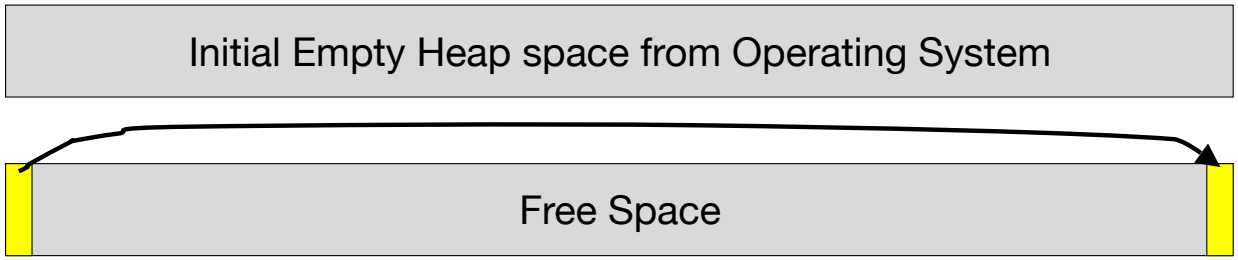
Managing the Heap

- Recall that C supports functions for heap management:
  - **`malloc()`** allocate a block of uninitialized memory
  - **`calloc()`** allocate a block of zeroed memory
  - **`free()`** free previously allocated block of memory
  - **`realloc()`** change size of previously allocated block
    - careful – it might move!

How are Malloc/Free implemented?

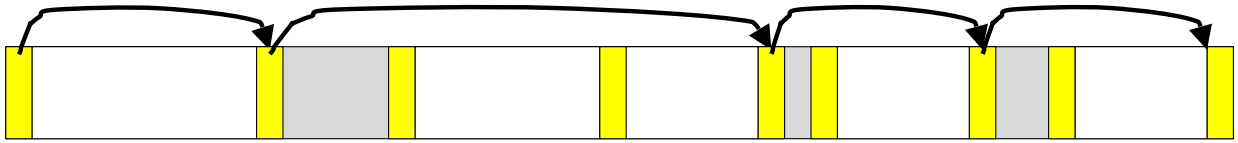
- Underlying operating system allows malloc library to ask for large blocks of memory to use in heap (e.g., using Unix **`sbrk()`** call)
  - This is one reason why your C code, when compiled, is dependent on a particular operating system
- C standard malloc library creates data structure inside unused portions to track free space
  - This class is about how computers work:
    How they allocate memory is a huge component

Simple Slow Malloc Implementation



Malloc library creates linked list of empty blocks (one block initially)

First allocation chews up space from start of free space



After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

The Problem Here: Fragmentation

- That memory heirarchy we saw earlier likes things small…
  - And likes things contiguous
- Things start to work badly when stuff is scattered all over the place
  - Which will eventually happen with such a simple allocator

Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- "Buddy allocators" always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:
  - Then can just use a simple bitmap to know what is free or occupied

Power-of-2 “Buddy Allocator”

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

Malloc Implementations

- All provide the same library interface, but can have radically different implementations
- Uses headers at start of allocated blocks and/or space in unallocated memory to hold **`malloc`**'s internal data structures
- Rely on programmer remembering to **`free`** with same pointer returned by **`malloc`**
  - Alternative is a "conservative garbage collector"
- Rely on programmer ***not messing with internal data structures accidentally***!
  - If you get a crash in **`malloc`**, it means that somewhere else you wrote off the end of an array

Conservative Mark/Sweep Garbage Collectors

- An alternative to **`malloc`** & **`free`**...
  - **`malloc`** works normally, but **`free`** just does nothing
- Instead, it starts with the stack & global variables as the "live" memory
  - But it doesn't know if those variables are pointers, integers, or whatevers...
- So assume that every piece of memory in the starting set is a pointer...
  - If it points to something that was allocated by malloc, that entire allocation is now considered live, and "mark it" as live
  - Iterate until there is no more newly discovered live memory
- Now any block of memory that isn't can be deallocated ("sweep")

The Problems: Fragmentation & Pauses...

- A conservative garbage collector can't move memory around
  - So it gets increasingly fragmented...
    When we get to both caches and virtual memory we will see how this causes problems
- A conservative collector needs to ***stop the program!***
  - What would happen if things changed underneath it? Ruh Roh...
  - So the system needs to pause
- Java and Python don't have this problem
  - Java is designed to understand garbage collection:
    Able to have ***incremental*** collectors that don't require a long halt but only short halts
  - Python doesn't do real garbage collection:
    Just uses "reference counting". Every python object has a counter for the number of pointers pointing to it. When it gets to 0, free the object
    - Reference counter can't free cycles

Clicker Question

- What will the following print:
  - ```
    int a, b, c, *d;
    a = 0;
    b = 1;
    c = 2;
    d = &a;
    (*d) += b + c;
    d = &b;
    (*d) += a + b + c;
    printf("a=%i b=%i\n", a, b);
    ```
  - A) a=0, b=3
  - B) a=3, b=3
  - C) a=3, b=4
  - D) a=3, b=7
  - E) I ditched class today and had a friend "borrow" my clicker
    ```

Administrivia...

- Lab 0 and 1 were due today...
  - Can be checked off next week for 1/2 credit
- Lab 2 released already
  - Additional C practice for the project
- Early labs are better!!!!
  - Far fewer people, so you can get a lot more help!
- Project Party #1: Thursday 8-10pm in 540AB Cory Hall
  - Sorry, I'm not allowed to make it a Project Raging Kegger... 🍺

Common Memory Problems: aka Common "Anti-patterns"

- Using uninitialized values
  - Especially bad to use uninitialized pointers
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
  - Writing to static strings
- Improper use of **`free`**/**`realloc`** by messing with the pointer handle returned by **`malloc`**/**`calloc`**
- Memory leaks (you allocated something you forgot to later free)
- Valgrind is designed to catch ***most*** of these
  - It runs the program extra-super-duper-slow in order to add checks for these problems that C doesn't otherwise do

Using Memory You Don't Own

- What is wrong with this code?

```
int *ipr, *ipw;
void ReadMem() {
 int i, j;
 ipr = (int *)
    malloc(4 *
       sizeof(int));
 i = *(ipr - 1000);
 j = *(ipr + 1000);
free(ipr);}
```

Out of bounds reads

```
void WriteMem() {
   ipw = (int *)
      malloc(5 *
         sizeof(int));
   *(ipw - 1000) = 0;
   *(ipw + 1000) = 0;
   free(ipw);
}
```

Out of bounds writes

Faulty Heap Management

- What is wrong with this code?
-
```
int *pi;

void foo() {
  pi = malloc(8*sizeof(int));
  …
  free(pi);
}

void main(){
  pi = malloc(4*sizeof(int));
  foo();
  … }
```

The first `malloc` of pi leaks

Reflection on Memory Leaks

- Memory leaks are not a problem ***if your program terminates quickly***
  - Memory leaks become a much bigger problem when your program keeps running
  - Or when you are running on a small embedded system
- Three solutions:
  - Be very diligent about making sure you `free` all memory
    - Use a tool that helps you find leaked memory
    - Perhaps implement your own reference counter
  - Use a "Conservative Garbage Collector" **malloc**
  - Just quit and restart your program a lot ("burn down the frat-house")
    - Design your server to crash!
      But memory leaks will ***slow down your program*** long before it actually crashes

So Why Do Memory Leaks Slow Things Down?

- Remember at the start we saw that pyramid of memory?
  - Small & fast -> cache
    Big & slow -> main memory
- Memory leaks lead to ***fragmentation***
  - As a consequence you use more memory, and its more scattered around
- Computers are designed to access ***contiguous*** memory
  - So things that cause your working memory to be spread out more and in smaller pieces slow things down
- There also may be nonlinearities:
  - Fine... Fine... Fine... Hit-A-Brick-Wall!

Memory Leaks & The Project...

- We have a test which ***will*** cause your program to crash if you leak in **`processInput()`**
  - How do we do this? We tell the OS to not give your program very much memory...
- But we won't check for leaks in your dictionary/hashtable
  - After all, you have to have it in memory for the entire program lifetime
- So keep that in mind when running valgrind...
  - "Leaked memory" allocated in **`readDictionary()`** 🤷‍♂️
  - "Leaked memory" allocated in **`processInput()`** 🤔

Faulty Heap Management

- What is wrong with this code?

- ```
  int *plk = NULL;
  void genPLK() {
    plk = malloc(2 * sizeof(int));
    … … …
    plk++;
  }
  ```

This MAY be a memory leak if we don't keep somewhere else a copy of the original malloc'ed pointer
  ```

Faulty Heap Management

- How many things are wrong with this code?

- ```
  void FreeMemX() {
    int fnh[3] = 0;
    ...
    free(fnh);
  }
  ```

  Can't `free` memory allocated on the stack

- ```
  void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    ...
    free(fum);
    ...
    free(fum);
  }
  ```

  Can't `free` memory that isn't the pointer from `malloc`

  Can't `free` memory twice

Using Memory You Haven't Allocated

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```

`sizeof(char)` is 1 but should have `sizeof` as a good habit

Write off of the end of the array!

Using Memory You Don't Own

- What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

Returning a pointer to stack-allocated memory!

Using Memory You Don't Own

- What is wrong with this code?

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

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
      head = head->next;
    }
    return head->val;
}
```

What if head is null?
Always check arguments.
Your code may be good...
But you make mistakes!
PROGRAM DEFENSIVELY

Using Memory You Don't Own

- What is wrong with this code?

```
void muckString(char *str) {
    str[0] = 'b';
}
void main(void) {
    char *str = "abc";
    muckString(str);
    puts(str);
}
```

Pointing to a static string...
Ruh Roh...

So Why Was That A Problem...

- When the compiler sees
  - `char *foo = "abc"`
  - The compiler interprets it as 'have the constant string "abc" somewhere in static memory, and have foo point to this'
    - If you have the same string "abc" elsewhere, it will point to the same thing...
      If you are lucky, the compiler makes sure that these string constants are set so you can't write
      - "Access violation", "bus error", "segfault"
- There is something safe however...
  - `char foo[] = "abc"`
  - The compiler interprets this as 'create a 4 character array on the stack, and initialize it to "abc"'
  - But of course we can't now say `return foo;`
    - Because that would be returning a pointer to something on the stack...

Managing the Heap: `realloc(p, size)`

- Resize a previously allocated block at **`p`** to a new size
- If **`p`** is NULL, then **`realloc`** behaves like **`malloc`**
- If size is 0, then **`realloc`** behaves like **`free`**, deallocating the block from the heap
- Returns new address of the memory block; NOTE: it is likely to have moved!
-
```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check NULL, contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}

int* fill_fibonacci(int *fib, int size) {
   int i;
   init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

Realloc might move the block!

Which means this hasn't updated *fib!

And Now A Bit of Security: Overflow Attacks


- ```
  struct UnitedFlyer{
      ...
      char lastname[16];
      char status[32];
      /* C will almost certainly lay this out in memory
         so they are adjacent */
      ...
  };
  ...
  void updateLastname(char *name, struct UnitedFlyer *f){
      strcpy(f->lastname, name);
  }
  ```
  ```

So what...

- Well, United has my status as:
  - `name = "Weaver", status = "normal-person: hated"`
- So what I need to do is get United to update my name!!!
  - So I provide United with my new name as:
    `Weaver super-elite: actually like`
  - `name = "Weaver super-elite: actually like", status = "super-elite: actually like"`
- And then update my name ***again*** back to just "Weaver"
  - `name = "Weaver", status = "super-elite: actually like"`
- Basic premise of a ***buffer overflow*** attack:
  - An input that overwrites past the end of the buffer and leaves the resulting memory in a state suitable to the attacker's goals

Exploiting Use-After-Free

- Using data after a free is also exploitable...
  - Because the data might get allocated for something else
-
```
name = (char *) malloc(sizeof(char) * 1024)
...
free(name);
...
foo = (char *) malloc(sizeof(char) * 1024)
scanf("%s\n", foo);
...
if(name == "root")...
```
- Attacker can write to foo whatever they want for name
  - Since both point to the same space, at least with some probability...

And double-free can create use-after-free


- ```
  name = (char *) malloc(sizeof(char) * 1024)
  ...
  free(name);
  ...
  foo = (char *) malloc(sizeof(char) * 1024)
  ...
  free(name)
  ```
- Now you have a use-after-free possibility for `foo`...
  - Since `foo` got freed but you are still using it
  ```

Hijacking Control Flow With Buffer Overflows...

- Reminder: The stack stores a lot of stuff...
  - Not just local variables, but where to return execution to when a function returns
- So if you write some code with a stack-allocated array
  - `char foo[32];`...
    `gets(foo);` /* gets reads input into the string until a newline */
- A bad dude gets to provide your program a string (e.g. because it is a server)
  - His string is significantly longer than 32 characters...
- Result:
  - His string overwrites other local variables, including where the function is supposed to return!
  - And now your program is ***his*** program!

And Valgrind...

- Valgrind slows down your program by an order of magnitude, but...
  - It adds a tons of checks designed to catch most (but not all) memory errors
- Memory leaks
- Misuse of free
- Writing over the end of arrays
- You ***must*** run your program in Valgrind before you ask for debugging help from a TA!
  - Tools like Valgrind are absolutely essential for debugging C code

And In Conclusion, …

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code

More Numbers...
Working Towards Floating Point

- Reminder, a collection of ***n*** bits can represent one of any 2^n "things"
- Our default is "unsigned integer"
  - 0 to 2^n-1
  - Naturally good for representing addresses
- Also like "signed" as 2s-complement
  - -2^{n-1} to $2^{n-1}-1$
- For both of these the math is "easy"
  - Addition ***and subtraction*** are the same for both
    - Subtract by just inverting and adding one...

Some other cool arithmetic tricks

- Does ***x*** == ***y***?
  - Easy test: does ***x - y == 0***?
- Multiply by 2^n?
  - We left shift (<<) (move the bits to the left) by ***n***
- (sorta) divide by 2^n?
  - We right shift (>>) by ***n***
  - For unsigned (logical) shift: Left gets 0s
  - For signed (arithmetic) shift: Left gets the sign bit
    - Not quite right for negative numbers:
      you'd say ***-1/2 = 0***, but in 2s complement ***-1 >> 1 = -1***

But "Any one of 2^n" is whatever we make it to be!

- One alternate representation: Sign/Magnitude
  - Lets have the first bit say the sign (+ or - as 0 or 1)
  - And the rest be unsigned
- Allows us to represent $-2^{n-1}+1$ to $2^{n-1}-1$
- This gives us two zeros (+/- 0)...
- This gives us a cleaner symmetry otherwise
  - Magnitude is consistent for both positive and negative
- But math is more of a pain...
  - So a poor choice if we want to do "simple" math like add and subtract...

Another Alternative Representation: Biased...

- The actual value is the binary value plus a fixed bias
  - So "bias = -127" means the actual number is the binary value with -127 added to it
  - Binary ***00000000*** -> ***-127***
  - Binary ***11111111*** -> ***+128***
- Why do this?
  - Can set our range to be arbitrary
  - No discontinuity around 0
- Disadvantages
  - All bits 0 != 0
  - Math more of a pain: To add A + B...
    - ***A + B - bias*** (To eliminate the extra bias)

Another Representation: Fixed Point...

- Set a decimal point in binary...
  So if we have 4 decimal places...
  - So its say $\boldsymbol{1011.0100_2}$ -> $\boldsymbol{2^3} + \boldsymbol{2^1} + \boldsymbol{2^0} + \boldsymbol{2^{-2}}$ -> $\boldsymbol{10.25_{10}}$
- Nice for math
  - Add and subtract as normal
  - Multiply/divide just shift to readjust the decimal point
- Are some oddities
  - Can't represent big numbers!
  - Can't ***exactly*** represent some interesting numbers: $\boldsymbol{0.1_{10}}$ has no exact representation in fixed point binary!

This Is All Leading Up To Floating Point....

- We want to be able to represent very small numbers...
  - EG, the size of an atom
- We want to be able to represent very large numbers...
  - EG, the size of the galaxy
- We want consistent precision...
- We want to degrade well...
  - Both as we go towards infinity and shrink down to zero...
- So think about the problem, and we will get to the solution next time...