

CS 0449 Notes:

C-Basics:

Hello, World:

Splashhhhh 💧

```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

1) What are the similarities?

- Syntax looks similar (**braces, semicolons, keywords**)
- We use some kind of "print" function to write to the console
- We have "int"

2) What things are different?

- There's **#include**, which probably works something like import
- **main** returns a number for some reason
- There's no **String[]** args on main
- There's no **class**

Falling Down the Rabbit Hole:

1) If you compile that C program, you get an executable file

- what does that executable file *contain*?
- what *happens* when you run it?
- *how* does the program actually make things happen?
- *how* do programs control all the computer hardware?
- how do *multiple programs* run at the same time?

2) That's what this course is all about.

Oh yeah, **gcc**:

1) **gcc** is the C compiler we'll be using (the C analogue of `javac`)

2) To compile *name.c* to a program called *name*:

```
gcc -o name name.c
```

3) If you just do `gcc name.c` the output will be named `a.out`

4) You might see me type:

```
gcc -Wall -Werror --std=c99 -o name name.c
```

5) The up/down arrows on your keyboard go through recent commands!

Remarks:

-Wall and **-Werror** are good habits – catch lots of errors which would otherwise go unnoticed

Oh yeah, **thoth**:

1) **thoth** is a server we've set up for you to do your work.

– You connect over the internet with **ssh** to run commands on it

– Your files are stored in a third place called **AFS**, which is universal to pitt (and in fact many institutions)



C-Basics:

Appearances Can Be Deceiving:

- 1) C might *look* a bit like Java
- But it sure doesn't *behave* the same way

```
int x = 5;  
printf("The value of x is: " + x);
```

- 2) What does this print?
- It prints "alue of x is: "
- If you change x, it changes where in string it starts printing, but never prints x



The Training Wheels are Off:

- 1) C is weird and prickly and unforgiving and like 50 years old
- 2) You will probably hate it
- 3) But like this flower and the weird bird that matches it, C and modern computers have *coevolved*
- 4) **Understanding C is understanding how modern computers work***



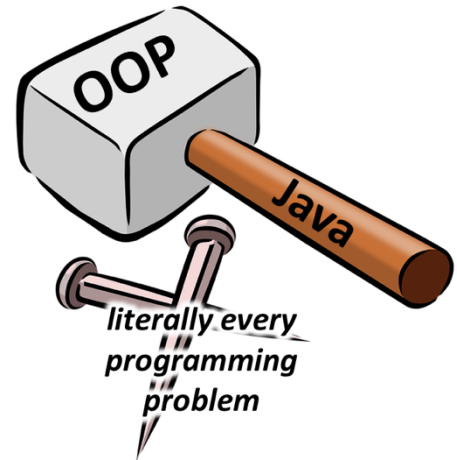
Which C?

- 1) C89 (**ANSI C**)
- Most portable, but annoying
- 2) C11
- New features not relevant for us here
- 3) We'll stick with C99 for this course.

```
gcc --std=c99 ...
```

OOPs:

- 1) Java is OOP: "**object-oriented programming**"
 - that's fine, but...
- 2) If someone tells you that **their way of thinking** will solve all your problems, *they're lying*.
- 3) Part of being an engineer is knowing what tool is **appropriate for the situation**
 - OOP is *not* always that tool
 - C is not always that tool either!



Remarks:

- 1) This is general life advice too
- 2) People had OOP fever in the 90s and 00s, but we've come to realize that many of its features are more trouble than benefit
- 3) Single-paradigm languages are annoying

Input and Output:

It's a lot Shorter Than `System.out.println()`...

- 1) Instead of using `+` like in Java...
- 2) We use a format string with format specifiers

```
int x = 5;  
printf("The value of x is: %d\n", x);
```

Specifier	Type
%d	int
%f	float
%s	string
%c	char

Escape	What it is
\n	newline
\t	tab
\\	\
\a	DING!!!

Remarks:

- You can absolutely use the "wrong" specifier
- If you have `-Wall` on, it will complain, but **otherwise**, it will compile and run and do weird stuff

One Big Difference with Variables in C:

```
int x;
```

```
printf("The value of x is: %d\n", x);
```

1) What will it print?

- Answer: **you don't know.**

2) **Uninitialized variables in C can contain any value.**

- Maybe 0!

- Maybe 2385!

- Maybe -100000!

```
use gcc -Wall -Werror
```

-Wall catches so many things that are absolutely errors, it's just that C lets you do virtually anything.

The Beginning of Weirdness:

1) How do you get a line of text input in Java with a **Scanner**?

```
String input = someScanner.nextLine();
```

2) It's... a **little different** in C

```
char input[100];
```

```
fgets(input, 100, stdin);
```

A Local Array:

1) Here's something **that doesn't exist** at all in Java:

```
char input[100];
```

2) It's an array of 100 chars, but...

- It works like a **local variable**

- It **disappears** when this function exits!

⇒ This can be dangerous...

⇒ because you can hand off that array to someone else, and then it can disappear, and then who knows what will happen when they use it

3) You will see **local arrays** used often in C

f'getsaboudit:

1) Then we have this function call:

fgets(**input**, **100**, **stdin**);

2) **fgets** = file **get** s string

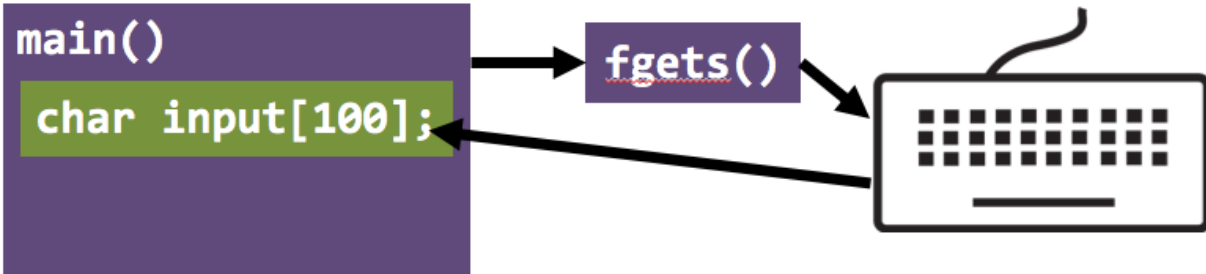
- That is, get a string (a line) from a file

3) **stdin** is C's name for what Java calls **System.in**

4) Here we see another common C pattern:

- Rather than **fgets** giving us an array...

- We ask **fgets** to fill in an array that we created



Remarks:

1) It's like **buying a bottle of water** vs. **carrying your own reusable one**

2) Java is like **buying a bottle of water**

3) This is like **carrying your own**

4) But you have to **watch out who you give your bottle to**, and **make sure they don't overfill it!!**

C-Functions:

Solving the **fgets** Problem:

Last Time...

1) We used this code to get a line of text from the user:

```
char input[100];  
fgets(input, 100, stdin);
```

2) But there was a problem with **1_fgets.c**...

Hello, Jarrett
!

3) Do you remember the escape characters from last time?

- One of the escape characters, `\n`, makes a newline...

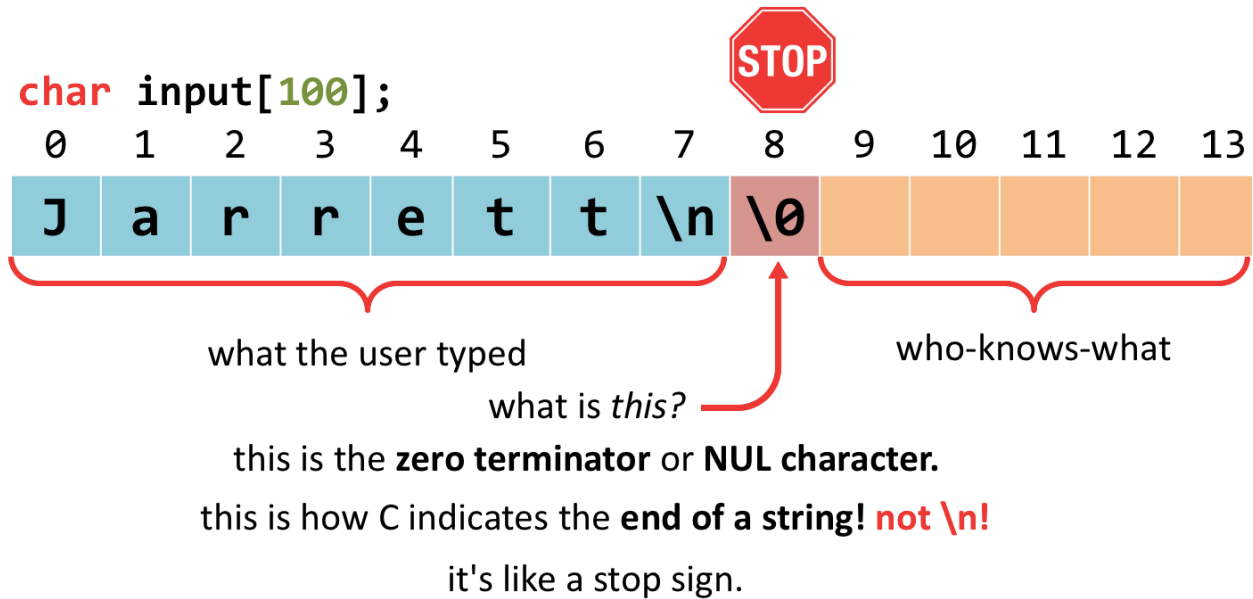
4) Well, what's the last key you press after you type your name?

- **enter/return** makes the **newline** character!

- **fgets** is getting the line of text, **INCLUDING** the **newline** at the end

What the String Looks Like:

1) C Strings are weird...



Remarks:

- 1) Clearly the array slots go up to 99 but not enough room here :P
- 2) `\0` is the escape sequence for the **NUL character**
- 3) Usually you don't have to write it, cause **C** will put it in for you in "string literals" and the **stdlib** functions do it too
- 4) Every character is really just a number – and `\0` is **0**


Truncating (Shortening) a String:

1) If we want to remove one or more characters from the end...

0	1	2	3	4	5	6	7	8	9	10	11	12	13
J	a	r	r	e	t	t	\n	\0					


2) You can put a zero terminator anywhere you want.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
J	a	r	r	\0	t	t	\n	\0					


`input[4] = '\0';`

3) Where should we put it if we want to get rid of the **newline character**?

0	1	2	3	4	5	6	7	8	9	10	11	12	13
J	a	r	r	e	t	t	\0	\0					


where should we put it if we want to
get rid of the newline character?

let's put it at **(length of string) - 1**.

Remarks:

- 1) You can change the characters in a string, since **it's just an array**
- 2) Single quotes are **character literals** – a single character value, like `\0`
- 3) You could also write `"input[4] = 0;"`
- 4) The **characters are still in the array** – just "no longer part of the string"

Getting the Length of a String:

1) Do you remember how Java does it?

- `someString.length()`;

2) In C, you have to count the characters until the terminator.

3) `strlen()` from `string.h` does this.

```
char input[100];  
fgets(input, 100, stdin);  
int len = strlen(input);  
input[len - 1] = '\0';
```

} all of this is what we need to read a line of text from the console.

If you wanted to read another line, would you **copy and paste** this?

NNNNOOOOOOOOOOOO

Remarks:

- **C strings are arrays**, and we saw there can be fewer characters in the string (before the terminator) than in the array

- `strlen()` is an $O(n)$ time operation – don't use it inside loops if you can avoid it

Functions:

What's a Function?

- 1) A named piece of code with inputs (parameters) and outputs (return values, side effects)
- 2) A *really useful problem-solving tool*

```
fgets(input, 100, stdin);  
int len = strlen(input);  
input[len - 1] = '\0';
```

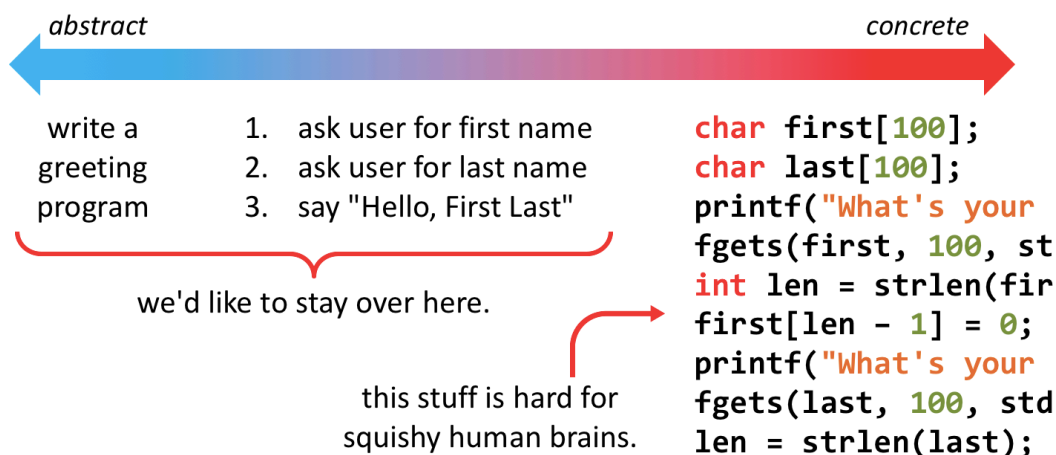
```
get_line(input, 100);
```

Remarks:

- 1) Java's static methods are functions!
- 2) The shorter code is more readable, and its purpose and effect are more identifiable

Abstraction: What to do vs. How to do it

- 1) **Abstraction** is about hiding details: focusing more on the *what* and the *why* and less on the *how*



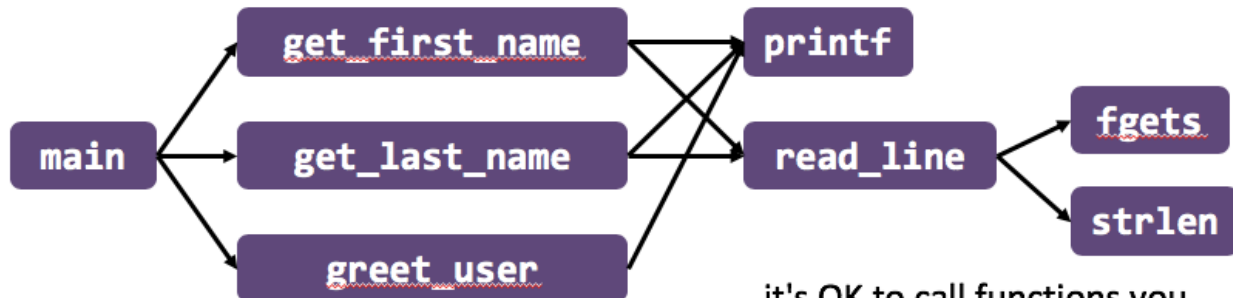
Remarks:

- abstraction is how you will tackle the rising complexity of the problems we ask you to solve
- abstraction also helps you solve problems even if you don't know all the details on how to solve them yet
- remember: programming languages are there for **YOUR** benefit, not the computer's
- **MAKE YOUR LIFE EASIER FOR YOURSELF**

Call Graphs:

1) One way to structure a program is **top-down**:

- Start with the **most abstract**, and split it up over and over



maybe we don't even know how to do these yet, but we have the idea down.

it's OK to call functions you haven't written yet!

Remarks:

- A **call graph** shows what functions call what other functions
- Having lots of arrows going ***into*** a function is **good** – means you are **REUSING CODE** and **REDUCING EFFORT**
- Having lots of arrows coming ***out of*** a function is **not good** – means you are **REPEATING YOURSELF** and **CLOUDING THE MEANING**

Too Much Code:

- 1) Think of a function as a scene in a play, show, movie:
 - 2) Every line of code is like a new actor coming out and saying a line
- Just like a scene with too many actors and lines and actions is hard to understand...
 - ...so, too, is a function with too many lines, and too many complicated lines



Naming:

- 1) Code is there for *your* benefit
- 2) Make it as easy as possible for you to read!

```
int temp = calculate(x1, x2);  
float temp2 = temp * 3.333333;  
doStuff(temp2, 9.807);
```

never use "temp" as a variable name.

name your functions according to **what they do, and to what.**

avoid **hard-coding important numbers. use constants.**

you can use camelCase or snake_case – whatever you like

Remarks:

- 1) Two hard problems in CS: **recursion**, **naming**, and **off-by-one errors**
 - 2) **NAMES INDICATE INTENT AND PURPOSE.**
 - 3) if something really is **temporary**, name it "**tempWhatever**" to indicate its **PURPOSE.**
- use "**verb noun**" for function names

Functions in C:

Let's Make a Better **fgets**:

1) So, we have DECIDED to write a function for this, right? ;)

```
char input[100];  
fgets(input, 100, stdin);  
int len = strlen(input);  
input[len - 1] = '\0';
```

2) Some issues:

- How do you know how long to make the array?
- Can you even return an array in C? (no)
- If we can't return an array, how do you take one as an argument?

3) Well let's start with how we decided we want to *call* it:

```
get_line(input, 100);
```

Remarks:

- 1) User could want to get lines of different lengths
- 2) Local arrays are like local variables, and disappear when the function returns
- 3) Let's go top-down again: start with the ***interface to the function*** and then figure out the details

The Function Signature:

- 1) We don't need it to return anything, so its return type is...?
 - it returns **void** (just like Java)
- 2) The first parameter is an array, and the second is an int, so...

```
void get_line(char[] input, int size)
```

- 3) Nope, doesn't compile...

```
void get_line(char input[100], int size)
```

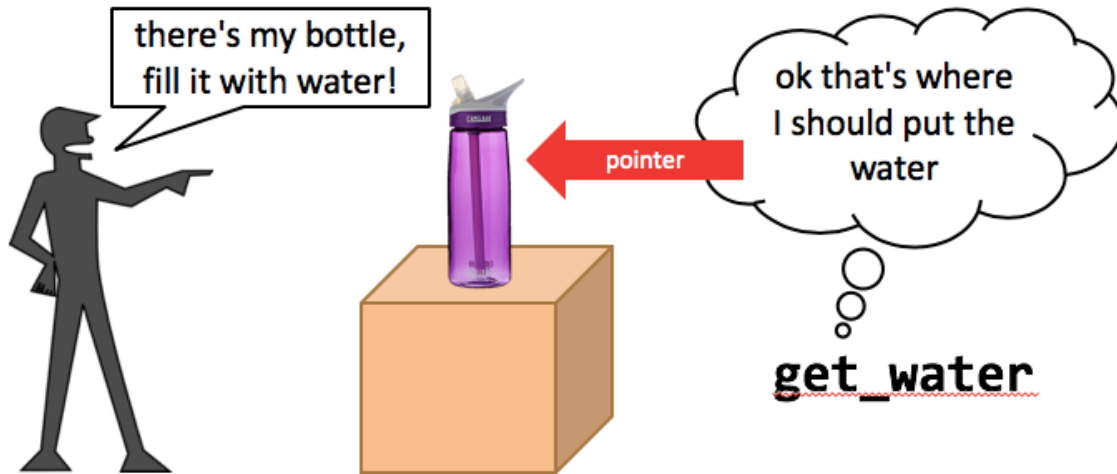
- 4) Huh, that compiles, but...
 - This doesn't do what you expect.
- 5) Here's something weird about arrays in C:
 - **They aren't real.**

Remarks:

- 1) How would you write an array parameter in **Java**? (**char**[] input)
- 2) The second signature is **NOT** how we usually write it in **C**.

Arrays aren't Real??

- 1) C doesn't treat arrays as "objects" the way Java does
 - 2) Instead, C uses *pointers*
 - 3) A pointer is a variable which holds a **memory address**
- We can access the data *through* the pointer



Remarks:

- 1) The array doesn't "know" how long it is
- 2) Pointers are a bit like Java references, but **lower level**
- 3) The pointer just specifies ***where*** something is, it doesn't specify how ***big*** it is

Arrays Become Pointers When Passed Into Functions:

- 1) and pointers are written like this:

```
void get_line(char* input, int size)
```

- 2) Input is not a char, it's a *pointer to a char*
- The docs for `fgets` will show the same thing:

```
char* fgets(char* str, int num, FILE* stream)
```

Remarks:

- 1) The pointer says "**here is a place where you can put one or more chars**"
- 2) any time you would write "**sometype[] blah**" as a parameter in Java, you write it "**sometype* blah**" in C
- 3) Why does this happen?

An array can be **HUGE** – thousands or millions of items long. Far easier to say "there it is" than try to move it around

Getting to the Point:

1) Here's how we'd write our `get_line` function:

```
void get_line(char* input, int size) {  
    fgets(input, size, stdin);  
    int len = strlen(input);  
    input[len - 1] = '\0';  
}
```

2) and NOW we can use it like we want

Something Silly:

1) Let's try moving `get_line` *after* main:

```
warning: implicit declaration of function 'get_line'  
warning: conflicting types for 'get_line'  
previous implicit declaration of 'get_line' was here
```

2) It still compiles and runs, but... what.

3) *C does not know about functions if you try to access them before you declare them.*

- If you do, it assumes they have the signature `int name()`

⇒ Which is patently ridiculous but hey it made sense in 1971

Remarks:

1) The earliest C-like language compiler ran on a machine with 4KB of RAM available

2) It could not read the entire program in at once; it could only read one line at a time, and compile it right there

3) It could only "remember" things it had seen

Function Prototypes:

- 1) *if* you want to access a function *before* you declare it...
- 2) You can use a prototype:

```
void get_line(char* input, int size);  
...functions which use get_line...  
void get_line(char* input, int size) {  
    blah blah!
```

- 3) It's the function signature, but with a **semicolon** instead of code.
- 4) Or, you can just **reorder** your functions :^)

Remarks:

- Reordering functions within a file is generally cleaner but prototypes are sometimes unavoidable (**mutually recursive functions**)
- But usually, we put all the prototypes in a separate "header file"

Returning Arrays?

Would be Nice...

- 1) Couldn't we do it in Java, like:

```
char* name = get_line(10);
```

- 2) Well... yes and no, but **DEFINITELY** not with a local array
 - 3) Returning an array like this is **so bad** that GCC will actually force your function to return null if you try to do it directly
- So **never**
 - ⇒ **ever**
 - **do it.**

C-Strings and Files:

Strings in C:

What do you get when you mix...

- 1) ...arbitrary-length strings...
- 2) ...whose length is only indicated by a special value...
- 3) ...where forgetting that value leads to accesses past the end...
- 4) ...in a language that has no array bounds checking?
 - *garbage*
 - I *cannot* overstress how terrible string manipulation is in C
- 5) strings are *so bad* in C that some people embed a Lua scripting language interpreter in their programs *just to have string manipulation that isn't prone to massive security holes*

Remarks:

- Like seriously, **AVOID DOING HEAVY STRING MANIPULATION IN C.**
- It is **NOT** the right tool for that job.
- Find a language that does it better. (that is, almost any other language in common use today)

Why Do Banks Only Allow 12-Character Passwords?

1) The problem is multi-faceted:

```
char str[10] = "Hi people";
```

0	1	2	3	4	5	6	7	8	9
H	i		p	e	o	p	l	e	X

2) If we try accessing past the end:

```
char str[10] = "Hi people";
```

0	1	2	3	4	5	6	7	8	9
H	i		p	e	o	p	l	e	\0

there's only room for 9
characters + \0

@
but *nothing* stops us from
accessing past the end
`str[10] = '@';`

3) If we lose the **NULL** terminator:

```
char str[10] = "Hi people";
```

0	1	2	3	4	5	6	7	8	9
H	i		p	e	o	p	l	e	X

there's only room for 9
characters + \0

@
but *nothing* stops us from
accessing past the end
`str[10] = '@';`

if this \0 is lost somehow, we have no
idea how long the string is

```
str[9] = 'X';
```

if we want a longer string, we just don't
have the room, so where do we put it??

- When there is **no zero terminator**, we will march right off the end of the array
and into unknown territory...

The Great Unknown:

- 1) You **cannot know in advance** how long an input string will be
 - From the user
 - From a file
 - From the network etc.
- 2) There are practical limits but in general, you have no idea
- 3) So, how much space do you need?
 - How do you allocate "**enough**" space?
 - ⇒ What do you do if you run out?
 - Why is this so hard??

Initializing an Array of Characters:

- 1) The easy way is like so:

```
char mystr[100] = "some string";
```

- 2) This:

- Allocates space for 100 characters
- Fills the first *however many* with that string
- Fills the rest with **\0** characters

- 3) **If you do this:**

```
char* mystr = "some string";
```

- 4) It is **totally different**. it:

- Puts "some string" in the *static data segment*
- Allocates space for a pointer
- Makes that pointer point to the static data segment
- **Don't use this if you want to manipulate the string.**

- 5) Every string argument to a function is gonna be a **char*** so that it can point to a string *anywhere*.

Some String Functions:

- 1) The `<string.h>` header contains many string functions
- 2) `strlen()` ... we saw last time!
 - it has to look through the whole string for `\0` every time
 - So, do *not* call it inside a loop if you can avoid it.
- 3) `strcmp()` is another common one
 - It compares two strings and returns a *comparison value*

if <code>strcmp(a, b)</code> returns...	then...
<0	a comes before b
>0	a comes after b
0	a is equal to b

this is easy to
mess up...

this is just like `.compareTo()` in Java!

Avoiding the Common Mistake:


- 1) What I like to do is:

```
int streq(const char* a, const char* b) {  
    return strcmp(a, b) == 0;  
}
```

- 2) Now we have a more sensible way to test for equality:

```
if(streq(command, "print")) {  
    // print stuff  
} else ...
```

The Scary Ones: String Manipulation

- 1) `strcpy(a, b)` copies the string from **b** into the memory at **a**
 - 2) `strcat(a, b)` copies the string from **b** into the memory *after a*
- "string concatenate" 
 - 3) To do this **correctly**, you would have to **check the lengths before and after** every single string manipulation operation
- a) Starting with the initial string `mystr`:

	0	1	2	3	4	5	6	7	8	9
mystr										

- b) If we `strcpy` the string, "this" into `mystr`:

	0	1	2	3	4	5	6	7	8	9
mystr	t	h	i	s	\0					

`strcpy(mystr, "this")`

- c) If we `strcat` the string, "is" into the modified `mystr`:

	0	1	2	3	4	5	6	7	8	9
mystr	t	h	i	s		i	s		\0	

`strcat(mystr, " is ")`

- d) If we `strcat` the string, "BAD" into the modified `mystr`:

	0	1	2	3	4	5	6	7	8	9		
mystr	t	h	i	s		i	s		B	A	D	\0

`strcat(mystr, "BAD")`

- e) Overall, after String Manipulation, `mystr` is now:

	0	1	2	3	4	5	6	7	8	9		
mystr	t	h	i	s		i	s		B	A	D	\0

`strcpy(mystr, "this")`

`strcat(mystr, " is ")`

`strcat(mystr, "BAD")`

(if I were to `strcpy` again it would overwrite 'this' etc.)

How Do We Avoid That?

1) Just don't do string manipulation in C.

- **Seriously.**

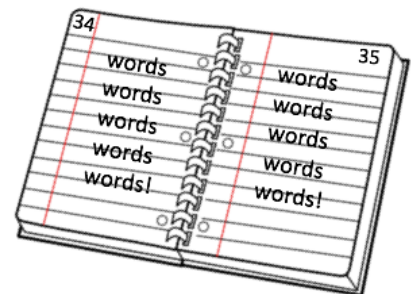
2) There are *absolutely* ways to do it correctly

- Such as using a non-C-standard string representation
- Or writing long, complicated functions to handle all the cases
- but GEEEZ **you have better things to do with your time**

Files:

The File Paradigm:

- 1) A file is **a big array of bytes**
- 2) Each byte has an **address** position within the file, which starts from 0
 - Just like an array
- 3) But unlike an array, the file has **current position**
- 4) It starts at 0, and every time you **read or write** something, it moves ahead
 - Like an old-timey tape drive
 - Or a notebook



What's **in** a File?

- 1) The meaning of the data in a file is *up to you*.
- 2) A **file format** defines the structure and meaning of data in the file.

all files are sequences of 0s and 1s, grouped into **bytes**.

A.java

```
public class A {  
    public static void  
    main(String[] args) {  
        System.out.println("h  
ello world"); } }
```

Text files are like a big string. They hold text and nothing else.

Binary files are... pretty much everything else.



Remarks:

- 1) Who defines file formats?
 - Whoever wrote the program that creates them.
- 2) Some file formats are **open**; many are **closed** (no documentation on them).
- 3) Reverse engineering closed file formats is a fun and enlightening pastime.

File Extensions are not Magical:

- 1) **BY THE WAY, ENABLE FILE EXTENSION DISPLAY IN YOUR OS**
- 2) A file's **extension** is just part of its name.
- 3) It has *no effect* on its contents.



well, it might be *named* like a powerpoint file, but it definitely *is not one*. the contents are unchanged.

Remarks:

- Opening this in PowerPoint would fail, since the format is completely wrong.
- But since it's no longer named correctly, other things that depend on it won't work anymore...

Opening and Closing Files in C:

- 1) You open files like this (all this is in stdio.h):

```
FILE* f = fopen(name, mode);
```

- **name** is the name of the file
- **mode** must be one of the following:
 - ⇒ "r", "w" for reading *or* writing **text files**
 - ⇒ "rb", "wb" for reading *or* writing **binary files**
 - ⇒ "r+", "rb+" for reading **AND** writing **files at the same time**

- 2) Close them like this:

```
fclose(f);
```

- 3) **Donut** 🍩 **forget to close them!**

- If you don't close them after changing them, **your changes may or may not actually end up in the file.**

What is a **FILE*** Exactly?

- 1) It is *not a pointer to the data in the file*
- 2) Pointers are also used to point to *objects*
 - C may not have built-in OOP facilities but it's still a common pattern!
- 3) You can't *do* anything with a **FILE*** besides pass it to functions which expect them as arguments
 - Like **fclose()**!
- 4) So, treat a **FILE*** as a sort of black box.

Reading and Writing Text Files:

- 1) Actually, we've seen this already!

```
fgets(buffer, sizeof(buffer), f);  
fprintf(f, "hello!\n");
```

- 2) This is because **stdin** and **stdout** (and **stderr**) are **FILE*s** too.
 - **printf("hi")** is short for **fprintf(stdout, "hi")**
- 3) Wait... but how is the *console a file*?
 - It's not stored in the file system
 - It's kind of... created in real time? as the user types???
 - **Everything's a file.**
 - the concept of a "**list/array of bytes**" is a common and useful one, so many things are "**files**"

Where are we?

- 1) **ftell(f)** gives the current file position (**distance from beginning**)
 - This is measured in **BYTES**.
- 2) **feof(f)** tells you if you are at the **end of the file (EOF)**.
 - It's commonly used with text files, like...
 - Reading all the lines from a file!

```
while(!feof(f))  
    read_a_line(f);
```

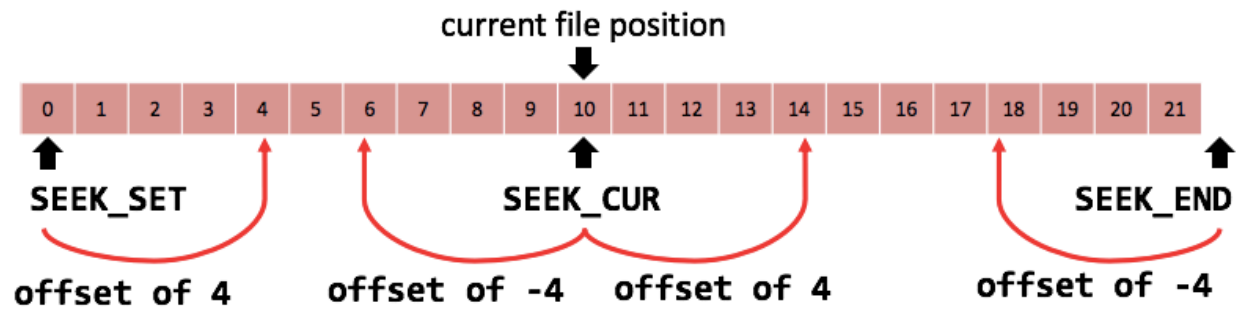
This is analogous to (in **Java** syntax):

```
while(scanner.hasNext())  
    scanner.nextLine();
```


Moving Around:

1) You move around the file with **fseek(f, offset, how)**

the meaning of the offset depends on what you pass for "how"...

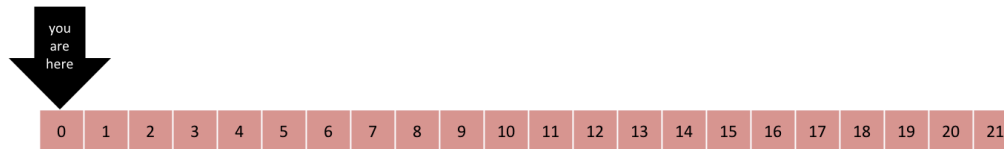


you can't go off the ends of the file.

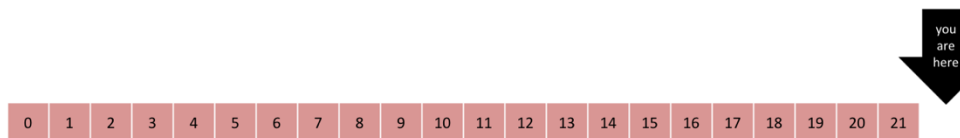
How Big is the File?

1) We can combine **fseek()** and **ftell()** to figure it out.

a) Initially, the file is at position 0:

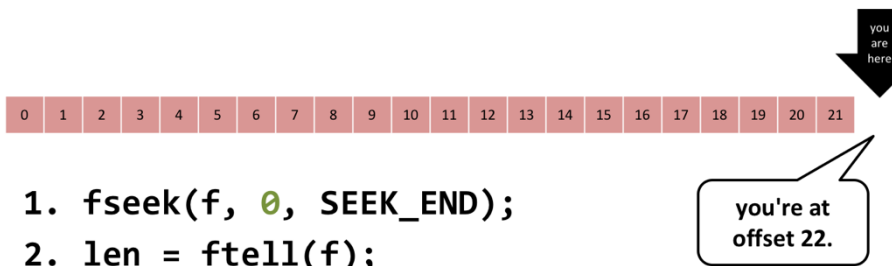


b) After **fseeking** to the end of the file:



1. `fseek(f, 0, SEEK_END);`

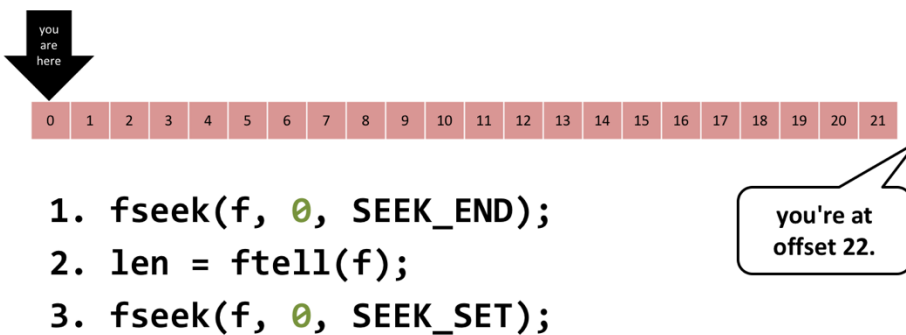
c) After **ftelling**:



1. `fseek(f, 0, SEEK_END);`

2. `len = ftell(f);`

d) After **fseeking** to the beginning of the file:



1. `fseek(f, 0, SEEK_END);`

2. `len = ftell(f);`

3. `fseek(f, 0, SEEK_SET);`

C-Structs, Enums, Typedefs:

Command-Line Arguments:

Oh yeah:

1) Remember `public static void main(String[] args)?`

2) Well you have those in C too

(`char**` is "an array of strings")

```
int main(int argc, char** argv)
```

`argc` is "argument count"

`argv` is "argument values"

`argv[0]` is always the name the user
used to run this executable.

the arguments really start at `argv[1]`.

and now you know how to handle
command line arguments :B

Remarks:

1) When you run a program from the command line, you can imagine it doing

`"status = main(things, you, typed)"`

2) To find out the status of a program (i.e. what it returned from main), you can use
the `$?` variable in bash

- Yes, a variable named `$?`

⇒ Who the hell designed bash

Structs, classes' evolutionary ancestors:

Adding Some Structure:

- 1) You can think of a **C struct as a class without most features**
- 2) You can put **data in it**, and *that's it*

```
struct Food {  
    char    name[10];  
    double  price;  
    int     stock;  
};
```

then, to make a variable of it:

```
struct Food grapes;
```

you have to write struct here. cause *reasons*.

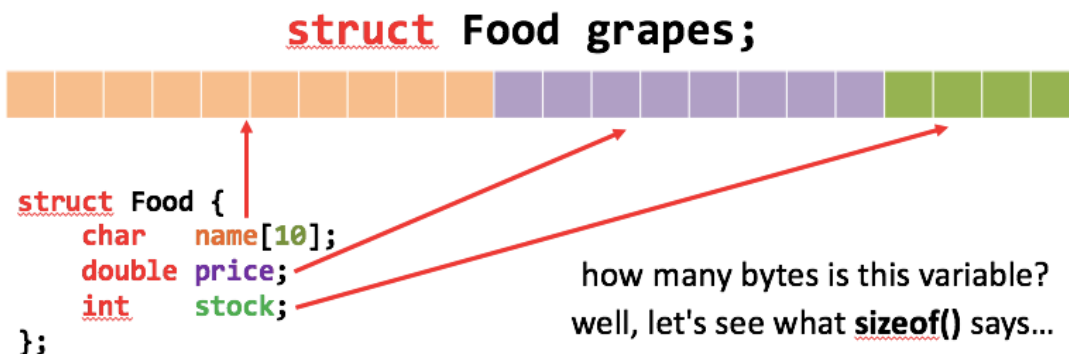
don't forget this dumb semicolon!

Remarks:

- 1) The **semicolon** is important and forgetting it will give you the stupidest flood of **compiler errors** imaginable
- 2) The name that comes after the "**struct**" keyword is called the "**tag**" and is in a separate namespace from all other names
 - this is a really silly "feature" from the earliest days of C's development

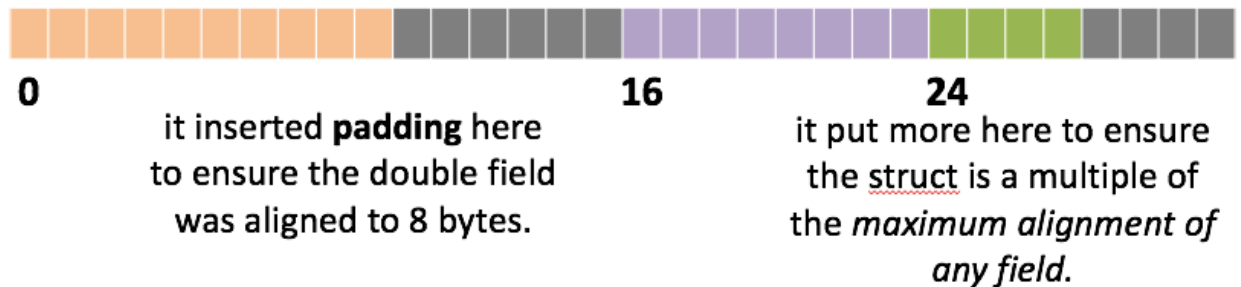
The Memory Representation:

- 1) All the **struct fields** are **allocated inside** the **struct variable**



Alignment and Padding:

- 1) The C compiler is free to **position** your fields in memory any way
– it **will** keep them in the same order, though.
- 2) Typically, it will **align** the fields
- 3) Let's use **offsetof()** from **<stddef.h>** to see where they ended up



Remarks:

- 1) **Memory Alignment** means "n-byte values have addresses that are a multiple of n"
- 2) **Doubles** are **8 bytes**, so they must appear at addresses that are multiples of 8
- 3) Since **doubles** need to be **8-byte aligned**, the whole struct must be a multiple of 8 bytes, so that an array of this struct will keep their doubles aligned
- 4) the **rules for alignment are kind of esoteric** but can sometimes be useful for laying out structs in a more memory-efficient way
– But it's not portable at all

typedef Struct:

- 1) Usually, you'll see **structs** declared like:

```
typedef struct Food {  
    char    name[10];  
    double price;  
    int     stock;  
} Food;
```

why? so you don't have to
write 'struct' on the variables:
Food grapes;

Remarks:

- 1) You can leave out the struct tag name in this case, but it's fine to duplicate the name so you can write things either way
– But what you shouldn't do is *give it a different name and tag, like who does that*

Typedefs:

OK What *is* That **typedef** Thing, Really:

- 1) **typedef** is a way of making a *type alias*
 - In other words, a more convenient **name** for a **type**
- 2) The syntax of a **typedef** is a **variable declaration**...
 - but with **typedef** in front.

typedef **int** **X**;

now **X** is another name for **int**.

X **x**;

this is the same as **int** **x**;

these'll come up again with function pointers...

Remarks:

- 1) In the same way that you can make constants that are convenient names for important values
- 2) Now, when you write "**typedef struct { ... } Foo**;" does that mean that "**struct { ... } Foo**;" is a **variable declaration**?
 - Yes. yes, it does. **Foo** is a variable whose type is... that **struct**.
 - You can make these "**anonymous**" **structs** in C.
 - ⇒ Not terribly useful, but there they are.

aaaaanywaaaaaaaaay Back to Structs:

Initializers:

1) You can initialize struct variables with a nice-looking syntax:

```
Food grapes = {"grapes", 3.99, 20};
```

you have to write the values in the
order they were declared.

arrays work too:

```
Food produce[] = {  
    {"grapes", 3.99, 20 },  
    {"bananas", 0.89, 300},  
    {"cucumbers", 1.49, 50 },  
};
```

2) This is one of the few places C syntax is nice is in initializers, for some reason.

Field Access (The . Operator):

1) It works just like Java!

...or does it

```
produce[0].price = 2.49; // good!  
Food* pgrapes = &produce[0];  
pgrapes.price = 2.99; // error!
```

for very silly reasons, pointers-to-structs use a
different operator to access fields:

```
pgrapes->price = 2.99; // good!
```

Passing and Returning **structs**:

- 1) If you pass a struct to, or return a struct from, a function...
- 2) **The whole struct is copied.**

```
typedef struct Huge {  
    int arr[256];  
} Huge;  
void func(Huge thing) {}  
...  
Huge huge;  
func(huge); // copies 1KB of data
```

besides being inefficient,
this is usually not what we
want.

- 3) Passing structs by value *can be very useful* – if you want the same copying semantics as e.g. ints and such

Passing **structs** by Reference:

- 1) Instead, often **structs** are passed as a pointer – "by reference"

```
void func(Huge* thing) {}  
...  
Huge huge;  
func(&huge); // copies size of 1 pointer
```

now, when func modifies the struct through the
pointer, those changes show up here too.

any type can be passed "by reference" like this!

- 2) **&** is the "address of" operator

Data Structures with **structs**:

- 1) Linked Lists, Trees, etc. have "**nodes**" that point to each other
- 2) To make a pointer inside a struct to the same type, you have to:

```
typedef struct Node {  
    int value;  
    struct Node* next;  
} Node;
```

you HAVE to use the **struct Node** form here.
if you leave it out... well, let's see

gcc is well-known for its, uh, *very helpful* error messages

- 3) "unknown type name Node", "request for member 'blah' in something not a structure or a union" etc etc etc

Enums:

Ah, Something Less Weird:

1) An **enum** is a way of defining (*usually related*) **constants**

- When I think of **enums**, I think of *choices*

```
typedef enum {  
    Red,      = 0  
    Orange,   = 1  
    Yellow,   = 2  
    Green,    = 3  
    Blue,     = 4  
    Purple    = 5  
} Color;
```

enum names work like **struct** names, and we **typedef** them too.

unlike **structs**, **enums** are *not* a new type.

the values are just **ints**, starting at 0.

2) Color is not really a special type; it's like another name for int

- I can write "int x = Red;" or "Color c = 45;", nothing is stopping me

Three Ways to Define **Constants** in C:

```
enum {  
    Red,  
    Green,  
    Blue,  
};
```

```
const int Red    = 0;  
const int Green  = 1;  
const int Blue   = 2;
```

these are all *subtly* different

```
#define Red    0  
#define Green  1  
#define Blue   2
```

but in most cases, they are virtually interchangeable

so why use **enums**?

1) **#define** came first; then **enum**; then **const**

2) if you like typing a ton, **const** is your friend

3) **#define** is slightly different as it will do *textual replacement* of the names with the values, but the end result is the same

Enums Indicate Intent:

- 1) When you choose to represent something a certain way...
- You are communicating to others that it has a particular meaning

```
typedef enum {  
    Red,  
    Green,  
    Blue,  
} Color;
```

this says to me: "there are **three possibilities** for **Colors**; here they are"

```
#define Red    0  
#define Green  1  
#define Blue   2
```

this says to me: "these are **convenient names** for these **integer values**"

using enums can also help the *compiler* understand your code better!

Remarks:

- 1) if you use a **switch()** on an **enum** value, the compiler can detect if you've missed a case!
 - This avoids so many bugs!!!!
- 2) Often you will use an **enum** and *never care* about its int representations – cause all you care about is the meaning of the constants.
 - this is **good**. this is in **general good**. same with the other kinds of constants.

Binary Files and **structs**:

sizeof()

1) **sizeof()** is a **compile-time operator** which tells you how many **bytes*** something takes up.

```
printf("%d\n", sizeof(char));  
printf("%d\n", sizeof(int));  
char carr[10];  
int iarr[10];  
printf("%d\n", sizeof(carr));  
printf("%d\n", sizeof(iarr));  
char* p = carr;  
printf("%d\n", sizeof(p));
```

by definition,
sizeof(char) == 1

what's this print?
it depends

BE CAREFUL:
sizeof(array) gives you
the **BYTE SIZE**.

*C does not know how big an
array at a pointer is.*

Remarks:

*it's actually the number of *chars* something takes up, but on any machine made and in common use in the past 35 years, a char is a byte.

1) **sizeof(int)** is often 4. *often. but not necessarily.*

- In the 16-bit era, it was often 2.

- On 64-bit targets, it might be 4 or 8, depending on your compiler options and your platform's conventions.

2) **sizeof** IS A **COMPILE-TIME OPERATOR**. IT CANNOT FIND THE SIZE OF SOMETHING AT RUNTIME.

Reading and Writing Binary Files:

- 1) `fread` reads data and puts it into a buffer you provide. (like `fgets`.)

```
fread (&thing, sizeof(thing), 1, f);
```

- 2) `fwrite` takes data out of a buffer you provide and writes it

```
fwrite(&thing, sizeof(thing), 1, f);
```

- 3) You can do this with any type – arrays, structs, ints...

```
int x = 34;
```

```
fwrite(&x, sizeof(x), 1, f);
```

Remarks:

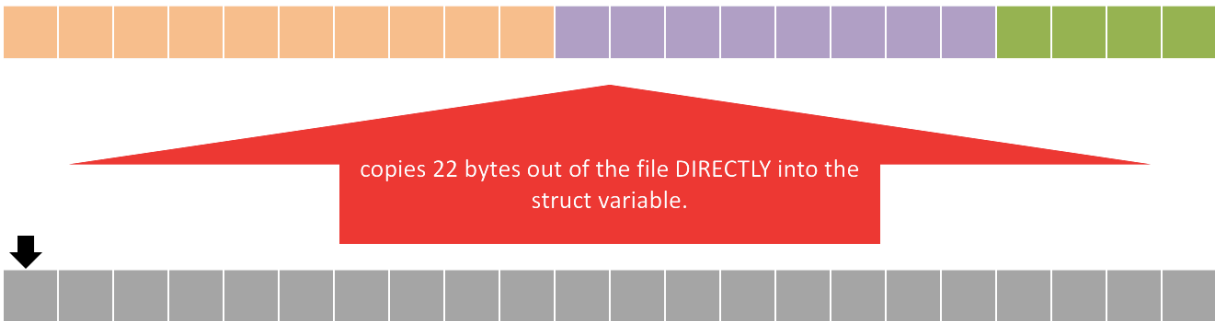
- 1) The second and third parameters are weird, they're "size of an item" and "how many items" respectively
- 2) They will read/write `size*number` bytes, always
- 3) and since multiplication is commutative, you can technically pass them in either order?
 - It will have effects for the return value, though.
 - ⇒ If you care about it.

When you use **fread**/**fwrite**...

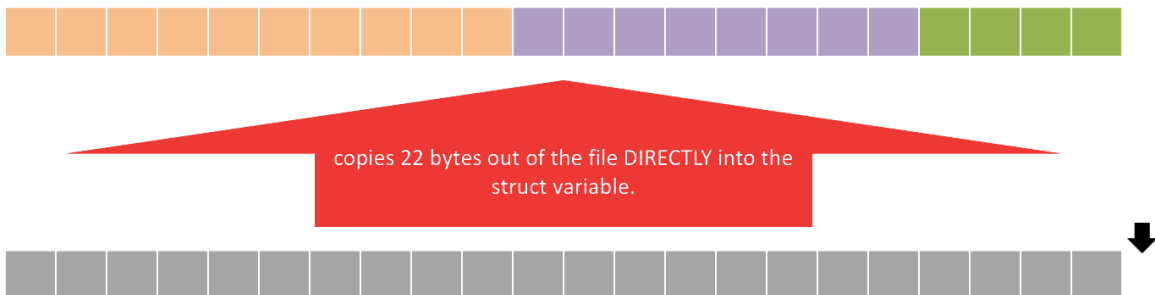
1) They just copy blobs of bytes directly between memory and the file

a) If you **fread**...

```
Food grapes;  
fread(&grapes, sizeof(grapes), 1, f);
```

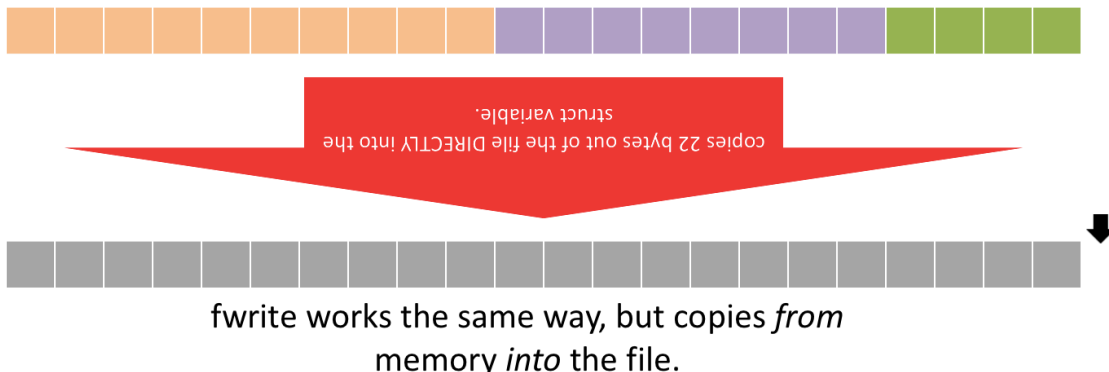


```
Food grapes;  
fread(&grapes, sizeof(grapes), 1, f);
```



b) If you **fwrite**...

```
Food grapes;  
fwrite(&grapes, sizeof(grapes), 1, f);
```



C-Pointers and Arrays:

What's a Type?

Values:

1) Your programs deal with all kinds of **values**.

17

which of these can you
add 1 to?

3.14159

which of these can
you use [] on?

"name?"

which of these can
you use . on?

'X'

can you use the same kind of
variable for all these?

{"grapes", 3.99, 40}

Remarks:

- 1) You can add 1 to the integer and the float, but not really to the others...
- 2) You can use [] on the string, but none of the others...
- 3) You can use . on the struct, but none of the others...
- 4) and no, you'd have to declare all these as different kinds of variables

Types:

- 1) **Types** are how we categorize **values**.
 - We do this based on **what we can do with those values**.
- 2) Types are what you put in front of variable names.
- 3) What type is each of these values?

int 17

float 3.14159

char*, "name?"

char 'X'

Food {"grapes", 3.99, 40}

? - well, it's a **const char***.

Type Constructors:

- 1) If you have a **value**, you can make a **new value** based on it.
 - If you increment 5, you get 6.
- 2) If you have a **type**, you can make a **new type** based on it.

int a single integer. in Java, [] is a type constructor.

int[] an **array** of integers.

int[][] an **array** of **arrays** of integers.

these are all **different, incompatible types**, because
they hold different *kinds of values*.

what other way does Java let you make **new types**? how
about C?

Remarks:

- You can stick [] on the end of **anything** to get "an array of that thing". **that's how you know it's a type constructor**.
- **Classes** and **structs** are **new types**. the whole declaration is the "type constructor."
- **Java** also gives you generics: **ArrayList<Integer>** is a **different type** from **ArrayList<Float>**.

What are Pointers?

Memory!

- 1) **Memory** is a **big one-dimensional array of bytes**
- 2) Every **byte value** has an **address**
 - This is its "**array index**"
 - **Addresses start at 0**, like arrays in C/Java
- 3) For values **bigger than a byte**, we **use consecutive bytes**
 - The **address of any value**, **regardless of size**, is the **address of the first byte** (*smallest address*)
- 4) Here is an **int** variable
 - What is its **sizeof()**?
 - ⇒ its **sizeof()** is 4 in this case - 4 bytes long
 - What is its **address**?
 - ⇒ its **address is 0xDC04**
 - What is its **value**?
 - ⇒ its value could be either **0xDECOEFBE** (**big-endian**) or **0xC0DEBEEF** (**little-endian**)

Addr	Val
...	...
DC0B	44
DC0A	04
DC09	02
DC08	88
DC07	BE
DC06	EF
DC05	C0
DC04	DE
DC03	2A
DC02	27
DC01	0E
DC00	42

Lockers:

- 1) Think about a locker. what is its purpose?
 - To **contain** things.
- 2) How are lockers identified?
 - They're **numbered**.
- 3) How do you access a locker?
 - By knowing the locker's **number** and **combination**.
 - but for this example, let's assume *there are no locks*.
- 4) How would you *give someone else access* to your (lock-less) locker?
 - Would you rip it out of the wall?
 - No... you **give them the locker number**.

Like a Locker Room, But Without the Nudity:

1) Just like a locker **contains things...**

- Variables contain **values**
- But a variable *is a thing itself*

2) Each variable is like a locker:

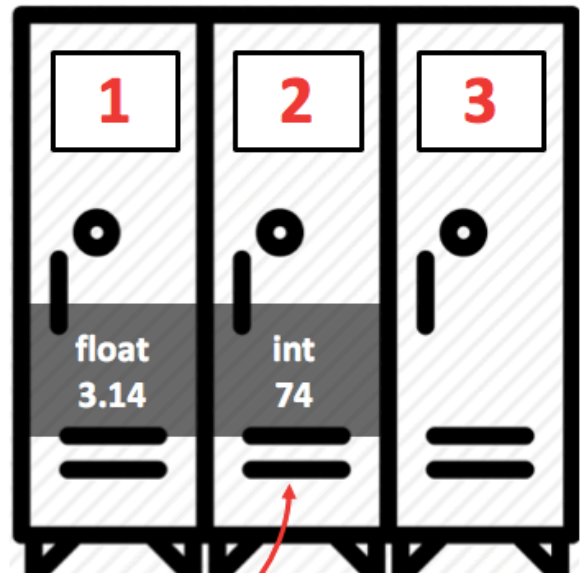
- It has a **number**: its **address**
- It **contains** something: its **value**
- It **belongs** to someone: its **owner**

3) How do you *give someone else access* to your variable?

- **You give them the locker number.**

⇒ Which is its **memory address**.

Variable names are just a convenient way to refer to their addresses!

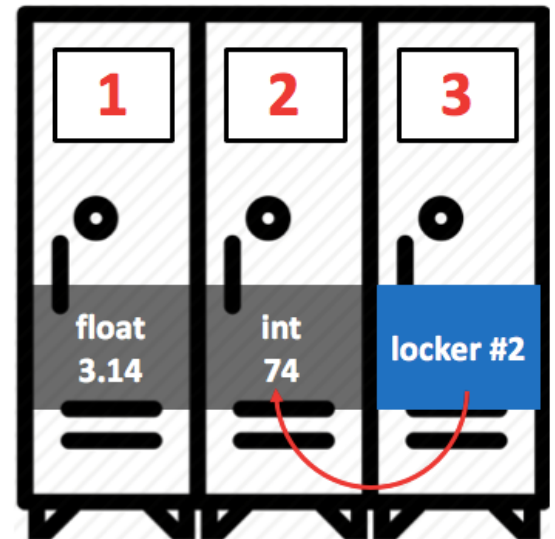


what if we put this
thing in a locker?



Pointers:

- 1) If we put a slip of paper in locker 3 which says "locker #2"...
- 2) Now we can access *two* things:
 - The locker *itself* (3), and
 - The locker that it *points to* (2)
- 3) **A pointer is a variable which holds another variable's memory address.**
- 4) If you have a pointer, now you can access *two* things:
 - The pointer variable *itself*, and
 - The variable that it *points to*
- 5) *"every problem in CS can be solved with another level of indirection."*



Pointers in C:

Pointer Variables and Values:

1) Like Java's `[]`, C's `*` is a **type constructor**:

```
int x;      // integer
int* p;     // pointer to an int
int** pp;   // pointer to a pointer to an int
```

2) You **get the address** of a variable with the **address-of** operator:

```
int* p      = &x;
int** pp    = &p;
```

3) You can use it on just about **anything with a name**:

```
&x          // address of x
&arr[10]    // address of item 10 in arr
&main       // address of the main function!
```

Remarks:

- 1) You **CAN'T** use **address-of** on "temporaries" (values without a name)
 - e.g. `&5`, `&&x`, `&f()`.. these are all invalid.

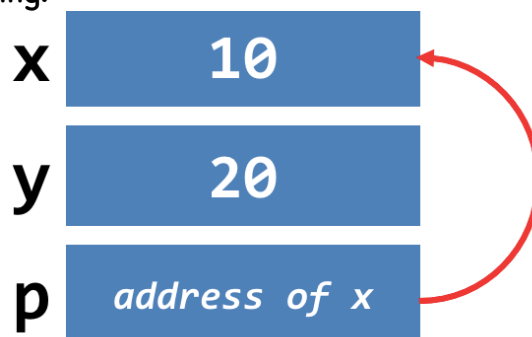
But What Does that Mean:

1) In this segment of code:

a) The statement, `int* p = &x;` does the following:

```
int x = 10;  
int y = 20;  
int* p = &x;
```

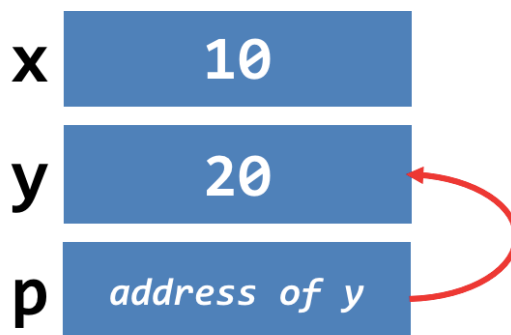
we say *p points to x*.



b) The statement, `p = &y;` does the following:

```
int x = 10;  
int y = 20;  
int* p = &x;
```

`p = &y;`



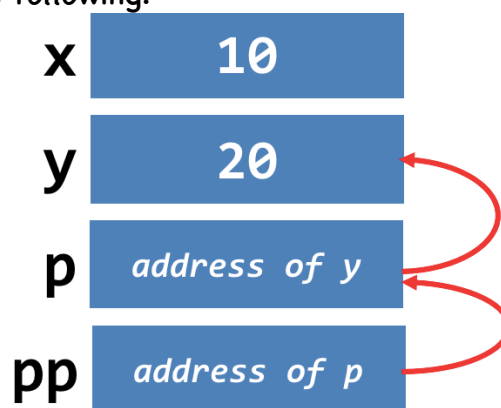
c) The statement, `int** pp = &p;` does the following:

```
int x = 10;  
int y = 20;  
int* p = &x;
```

`p = &y;`

you can reassign **where**
pointers are pointing.

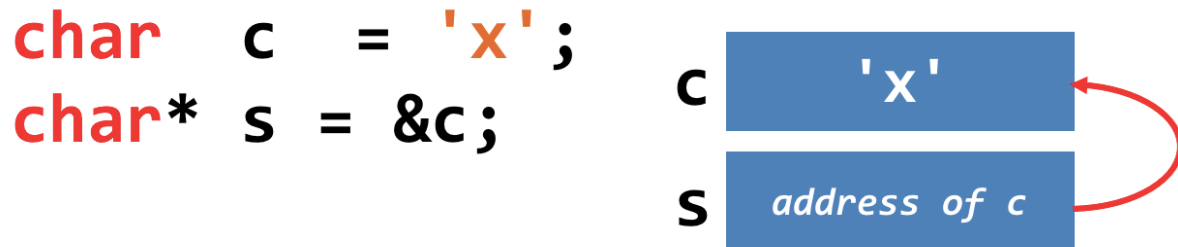
```
int** pp = &p;
```



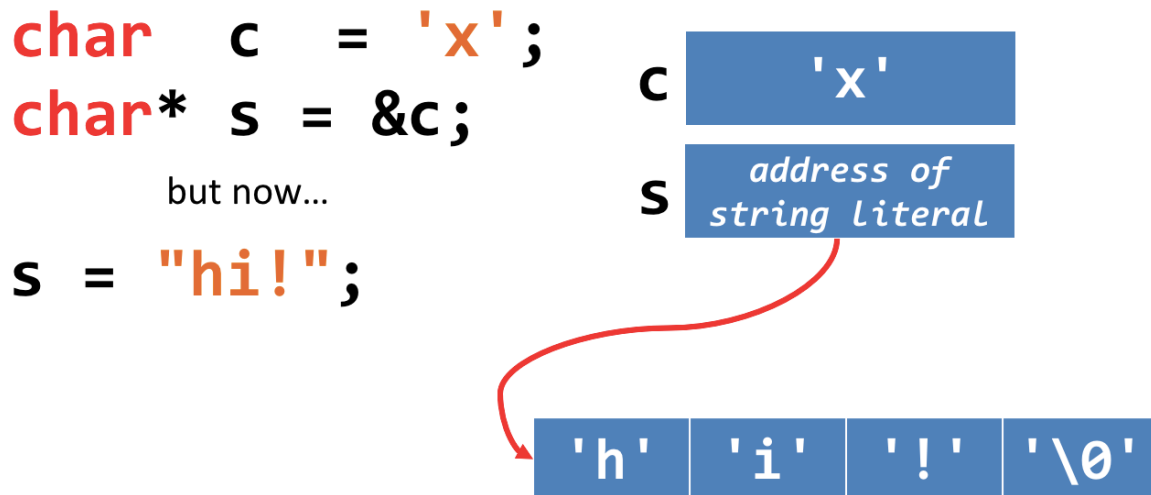
double pointers aren't
that scary...

Pointers and Arrays:

- 1) A pointer can point to **one or more values**.
 - 2) A **char*** may point to a **single char**, or to **an array of characters**.
- a) Initially, with the given code:



b) But now:



Remarks:

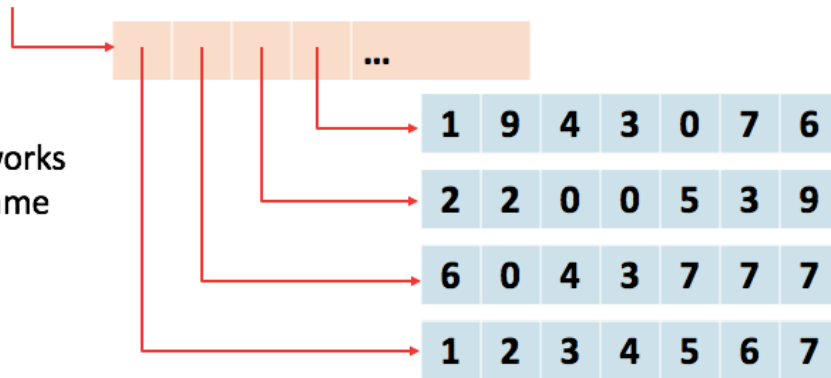
- 1) Lots of people get confused by this, and it's ok, because it's confusing.
- 2) It's just context. you have to know what it's pointing to, to know which it is.
 - e.g. the `char**` given to `main` is an array of strings. it's documented that way, so you treat it that way.
- 3) If you think about it, "**pointing to a single value**" is the **same as** "**pointing to an array of length 1**"...

Multi-Dimensional Arrays:

1) We already saw single-dimensional arrays, but...

```
int** arr2d = ...
```

a Java `int[][]` works
exactly the same
way!



2) **THIS IS ACTUALLY COMPLETELY DIFFERENT FROM AN "int arr[4][6];" OR SOMETHING**

Printing Pointers:

1) The `%p` format specifier is used to print pointers:

```
printf("address of x = %p\n", &x);
```

2) This prints a **hexadecimal representation of a pointer**.

3) Pointers can be **null**, too, but you have to **YELL IT**:

```
int* p = NULL;
```

```
printf("p = %p\n", p);
```

4) Weirdly, on Linux (which thoeth runs), it prints "**nil**" instead of "**null**."

5) Oh, and one more thing...

6) **In C, it's possible to have a pointer that is *not null*, but is *invalid*.**

- Accessing an invalid pointer gives **undefined behavior**.

- It might **segfault**. it might mess up other variables. you don't know.

OK, We Get It, You're Bizarre:

- 1) There is *one kind* of variable that behaves strangely with &
- 2) **GUESS WHICH KIND**

```
int arr[10];    // OH BOY.  
int* p1 = arr;  
int* p2 = &arr;  
printf("p1 = %p, p2 = %p\n", p1, p2);
```

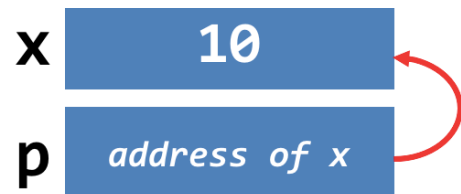
- 3) **p1** and **p2** are exactly the same here.
- 4) **ONLY for array variables:** you can get their address **by using their name alone** *or with the address-of operator.*
- just... why?

Accessing the Value(s) at a Pointer:

The **Value-At** (or "**Dereference**") Operator:

- 1) ***** is the **value-at operator** - it's the **inverse of &**
 - Every time you use it, you **remove** a star
 - 2) It **accesses the variable** that a pointer **points to**
 - We say that it "**dereferences**" a pointer
- a) In the given code, **p** currently points to:

```
int x = 10;
int* p = &x;
```



- b) Changing the value of **p** (via **dereference**) does the following:

```
int x = 10;
int* p = &x;
*p = 15; // changes x!
```



- c) Printing out of the pointer's value does the following:

```
int x = 10;
int* p = &x;
*p = 15; // changes x!
printf("%d\n", *p); // prints 15
```



Remark:

- 1) You'd think it'd be the other way... like, ***** would give you an **int***, and **&** would give you an **int**?
 - Or maybe pointer types should be **int&**? like "**address of int**"?

That Stupid -> Operator:

1) If you have a **pointer** to a **struct**, you must **access its fields** with ->

```
Food grapes = {"grapes", 3.99, 20};  
grapes.stock--;  
Food* pgrapes = &grapes;  
pgrapes->price = 2.99;  
(*pgrapes).price = 2.99;
```

} these are identical
in meaning.

but no one writes the second
one...

Remarks:

1) The **(*a).b** syntax is there to show you that yes, it really is just using the
dereference operator underneath

The **Array-Indexing** Operators:

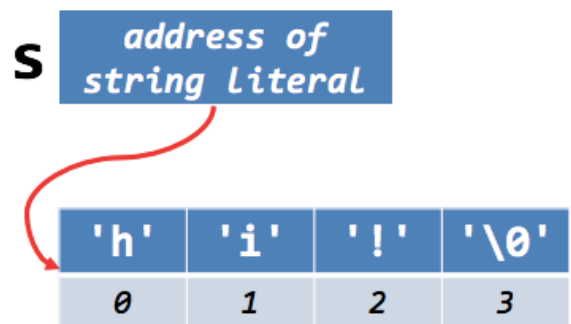
1) **p[n]** means "access the **n**th item pointed to by **p**."

```
char* s = "hi!";
```

```
char c = s[2];
```

now **c** contains '!'

```
char d = 2[s];
```



wait whaaaaaaaaaaaaaaaaaaaaaaaaaaaaaat

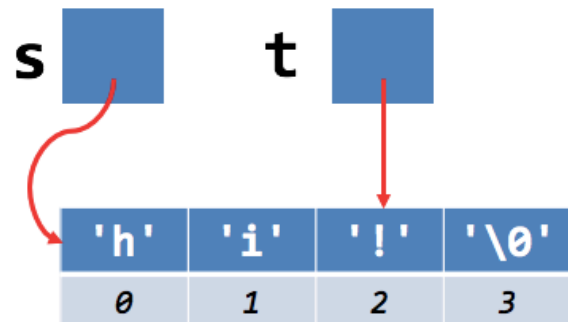
Pointer Arithmetic:

It's All Just, Like, Numbers, Man...

- 1) **Pointers** hold **memory addresses**.
 - **Memory addresses** are **numbers**.
- 2) It's useful to do **arithmetic** on **memory addresses**.

```
char* s = "hi!";  
char* t = s + 2;
```

what we've just done is **calculated a new pointer** based on an old one.



what would this print?

```
printf("%c\n", *t);
```

- It prints '!'. because **t** is pointing at the '!'.
The `%c` format specifier prints a single character, and `*t` dereferences the pointer `t` to get the character at the memory address it points to.

Remarks:

- 1) There is **no dereferencing** happening in the **pointer arithmetic**.
 - We are operating **on the pointer itself**.

What the Brackets Really Do:

1) `p[n]` in C *really* means "dereference address `p + n`"

$$\begin{aligned}
 & s[2] \\
 = & *(s + 2) \\
 = & *(2 + s) \quad \text{by commutativity.} \\
 = & 2[s] \quad \text{same as first step.}
 \end{aligned}
 \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \text{these are identical in meaning.}$$

we call the original pointer the **base** and the number we're adding to it the **offset**.

buuuuuuuut what about values
bigger than a char?

Tipping the Scales:

1) Let's say we have an array of ints:

```
int arr[3] = {1, 2, 3};
```

2) Let's say `arr` points to address `0xDC00`

3) What is `sizeof(int)`?

- 4 bytes.

4) `arr[1]` means `*(arr + 1)`...

- but where is the next value? at `0xDC01`?

- no. at `0xDC04`.

- and `arr[2]` is at `0xDC08`...

5) when you add an offset to a pointer, the offset is multiplied by the size of the item being pointed to before being added to the base address.

- this is called "scaling."

	Addr	Val
arr[2]	DC0B	00
	DC0A	00
	DC09	00
	DC08	03
arr[1]	DC07	00
	DC06	00
	DC05	00
	DC04	02
arr[0]	DC03	00
	DC02	00
	DC01	00
	DC00	01

C-Scope, Lifetime, and the Stack:

More Pointer Stuff:

const Pointers:

- 1) For any type *T*, a `const T*` is a `read-only` pointer to a *T*
 - You can `read` the data that it points to, but you can't `write` it
 - You *can* however change *where the pointer points to*

```
const char* s = "hello";  
s = "goodbye"; // fine!  
s[3] = 'x';    // COMPILER error!
```

- 2) `const` is a `type constructor` as well!

Pointer Casting:

- 1) `casts` convert from one type to another, like so: `(type)value`

```
int a = 20, b = 25;  
double x = a / (double)b; // now it's 0.8
```

- 2) You can cast `pointers` too

```
float f = 3.567;  
int* p = (int*)&f;    // p points to f...  
printf("%08x\n", *p); // interprets f as an int!
```

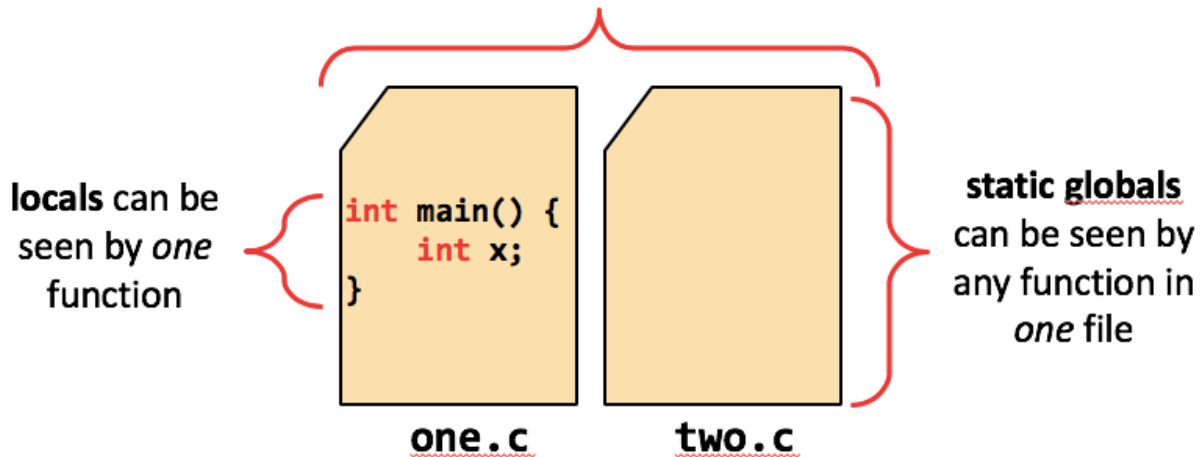
- 3) The computer **does not care what bits represent**
 - This kind of cast does **not** change anything in memory
 - ⇒ `f` is still there and it still holds 3.567
 - It only changes how we `view` that memory

Scope and Lifetime:

Scope:

- 1) **scope** is "where a name can be seen"
- 2) C has three levels of scope

globals can be seen by *any* function in *any* file



Remarks:

- 1) "name" in this case refers to variables, but can also apply to functions in C, and other things in other languages.

Global Variables are Terrible:

- 1) and you should almost never use them.
- 2) Almost any problem where you think you need one can be instead solved by using a local and passing by reference.
- 3) There *are* legitimate uses for them, but...
 - Unless you are being *forced* to use them...
 - ⇒ **avoid them.**

This goes for **statics** in **Java** too!

Lifetime:

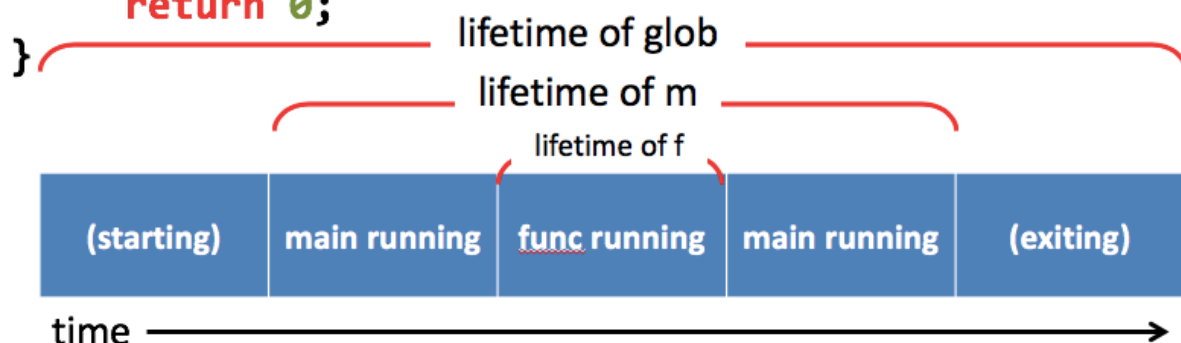
- 1) Every variable takes up space **in memory**
 - That memory **must be allocated**: reserved for the variable
 - and when no longer needed, **deallocated**: released for other use
- 2) **Lifetime** is a little more subtle than scope...
 - It's the time **between allocation and deallocation**
- 3) **Global Variables** last from program start to program exit
- 4) **Local Variables** only last as long as the enclosing function

Remarks:

- 1) Both kinds of globals do.
- 2) Lifetime and Scope are **not** the same thing, although the lifetimes and scopes of locals and globals overlap.

You're Watching the Lifetime Channel:

```
int glob = 0x910B;    void func() {  
int main() {          int f = 10;  
    int m = 10;      }  
    func();  
    return 0;  
}
```



Ownership:

1) *Ownership* answers the question: **who is responsible for deallocating a piece of memory?**

- Or: how do we determine *when it's okay* to deallocate memory?

2) Different languages deal with this in different ways.

- C mostly sticks its fingers in its ears, goes "LA LA LA," and pretends the problem doesn't exist.

3) Locals and Globals are easy:

- **Locals** are owned by **functions**

- **Globals** are owned by the **program**

4) but...

Remarks:

1) Hey, C's a **product of its time**, and to be fair, **ownership is a really tricky concept to get right!**

2) Rust seems to be doing a pretty good job of it, though.

If it Were Only That Simple:

1) In this Java code:

2) What is the **scope** of each variable?

3) What is the object's **lifetime**?

- It starts in func...

- But ends...

-where???????

```
public static void main(String[] args) {  
    String s = func();  
    s = null;  
}  
String func() {  
    String t =  
        new String("hello");  
    return t;  
}
```

Remarks:

1) We can't get around this.

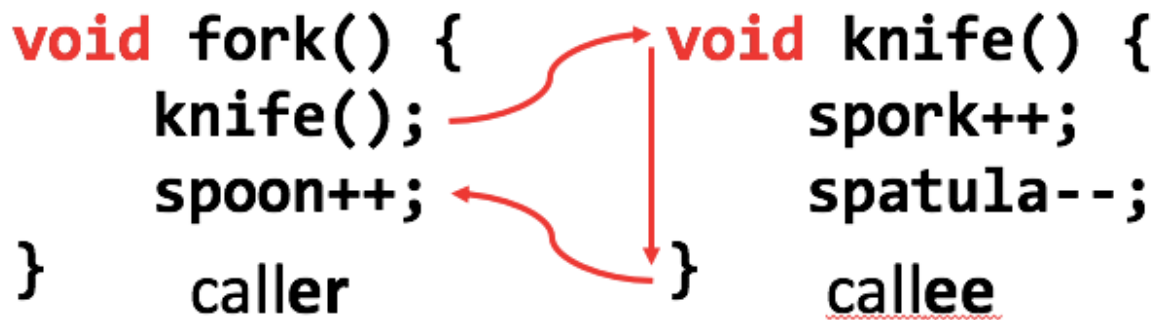
2) Global/Local lifetimes are just not flexible/powerful enough for many tasks.

3) unfortunately, beyond globals/locals, C totally screws it all up, oops

The Stack!!!

The Flow of Control:

- 1) When the caller calls a function, where do we go?
- 2) When the callee's code is finished, where do we go?

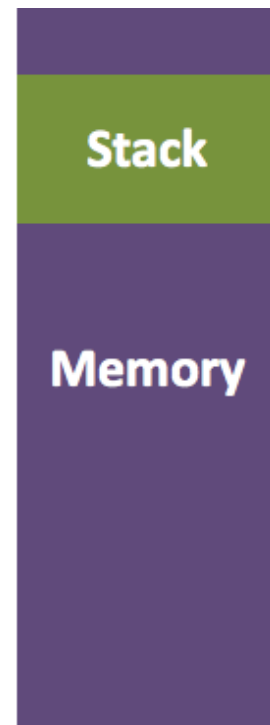


- 3) It doesn't matter if you use a **return** or not, when the **callee** finishes, it goes back to the line after the call in the **caller**.

What's the Stack?

- 1) It's an **area of memory** provided to your program by the OS
 - When your program starts, it's already there
- 2) The stack holds **information about function calls**.
- 3) It's not a *strict* stack
 - You can read and write any part of it
 - But it **grows** and **shrinks** like a stack
 - and we use "**push**" and "**pop**" to describe that
- 4) Each program* gets **one** stack
 - Cause only one function is running at a time

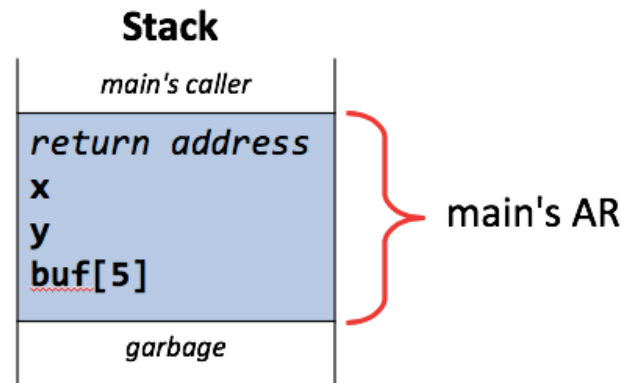
* really it's each **thread**



Activation Records (ARs):

- 1) When a function is called, a bunch of data is **pushed** onto the stack
- 2) This is the call's **activation record** (or "stack frame")
- 3) It contains **local variables** (including arguments) and the **return address**

```
int main() {  
    int x, y;  
    char buf[5];  
    return 0;  
}
```



Remarks:

- 1) *Somebody* calls main.
 - There's actually a few functions which run before main gets a chance.
- 2) The **return address** is where to go in the caller, when this function returns.

The Low-Level Layout:

- 1) Each variable (*including* array variables) gets enough bytes in the **AR** to hold its value
- 2) *Where* the variable is located is up to the compiler

```
int x;  
char arr[3];  
short y;
```

when I compiled this on thoth, this is the arrangement I got:

sizeof(int) == 4, so x gets 4 bytes
sizeof(arr) == 3, so it gets 3 bytes
sizeof(short) == 2, so.. yeah

but what's all that grey space?

idk lol, compiler does what it wants

Addr	Value
d02f	
d02e	
d02d	
d02c	return
d02b	
d02a	
d029	
d028	
d027	
d026	y
d025	
d024	
d023	
d022	
d021	
d020	x
d01f	
d01e	
d01d	arr
d01c	
d01b	
d01a	
d019	
d018	

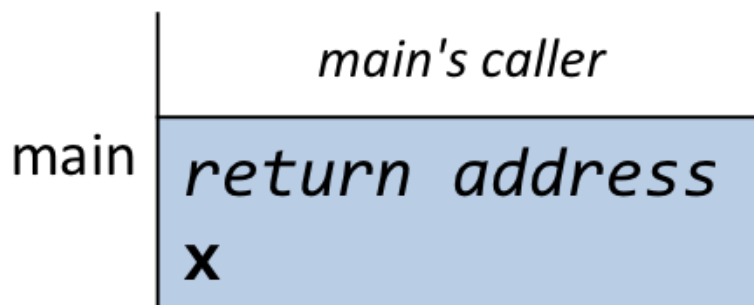
Call = Push, Return = Pop:

1) The stack grows when we call a function and shrinks when it exits

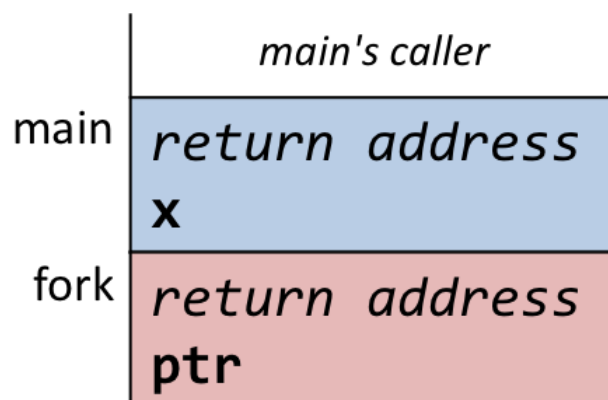
```
int main() { void fork(int* ptr) {  
    int x;    *ptr = knife();  
    fork(&x); }  
    return 0;  
}
```

```
int knife() {  
    return 10;  
}
```

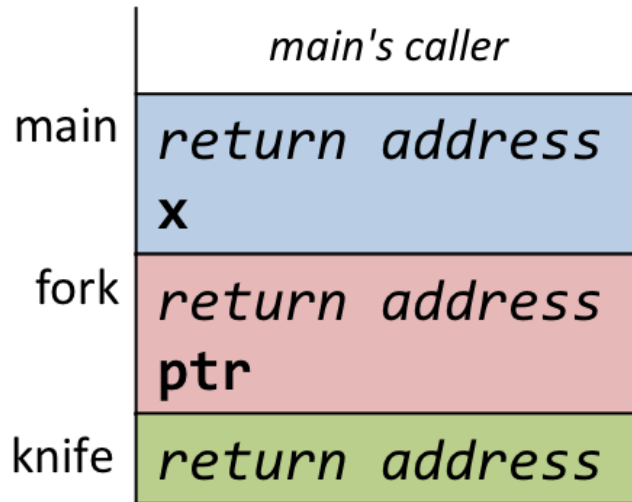
a) The program's AR in main without any other function calls is:



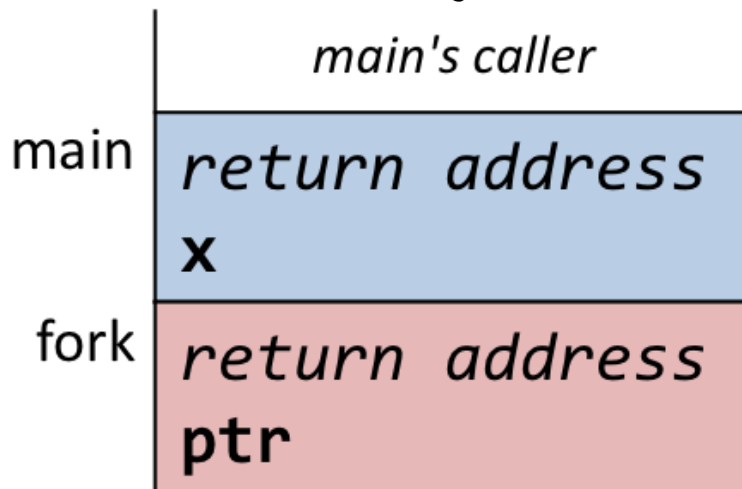
b) After `fork()` is called, the AR becomes:



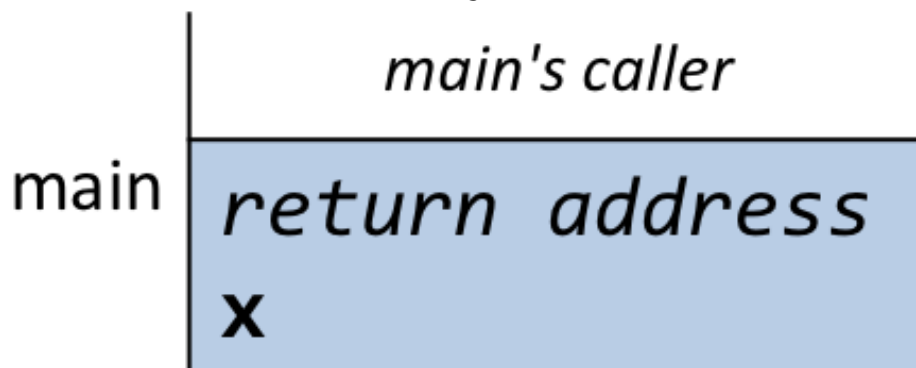
c) After **knife()** is called within **fork()**, the **AR** becomes:



d) After **knife()** is finished returning, the **AR** becomes:



d) After **fork()** is finished returning the **AR** becomes:



Recursive Functions:

1) Recursive functions *work by using the call stack as an implicit stack data structure*

a) Given the following recursive function (factorial):

```
int fact(int x) {  
    if(x <= 2) {  
        return x;  
    } else {  
        return x*fact(x-1);  
    }  
}
```

b) If **fact(5)** is called, the initial AR with **fact(5)** is:

	fact(5)'s caller
fact(5)	return address x = 5

c) After **fact(5)** is called, **fact(5)** calls **fact(4)**, and the new AR is now:

	fact(5)'s caller
fact(5)	return address x = 5
fact(4)	return address x = 4

d) After **fact(4)** is called, **fact(4)** calls **fact(3)**, and the new **AR** is now:

	<i>fact(5)'s caller</i>
fact(5)	<i>return address</i> x = 5
fact(4)	<i>return address</i> x = 4
fact(3)	<i>return address</i> x = 3

e) After **fact(3)** is called, **fact(3)** calls **fact(2)**, and the new **AR** is now:

	<i>fact(5)'s caller</i>
fact(5)	<i>return address</i> x = 5
fact(4)	<i>return address</i> x = 4
fact(3)	<i>return address</i> x = 3
fact(2)	<i>return address</i> x = 2

We've now reached the base case, so the recursive function begins returning value, and the call stack begins popping, until we're left with:

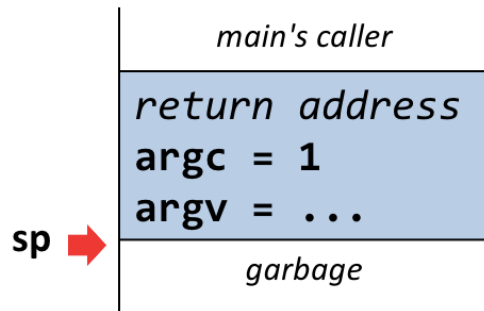
	<i>fact(5)'s caller</i>
fact(5)	<i>return address</i> x = 5

Remarks:

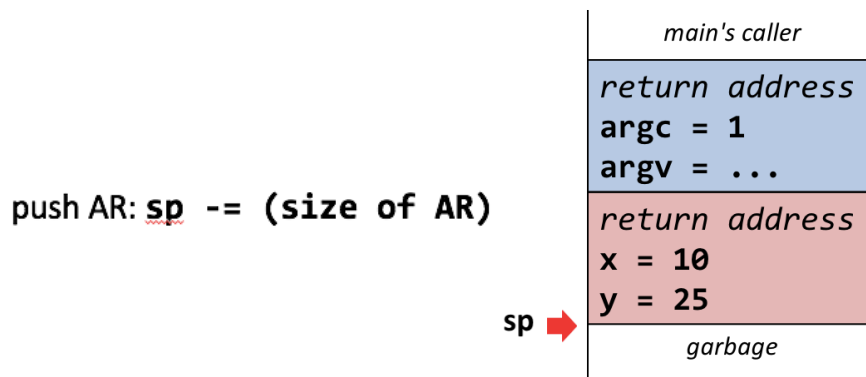
- 1) This is why anything recursion can do, iteration can do:
 - Recursion uses the call stack to store its data
 - Iteration can use a separate stack data structure to store the same data

But They Don't Really Go Away:

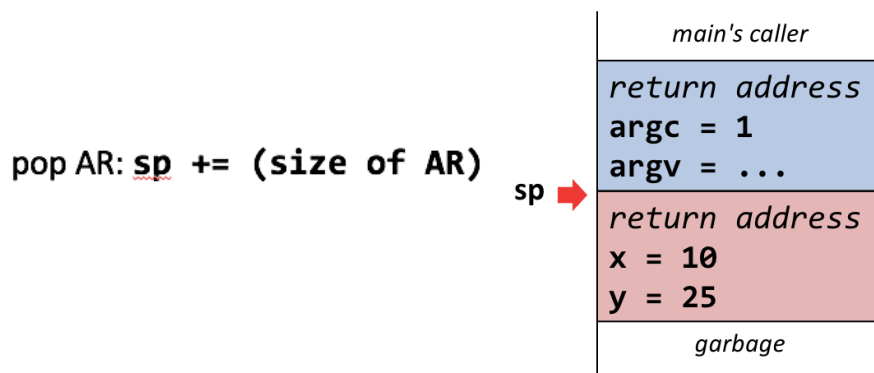
- 1) The stack is used *constantly*
 - So, it needs to be *really fast*
 - 2) So, we implement it as a **pointer**
 - The **stack pointer (sp)**
 - 3) **Pushing the AR** moves the **sp down**.
 - 4) **Popping the AR** moves the **sp up**.
- a) Initially, the stack pointer is:



- b) After pushing onto the call stack, the stack pointer is:



- c) After popping onto the call stack, the stack pointer is:



d) The **AR's memory** is still there, so calling another function is **BAD!**

but the AR's memory is still there.

so if we call another function...

this is where that garbage in uninitialized variables comes from!

sp →

main's caller
return address argc = 1 argv = ...
return address a = 10 b = 25 c = 10905928
garbage

Calling a function → Moves the sp.

Deallocate → changing where the sp points to, the memory still exists, though.

Don't Return Stack Arrays:

1) I think I showed this before

"function returns address of local variable"

2) When **func()** returns, what happened to its variables?

- *You have no idea*

3) You now have an **invalid pointer**

- It points to who-knows-what

⇒ it might crash

→ it might expose secrets

» who knows!!

```
int main() {  
    char* str = func();  
    printf("%s\n", str);  
    return 0;  
}  
char* func() {  
    char str[10] =  
        "hi there";  
    char* dummy = str;  
    return dummy;  
}
```


C - Memory Management:

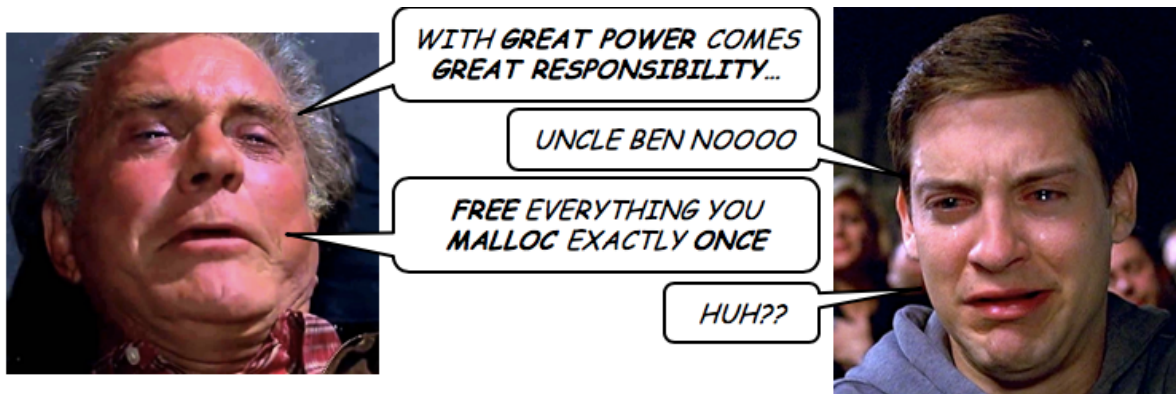
The Heap:

Two New Worlds:

- 1) **Local** variables are allocated... where, again?
 - What is their lifetime (from allocation to deallocation)?
 - ⇒ Locals are allocated on the stack (within activation records), and their lifetime is the body of the function
- 2) **Global** variables are put in the **data segment**
 - This area is **automatically allocated** by the OS
 - Its **lifetime** lasts for your entire program
 - ⇒ There is also a **read-only** data segment for constants
- 3) But the third place is **new...ish**
 - New
 - Like... `new Object()` kinda new

The Heap:

- 1) The **heap** is a **programmer-managed** area of memory
 - It has **nothing to do with the heap data structure**
- 2) You can **create and destroy pieces of memory on demand**
- 3) This is a powerful ability!
 - but remember...



Trainwreck:

- 1) The **stack** and **heap** both change size
- 2) The **stack** grows **down** (higher addresses to lower)
- 3) The **heap** grows **up** (lower addresses to higher)
- 4) If they meet, the program is **out of memory**
- 5) Why?
 - Why not have the stack grow up and the heap down?
 - Why not have them grow *outward* from the center?
 - Why not have them both grow up or down?
 - Why have them in the same area of memory *at all*?
 - ⇒ 64-bit architectures win here!
 - ⇒ In 16- or 32-bit address spaces, we don't have a choice



Remark:

- 1) Having the stack and heap grow towards each other allows for maximum memory utilization, e.g. if you have a huge heap and small stack
- 2) The stack *could* grow up but for whatever reason we've settled on the stack growing down in most architectures
- 3) In 64-bit architectures, the stack and heap addresses can be exabytes apart and therefore will never, ever collide

The Functions:

- 1) The heap functions are in **<stdlib.h>**
- 2) To allocate memory, you use **malloc**:
int* arr = malloc(sizeof(int) * 20);

- This allocates a 20-item int array **on the heap**
 - **malloc** takes the number of **bytes** to allocate, makes a block of bytes *at least* that big, and returns a pointer to it
- 3) When you're done with that block of memory, you use **free**:
free(arr);

Void Pointers:

1) If you look at the C stdlib docs...

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

2) An **int*** points to an **int**, a **FILE*** points to a **FILE** object...

3) A **void*** is a "universal pointer:" it can point to *anything*

4) A **void*** can be assigned *any other type of pointer*

```
int x;   float y;  
void* vp = &x; // fine!  
vp = &y;      // fine!
```

5) You can't *access the data* from a **void*** without casting

- Seems kinda useless...

Allocating a **struct** On the **Heap**:

1) This is the closest to Java's **new** as C gets:

```
Food* f = malloc(sizeof(Food));
```

2) But just like anything on the stack...

- **Everything you malloc contains garbage**

3) To fix this you can use **memset** from `<string.h>`

```
memset(f, 0, sizeof(Food));
```

- This says "fill the memory at f with 0s"

4) orrrr use **calloc()** instead of **malloc()**

```
Food* f = calloc(sizeof(Food), 1);
```

- This says "allocate 1 * sizeof(Food) bytes and fill them with 0s"

Remarks:

1) It's garbage for the same reason the stack contains garbage: it's old, dirty, used memory

The Caveats:

1) If you do this:

```
while(1)
    malloc(1048576);
```

2) Your program will run out of memory

- But you won't get an error
- This will **loop forever** because...
- If you run out of memory, **MALLOC RETURNS NULL**

The Caveats (Continued):

1) and what's worse:

```
int* arr = malloc(sizeof(int) * 20);
free(arr);
arr[0] = 1000; // hmmm?
```

2) Who knows what'll happen!

3) When you free heap memory, **all pointers to it become invalid, and it's your responsibility to never use them again**

- So, if multiple things point to this memory...
- well, they're allowed to *point to it*, but they better not access it.
 - ⇒ It's like a bomb rigged to explode when you dereference it.

4) Here's another kind of **invalid pointer**: it *used* to point to some **valid heap memory**, but **not anymore**.

The **Heap** Commandments:

1. When you malloc, you *should* check if it returns NULL.
2. You must free everything you allocate.
3. ...and you must free it exactly *once*.
4. You must *not* access memory that has been freed.

these are the *really tricky* ones

how do you know how many things are using a piece of memory?

...can you guarantee that?

who is responsible for deallocating? (who's the owner?)

what if ten things point to it?

what if... WHAT IF NOBODY POINTS TO IT

VLAs (in C99+):

Variable-Size Convenience; Automatic Deletion!

1) Modern C code can **allocate variable sized pieces of memory on the stack**, instead of the **heap**

```
int len = strlen(str);  
char newArray[len + 1]; // not a constant size!
```

2) This is a C99 feature, called **variable-length arrays (VLAs)**

- It's still on the **stack**!
- Which gives you **automatic lifetime**!

3) Well these are great! why don't we use them for everything??

Well, You Can, But Watch Out:

1) On most systems, **stack space is limited**.

- On Unix systems like thoht, **ulimit -s** tells you the limit
- On thoht, it's only **10 MB**

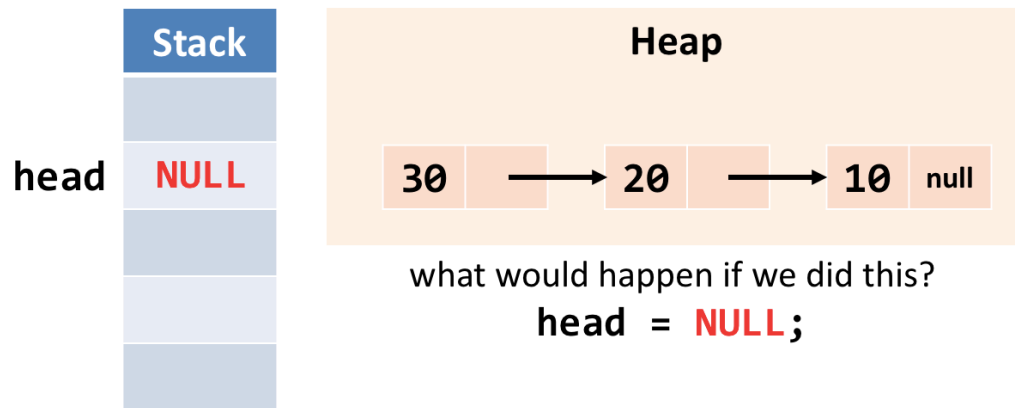
2) This doesn't solve the problem of "ownership."

- In cases where it's obvious that the data won't need to stick around after this function, **VLAs** work fine.
- But if you need to hand this pointer off to someone else... or if you need the data to stick around for longer...
 - ⇒ You're asking for trouble.

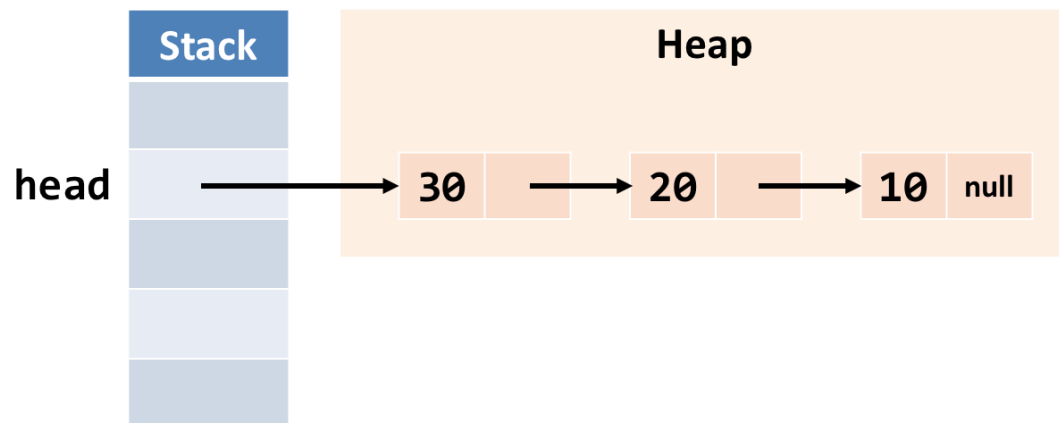
What's Garbage Collection?

Messy, Messy:

1) Let's say you had a **linked list**:

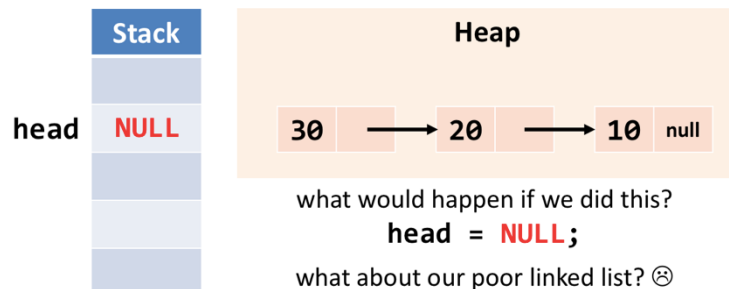


a) If the head is made **NULL**:



b) What happened to our **linked list**?

- Well nobody is pointing to that list anymore.
- Where does it go...? **NOWHERE**.



Dynamic Memory Management is Hard:

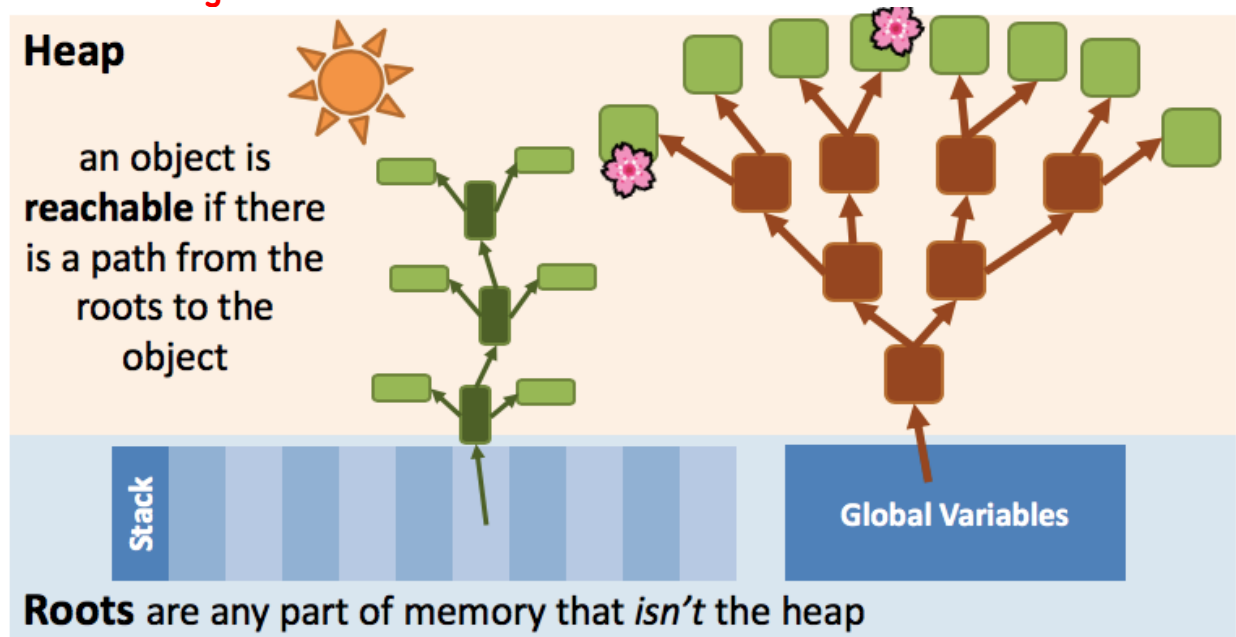
- 1) Don't believe anyone who says otherwise
- 2) But mathematically, it's not a very hard problem
 - if only we had some kind of automatic, programmable machine that could do it for us ☹

⇒ ...

- Oh wait! **WE DO!!**

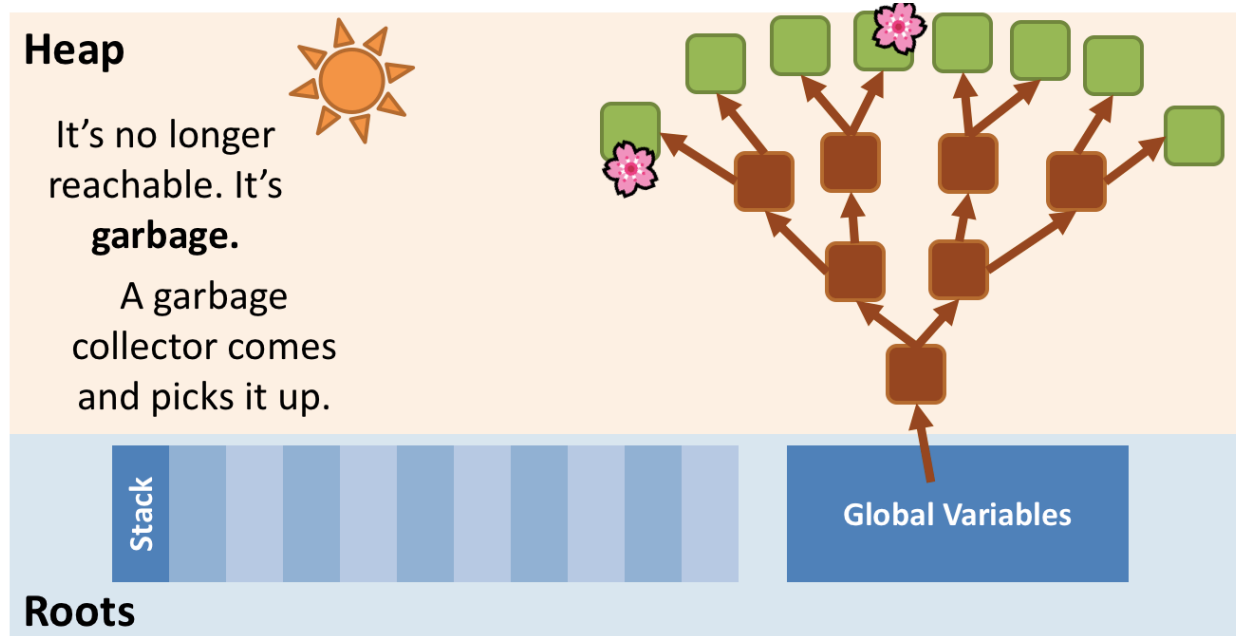
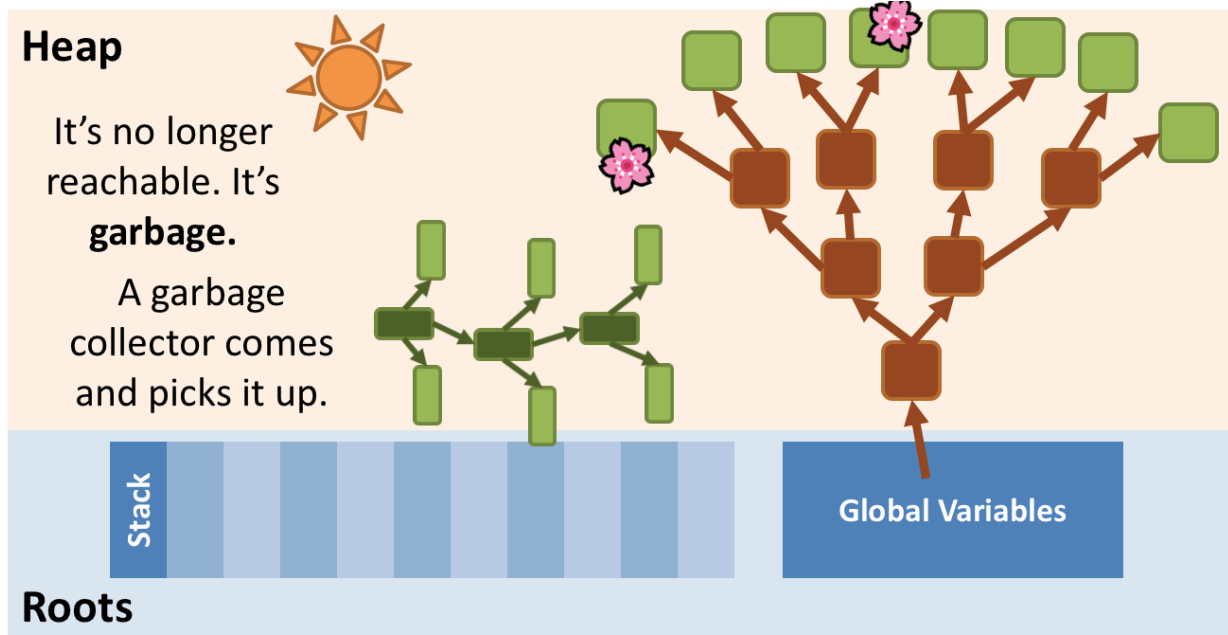
A Touching Story:

- 1) There are two useful concepts from **GC**: **roots** and **reachability**
- **GC = Garbage Collection**



Pruning Time:

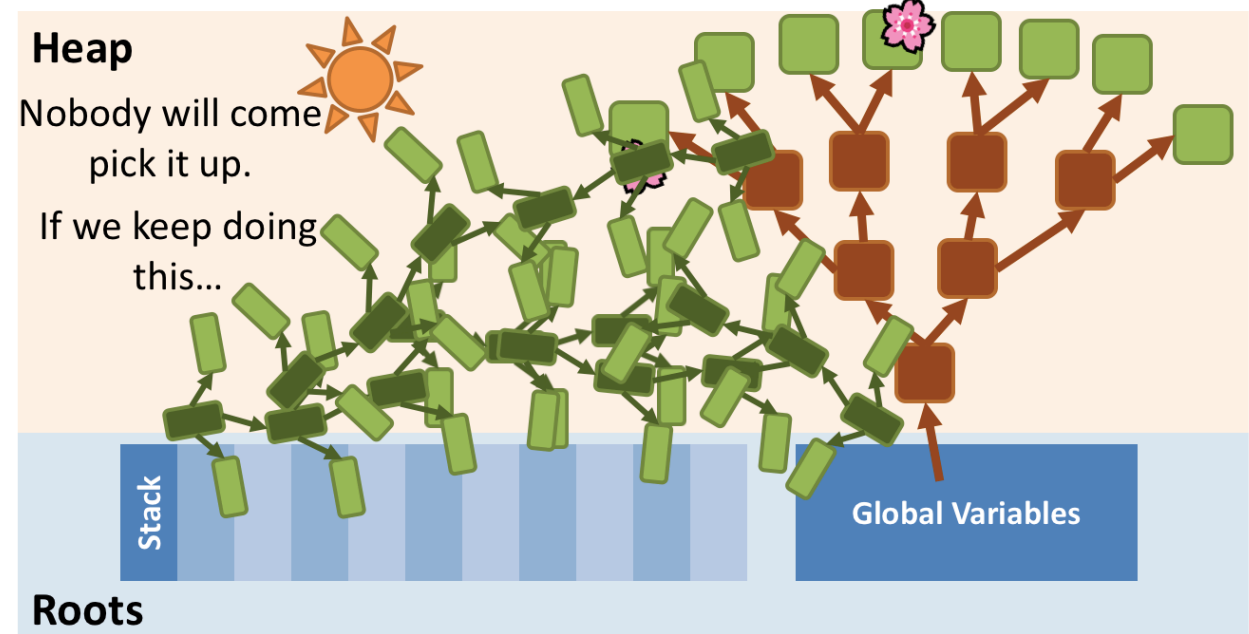
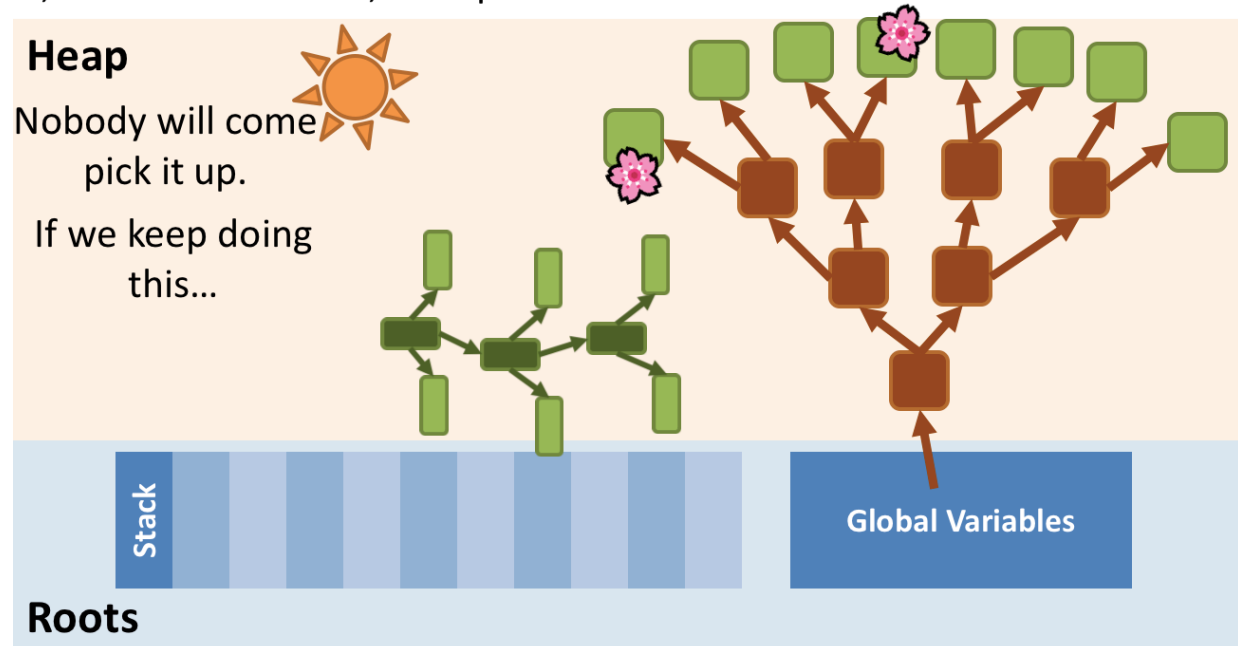
1) If we remove the only link to an object...



- We can cause a whole graph of objects to become **unreachable by removing only one link**.

Get a Rake:

1) C, on the other hand, is stupid. If we cut the last link:



This is a **MEMORY LEAK**:

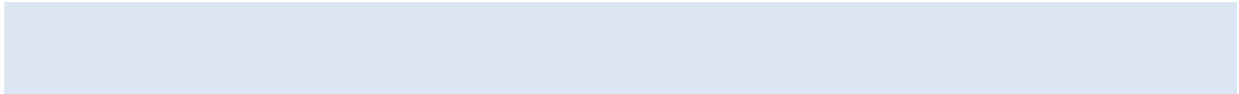
- 1) In C, if heap memory becomes unreachable, **it's a memory leak**.
 - *We can never deallocate it*. it's like it "leaked out" of your program.
- 2) The **only way to deallocate it** is to **exit your program**.
 - *All* your program's memory is deallocated when you exit.
- 3) Decades of research have gone into making **GC** useful!
 - **Never take it for granted!**
 - Well, now that you're using C, you'll learn not to...
- 4) There are other "**flavors**" of memory leaks
 - *"a cache without a removal policy is just a memory leak in disguise"*

OS - Memory Allocation:

How Does the Heap Work?

The Basic Idea:

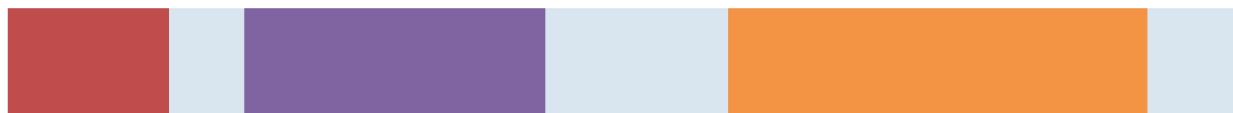
- 1) The **heap** is a **big piece of memory**



- 2) **Allocating memory (malloc)** slices off pieces of it



- 3) **Deallocating memory (free)** removes pieces, leaving **holes**



- 4) Now we want to allocate a new thing on the **heap**...



- Where can we put it? D:

Remarks:

- Unlike a **stack**, parts of the **heap** can become "**free**" in the middle, instead of only at the end

External Fragmentation:

- 1) "holes" - empty spaces that are too small to be useful - are bad
- 2) This is external fragmentation:
 - The free memory is split into lots of small fragments
 - External because the free memory is outside the allocated blocks



free space (blue) is scattered across the heap



free space (blue) is all together in one place

An Impossible Problem:

- 1) Given a finite space, and a number of items to fit into that space, find the optimal way to pack those items to minimize wasted space
 - This is the bin packing problem and it's NP-hard!
- 2) Dynamic memory allocation is an online variant of this problem
 - "online" means "we can't predict the future"

And Yet:

- 1) Life is about making things that aren't perfect, but are good enough
- 2) So, let's lay out the boundaries of the problem at hand:
 - When you free memory between two blocks, it leaves a hole
 - Holes are bad. (we don't want a swiss cheese heap)
 - We can ask the OS for more space if absolutely needed...
 - ⇒ ...but memory space is not infinite

The Memory Allocator's Job:

- 1) **Keep track** of what space is being **used** and what space **isn't**
 - The **memory allocator** can have "**secret**" information that the client program doesn't see
 - "**Used**" means "the client hasn't **freed** it"
 - ⇒ Whether the client is *actually* using it or not is another story...
- 2) **Allocate Memory** by finding unused (**free**) spaces
 - and of course, marking that space as **used**
- 3) **Deallocate Memory** by turning a **used space** into an **unused space**
 - So that it can be **reused**!

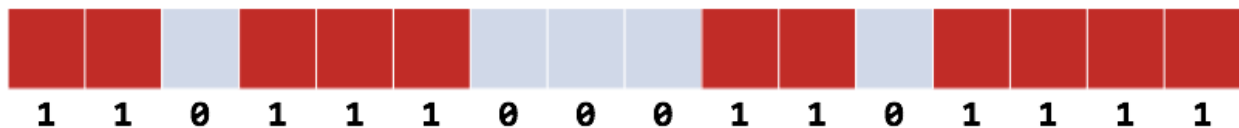
Bitmaps:

Movie Time:

- 1) You and your friends are seeing a movie.
- 2) There are 5 of you.
- 3) You get to the theater and...
 - How do you tell if a seat is taken?
 - ⇒ A seat is taken if someone is sitting in it
 - What are you looking for?
 - ⇒ You're looking for a block of 5 seats in a row.
 - How can you "reserve" a seat?
 - ⇒ You can reserve a seat by putting something in it, like a coat or purse or whatever.
- 4) A **bitmap** is like this.

Keeping Track:

- 1) If we divided up the heap into **uniformly-sized** chunks:



- 2) What's a simple way to say which are used and which are **free**?
 - Use a boolean (true/false) **per chunk**!
- 3) We can use one **bit** for each, and pack all those bits into one **int**
 - $11011100\ 01101111_2 = 0xDC6F$
- 4) This is a **bitmap**
 - The **bits** are a **map** of the heap
 - Bitmaps are forced to use **uniform size chunks**.
- 5) Let's say each chunk represents 64 bytes
 - 64×16 chunks = **1024 bytes** of memory
 - But the bitmap to represent it is only **2 bytes**
- 6) We have to store the bitmap elsewhere, but whatever.

Allocating:

1) The chunks are 64 bytes, and I do `malloc(100)`

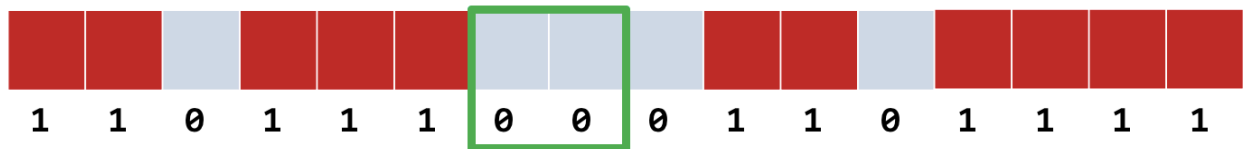
2) What do we have to do **with the bitmap** to find that **free space**?



3) We've got to find **at least** 2 0s in a row

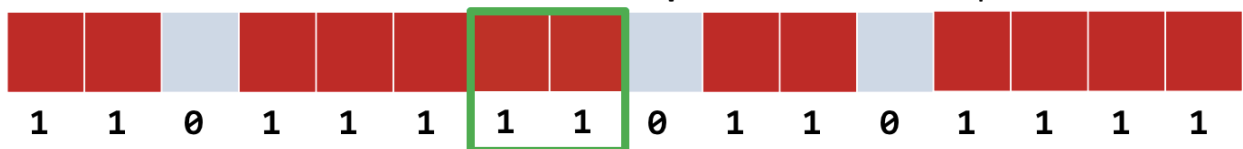
- Where's that?

- There it is!



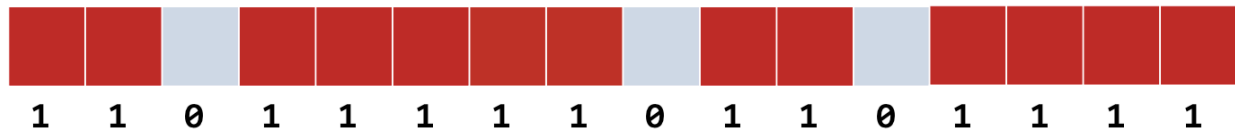
4) and then to mark them as **used**, we have to...

- Set those bits to 1s

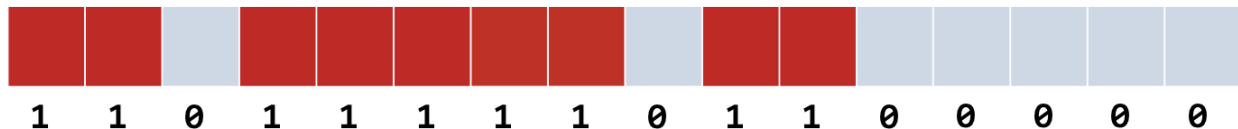


Deallocating/Freeing:

1) How do we **deallocate** (**free**) the **rightmost** group of chunks?



2) Just **set those bits to 0s**.



3) Now, the next time we want to allocate something that takes up 2 to 5 chunks, we have a **nice free space** for it.

4) So, what's the catch?

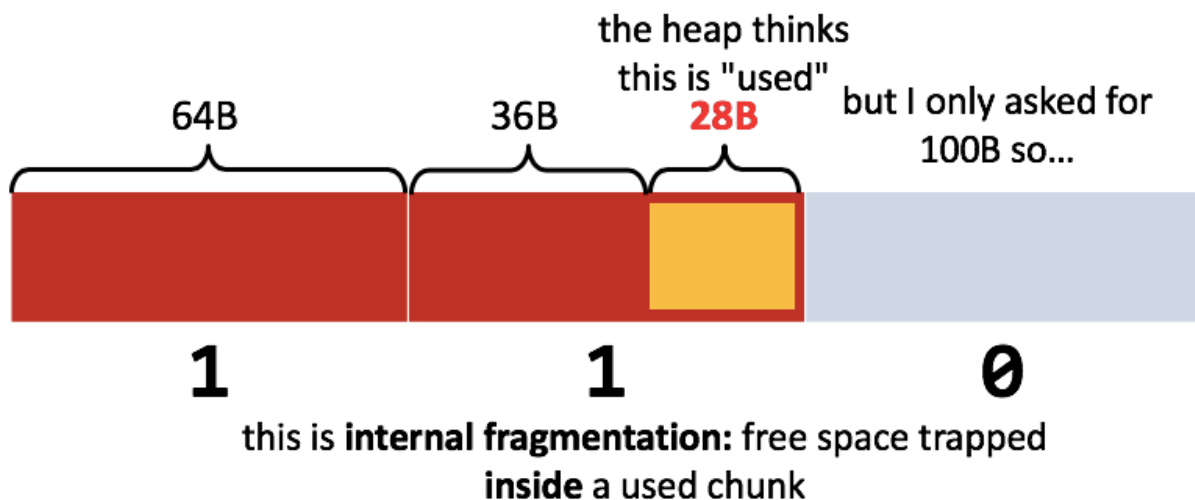
- Since the **bits are adjacent to one another**, whenever you set some bits to 0, you **"automatically"** get bigger blocks of **free memory**

A Glass Half-Full:

1) With **64-byte chunks**, when we did **malloc(100)**...

2) How much space did we mark as **"used"**?

- 2 chunks, or **128 bytes**



- It's like someone putting their purse on a seat and not sitting there. do you **REALLY** need a whole seat for your purse?

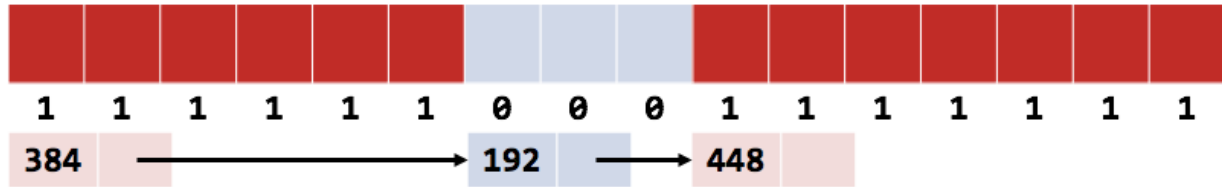
A Glass Full of Foam:

- 1) If we made the chunks smaller to better fit the allocation...
 - With 16B chunks, we could allocate 112B, only wasting 12B
 - But that means **four times as many chunks** in the same space
 - Which means **the bitmap is four times bigger**
 - Which means there's **less space to store interesting stuff**
- 2) No matter what, we're going to waste a lot of space **inside** chunks.
- 3) Also, "**find a sequence of n 0 bits inside an integer**" isn't very fast

Something Else:

Entropy:

1) What do you notice about the pattern of 1s and 0s here?



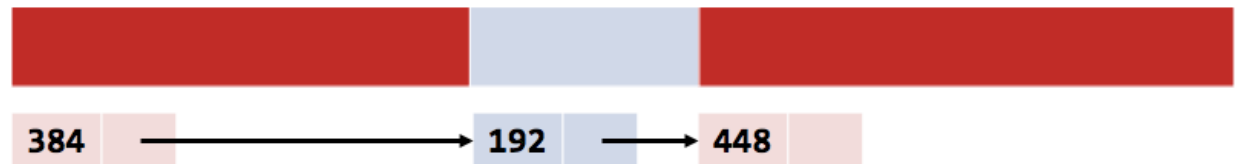
2) Seems silly to store a bunch of 1s, then 0s, then 1s...

3) What if we did *this* instead:

- Record the exact sizes of each contiguous block
- Then link em together... into a list

Did This Improve Anything?

1) How big was the bitmap for this? 2 bytes?



2) Now we have 3 linked list nodes

- Each of which has an int (4 bytes?) and a pointer (4 bytes?)
 - $8B \times 3 \text{ nodes} = 24B$ to store the nodes
 - 24 bytes vs. 2 bytes for the bitmap
- ⇒ this seems like a loss??

Welllll:

- 1) This looks bad for small examples, but for a bitmap...
 - With 64B chunks, a 1MB allocation would need **16,384 chunks**
 - ⇒ that's **2 kilobytes of 1s in the bitmap**
 - Could make chunks bigger to make the bitmap smaller...
 - ⇒ But that worsens internal fragmentation
- 2) **Linked List** avoids internal fragmentation by measuring in *bytes*
 - Always make a block **exactly** the size it needs to be
- 3) In the **worst case**, a **linked list** uses **more memory** than a **bitmap**
 - But in the **average** and **best** cases, it uses **far less**.

Remarks:

- 1) **Bitmap** grows linearly in the **number of bytes** used
- 2) **Linked List** grows linearly in the **number of allocations (blocks)**
- 3) Depending on the **average size of blocks**, the **linked list** or **bitmap** can be more **efficient** – but for **memory**, **linked list almost always wins**.

Pay the Piper:

- 1) **Linked Lists change size**. So, where do we allocate it?
 - We can **embed the list within the free and allocated blocks**



- 2) Every block starts with a **header** that includes the size, pointer, and its status (used or not)
- 3) So, a 100B block will take up maybe 112B total:
 - 12B for the header
 - 100B for the space the user asked for
- 4) The **allocator** is the only one who sees these headers.

Allocation Algorithms:

Tabula Rasa:

1) When your program first starts up, the **heap** looks like this:



2) If we want to start allocating stuff, **what's the most obvious way to allocate?**

- Slice off a part of the **beginning of the free space** each time

3) But we're using a **linked list**, and there are now **6 nodes** in it

Tabula... Not-So-Rasa:

1) In the middle of the program, the **heap** will look more like like this:



2) Now we want to allocate a block like this:



3) How do we find a spot where it will fit?

We're using a **linked list**...

- We could start at the **beginning** of the **heap**, looking for a spot

- When we find a spot **big enough**, that's where it goes

4) This is called **first-fit** cause it puts it in the **first** spot it **fits**.

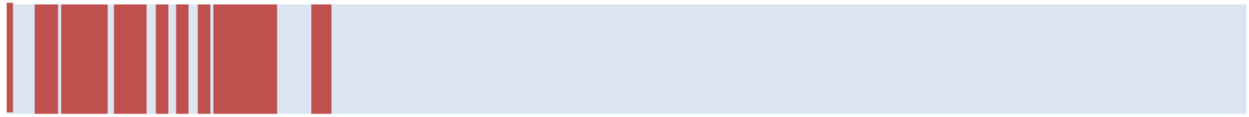
5) **how long does it take to find that free block?**

- It's **linear** in the number of blocks

- You might have to search through thousands of blocks each time...

The Bigger Picture:

1) With **first-fit**, the **heap** will often end up looking like this:



2) Everything's clustered around the beginning. this is good actually!

- Cause it leaves us with a nice big empty space

3) But...

- There are lots of **holes**

- The **holes** will likely be **small** and therefore **useless** (wasted)

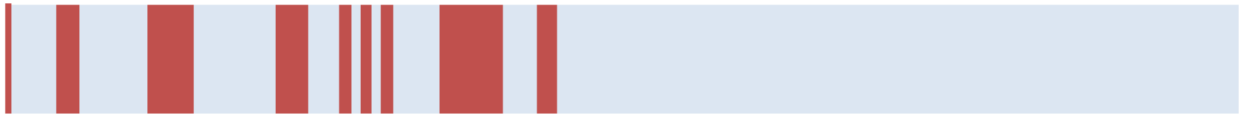
- We waste a lot of **time** looking through those **small holes** before we get to that **big empty space**

⇒ *why start from the beginning every time?*

Next-Fit:

1) What if we start looking from where we **last allocated** a block?

2) **Next-Fit** makes it faster to find a block, but spaces things out more:



3) Since it starts from the **last allocation**, it spreads things out and misses what might be **good spaces** earlier in the **heap**

A Square Peg in a Slightly Larger Square Hole:

1) Let's say our **heap** looks like this, and the orange block is the **most-recently-allocated** one:



2) Now we want to allocate this:



3) Where would **first-fit** put it?



4) Where would **next-fit** put it?



5) But what place(s) would make **more sense** (arguably)?

- Here, because the **free space** is **closest to the right size?**



This is **best-fit**.

6) Here, because it'll leave a bigger block of free space?



This is **worst-fit**.

The Best of Intentions:

- 1) **Neither of these schemes** really makes things any better
- 2) **Best-Fit** will leave **tons of tiny unusable holes**



- 3) **Worst-Fit** will "clump" allocations better, but tends to leave **several similarly-sized large holes that all gradually get smaller**



- 4) and worst of all, both of these algorithms force us to **look at every free block in the heap on every allocation**
– *how could we fix that?*

Remarks:

- 1) **Best-Fit** *can stop early* by *finding an exactly-sized hole* for the desired block, but **Worst-Fit HAS** to look at all the *free blocks*.

OS - Memory Deallocation:

Let's Revisit This:

You Can't Always Get What You Want:

1) Let's say the user did **malloc(60)**

a) Initially the block of memory looks like:



b) If we allocate a 100B block:



could we give them this
whole 100B block?

sure, but... why is that bad?

c) This results in **Internal Fragmentation: Trapped free blocks**



they only asked for 60, so
these 40 bytes are wasted.

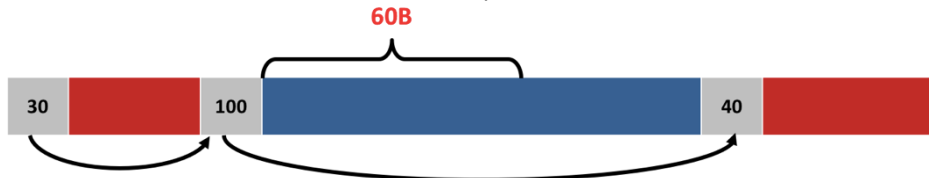
we want to avoid **internal
fragmentation.**

Splitting a Free Block:

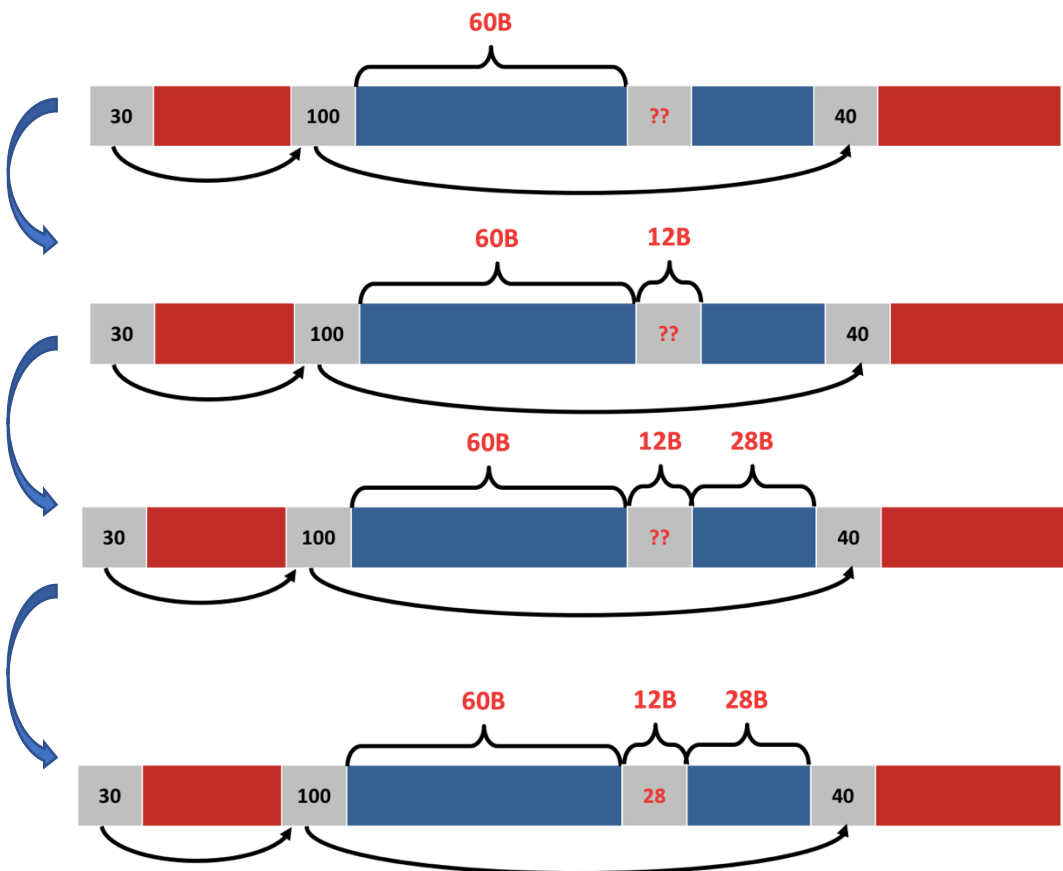
We want to give them **exactly** 60 bytes.



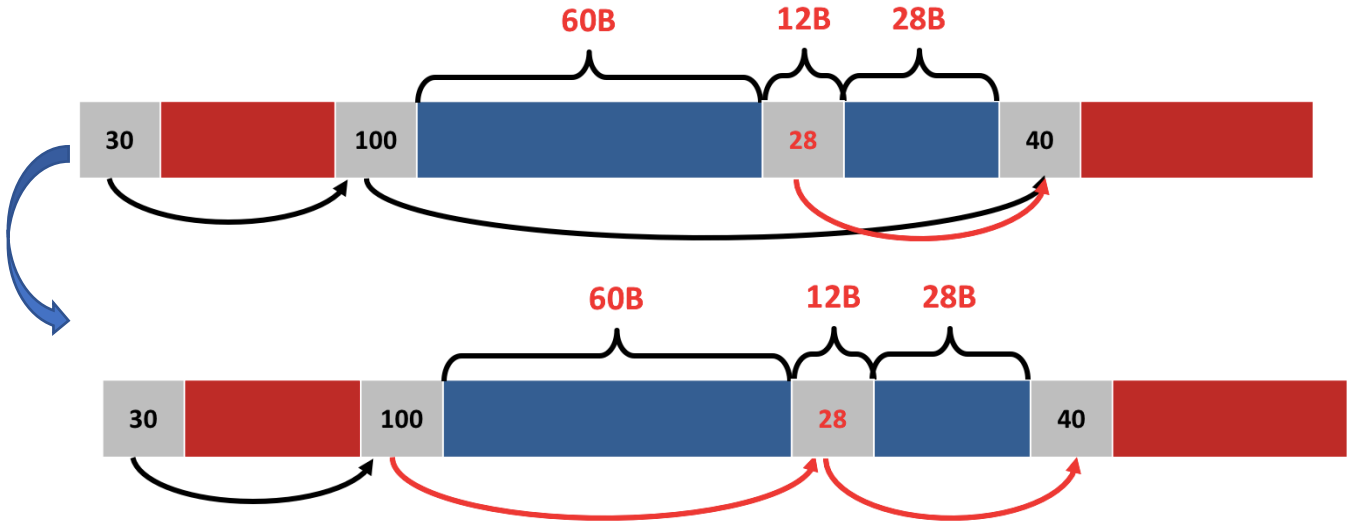
1) Count 60B into the block and put a **new header** there



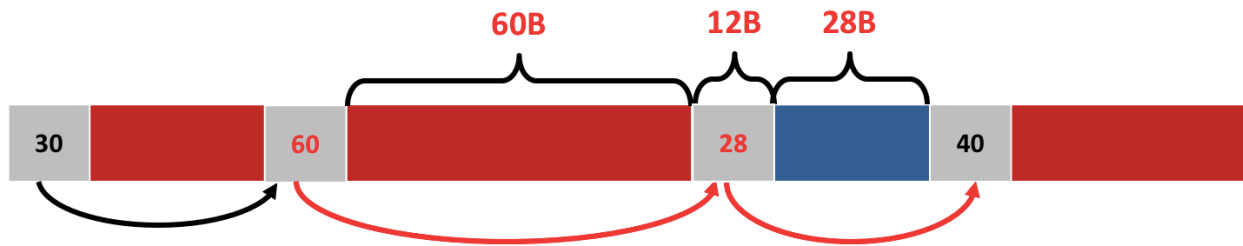
2) Do $100 - 60 - \text{sizeof}(\text{header})$ to get the size of the **new free block**



3) Insert the **new header** into the **linked list**



4) Update the **old header** to the **new size** and **mark it used**



Extending the Heap:

Mentioned Last Time:

- 1) The stuff on the **heap** is **entirely** managed by your program.
- 2) But the "**real estate**" that the heap occupies **comes from the OS**.
- 3) When your program needs more **heap space**...
 - You can ask the **OS to EXPAND HEAP**
 - it **may or may not** allow that to happen!

Heap Deallocation:

A Lot Easier:

1) Turning a used spot into a free spot is easier than finding one was.



the user wants to free the 100B block.



the user wants to free the 100B block.

ok.

OK, There's a Problem:

1) What if we end up with **two free blocks in a row?**



well, that seems kinda silly.

what if the user wanted 120 bytes now?

why don't we do the **inverse of splitting?**

- Even though we have **2 free blocks in a row** with **>120 bytes of space**, **we couldn't give it to them.**

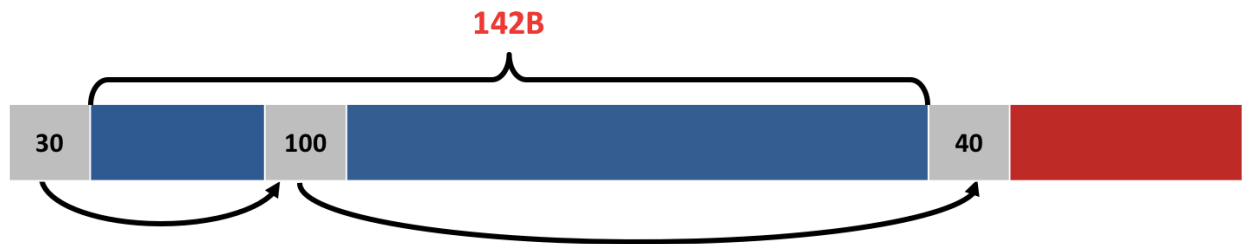
Coalescing:

If we end up with adjacent free blocks, we can coalesce them.

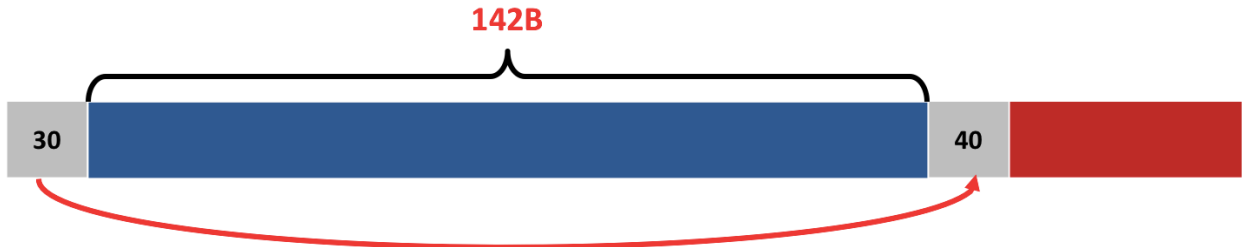
- coalesce = smoosh all together



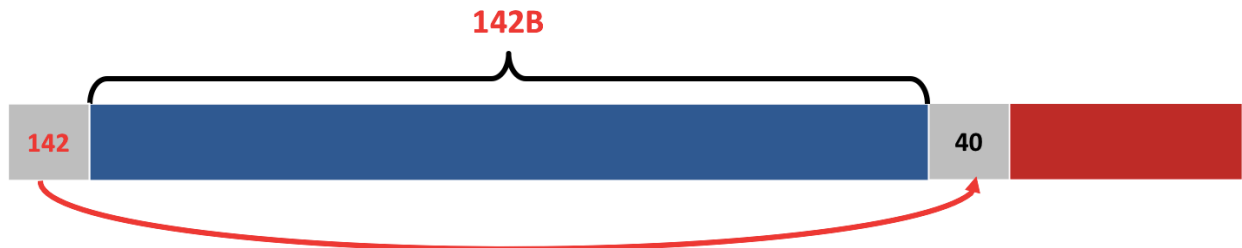
1) Sum the sizes of the two blocks + sizeof(header)



2) Remove the middle header from the linked list.



3) Update the first header with the new size.



Can You Have More Than Two Free Blocks in a Row?

1) Yep, if you free a block between two free blocks.



a) Coalescing the first two free blocks yields:



but this doesn't have to be a special case.

coalesce the first two...

b) Coalescing the new free block with the third free block yields:



but this doesn't have to be a special case.

coalesce the first two...

and then coalesce that with the third.

Remarks:

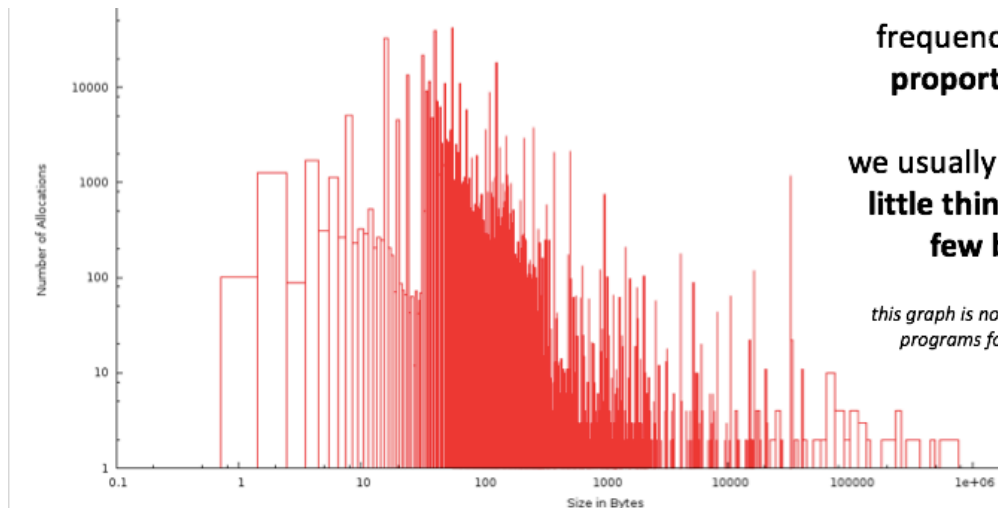
1) If you think about it, if you do this every time you free, you will never have more than 3 free blocks in a row.

2) Any case where you would have ≥ 4 blocks, it could only arise if you forgot to coalesce on a previous free.

Making Allocation Faster:

Expect the... Expected:

1) If you graph the **size of allocations** against the **frequency** of those allocations (how many blocks of that size are allocated) you get:



frequency is **inversely proportional** to size

we usually allocate lots of little things, and only a few big things

this graph is not universal – but many programs follow similar trends

Musical Chairs:

1) Think of it like... chairs of different sizes.

if someone small (a kid?) wants a chair, which chairs do we even *care about*?

the tiny chair.

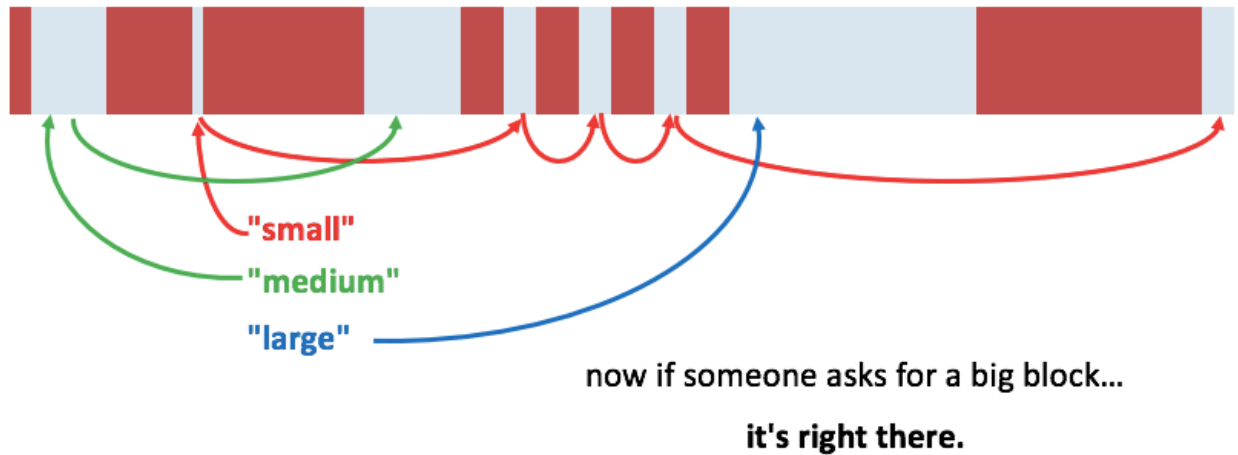
why bother looking at chairs that you *know* are the wrong size?



– That giant chair is **internally fragmented!!**

Quick-Fit:

- 1) Instead of using a *single linked list* for the *whole heap*....
- 2) Let's have *multiple lists*, **based on size**.



Throw it in the Bin:

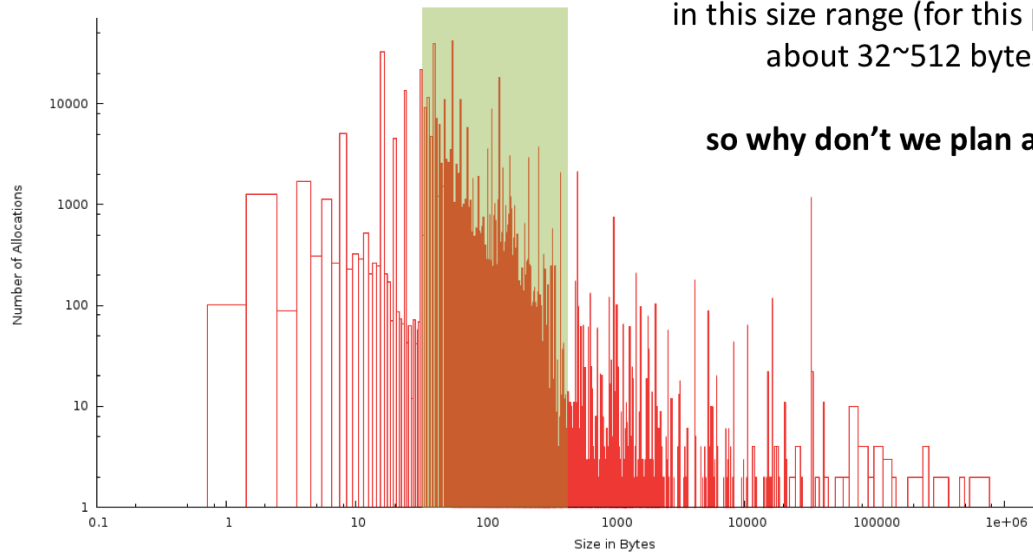
- 1) A *quick-fit* scheme keeps *several bins* of **varying sizes**
 - Maybe based on powers of 2
 - Maybe linearly spaced
 - Who knows
- 2) You can think of each bin as a "*set of free blocks* in that **size range**"
- 3) Allocation becomes much *faster*, as we can immediately know if there's a block of the "**right**" **size**
- 4) On *deallocation*, we insert the newly-freed block into the right bin
- 5) This is a pretty cool time-space tradeoff!

Hey!! This is like Radix Sort!!

Reducing External Fragmentation:

Expect the... Expected:

1) This graph again...



Adult Swim:

- 1) If our program is going to need 10,000 128-byte objects, let's just **preallocate 1.28MB of space** – we call this a **pool**
- 2) We can **very quickly slice new allocations** off the **pool**



- 3) When objects are deleted, this leaves nicely-sized holes



- 4) A **pool** paired with a **quick-fit** allocator can be **really fast**.
– but what's the downside?

You Can't Please Everyone:

- 1) This kind of memory pooling **doesn't work in the general case**
 - It works great for some programs and terribly for others!
- 2) The allocator either has to:
 - **Know about it in advance**; or
 - **Adapt dynamically**
 - ⇒ Which means doing **more work** during allocation
 - ⇒ Which means **slowing down** allocation
- 3) The default **glibc malloc** does *some* pooling by keeping pools of small objects *after* they've been allocated and freed
 - It tries to balance several factors, so it's kind of a well-rounded **memory allocator** without being the best at anything
 - ⇒ it's the **Mario** of **mallocs**

Heap Compaction??

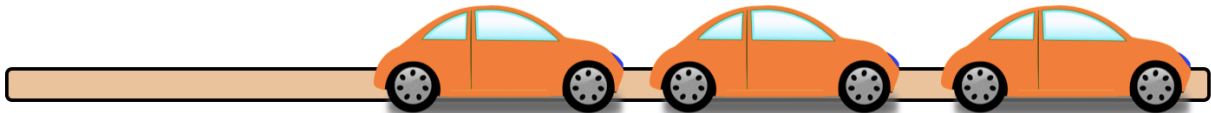
Move Down!!!

1) You're trying to parallel park



oh COME ON

why can't everyone just MOVE
DOWN cause THEN I'D HAVE
ROOM TO PARK



oh COME ON

why can't everyone just MOVE
DOWN cause THEN I'D HAVE
ROOM TO PARK

this is the idea behind
heap compaction

Intractable:

1) Let's say we have this situation:



head

if I want to move that block to the left, what do I have to do to the **head** pointer?

I have to change the pointer, too.

I have to change every pointer that points to that block.

this is completely unworkable.

but remember: every problem in CS can be solved with another level of indirection...

Tractable:

1) Instead of pointing *directly* to the heap, we make **EVERY** pointer...



...point to a pointer.

now we can have unlimited pointers to that pointer



...point to a pointer.

now we can have unlimited pointers to that pointer

and if we want to move the block, we update the **one** pointer in the table.

what are the downsides?

Remarks:

- 1) It's **slower**
- 2) It requires you to only **access the heap** through this **double-pointer mechanism**
- 3) You cannot rely on things **"staying put"** (which is sometimes important)

Wellllll Guess What, We Can't Do it in C:

- 1) **Java** does this...
 - because **they designed it from the start** to be able to do this.
- 2) Doing this requires:
 - **Perfect knowledge of where every block is**
 - **No pointers** to the **"insides"** of blocks
 - **No access** to the **heap** *without* these double-pointers
- 3) I guess you *could* do this in C if you were **reaaally** disciplined
 - But **there's absolutely nothing preventing you from messing up.**

C – Multi-File Development and **make**:

Efficient Function Calling:

1) If you have three functions to run in a row, it's tempting to do:

```
void c() { /* c stuff */ }  
void b() { /* b stuff */ c(); }  
void a() { /* a stuff */ b(); }
```

2) So, **a** calls **b**, and **b** calls **c**.

3) **Don't do this.**

4) Instead, do this:

```
void c() { /* c stuff */ }  
void b() { /* b stuff */ }  
void a() { /* a stuff */ }  
void caller() { a(); b(); c(); }
```

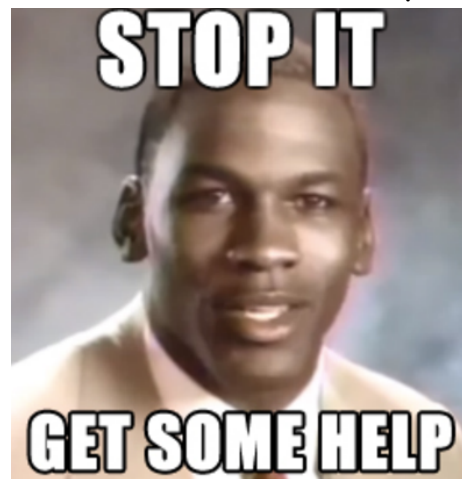
Also **Stop** Putting **Linked List** Crap in Your Higher Level Functions, Okay??

1) Almost all the special cases in the allocator come down to linked list manipulation

2) **Abstract it**

3) **Call it from other functions**

4) **It will make your life so much easier**



The Preprocessor:

- 1) Compilers were really, really dumb and simple programs
- 2) An easy hack was to use a **preprocessor**
- 3) It's a **text processing system** that can do *textual* substitutions
 - **it's automated copy and paste.**
- 4) You can see the results of preprocessing with **gcc -E**
- 5) You tell the preprocessor what to do with **directives**
 - These are the lines that start with **#**

The **#include** Directive:

- 1) **#include** is *so* stupid
- 2) It copies and pastes the **entire contents of the given file** right there

myheader.h

```
int foob();  
void blarb();  
const int F = 10;
```

program.c

```
#include "myheader.h"  
  
int main() {  
    return 0;  
}
```

myheader.h

```
int foob();  
void blarb();  
const int F = 10;
```

program.c

```
int foob();  
void blarb();  
const int F = 10;  
  
int main() {  
    return 0;  
}
```



<angular> or "quoted":

1) .h stands for "header"

2) #include <filename.h>

- Means to include a standard library header

⇒ or sometimes an OS header

3) #include "filename.h"

- Means to include some other header

⇒ Your headers, third party libraries, etc.

4) What's in these .h files???

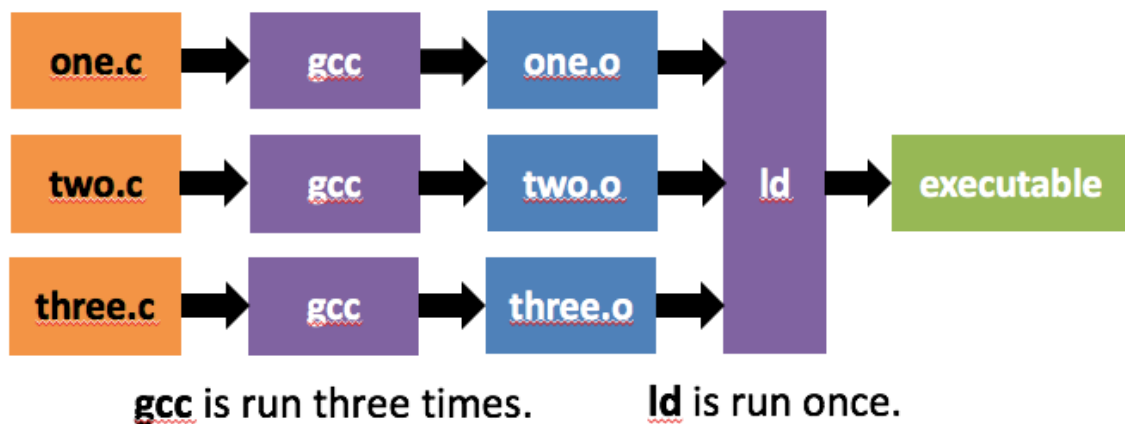
- We'll see shortly.

5) and that's all we'll say about the preprocessor for now.

Multi-File Compilation:

The Old Ways:

- 1) Each C source file is compiled **independently**
- 2) C calls this a **translation unit**: one source file → one object file
- 3) Multiple translation units are then **linked** to make one program



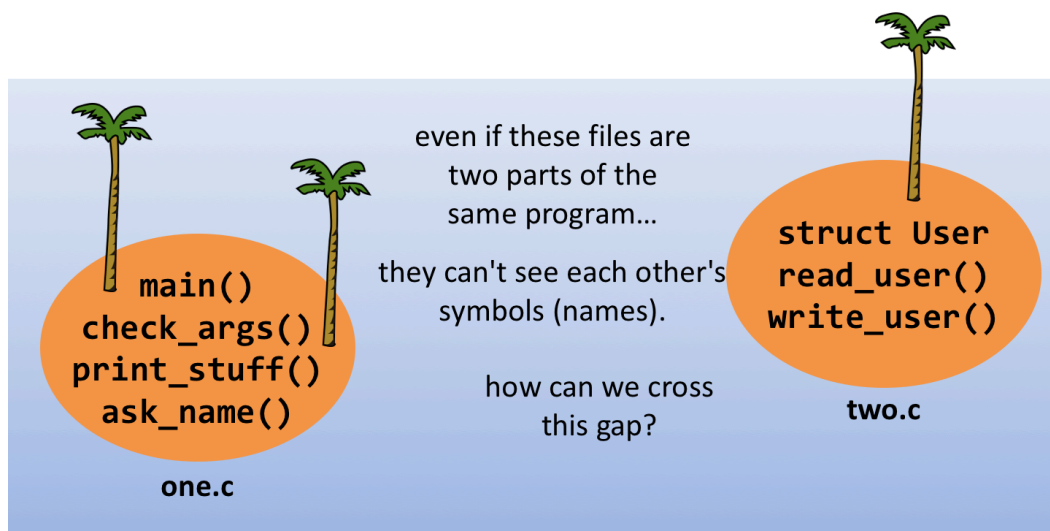
even if you write `gcc one.c two.c three.c`.

Remarks:

- 1) As you might imagine, running `gcc` 1000 times to compile 1000 files is **pretty slow**.

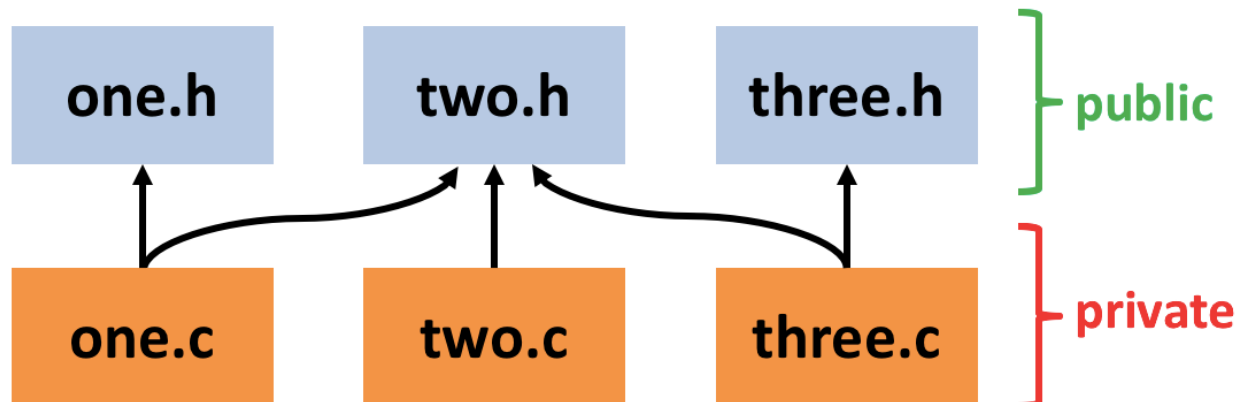
Never the Twain Shall Meet (until **linking**, anyway):

- 1) Each `.c` file is like its own private island



Headers!

1) each compilation unit usually has a **header file**



2) The header is a compilation unit's **public interface**

- ...and the source file is its **private implementation**

3) Each source file includes its own header

- and they can include headers of **other** compilation units

4) Each source file can **only see what other headers expose to it**.

The General Form a Header:

1) If you wanna write **users.h**, it would look like:

```
#ifndef _USERS_H_
#define _USERS_H_
```

```
// all the contents!
```

```
#endif
```

2) What the heck is that preprocessor crap

- Let's take it out and use **gcc -E** to see what happens if we *include this file twice*

3) This preprocessor stuff is called an **include guard**

- IDEs and stuff will put one there for you

Remarks:

- Without the **include guard**, if you end up accidentally including a header twice (such as through nested includes), you will get redefinition errors

- There is also **#pragma once** which is supported by virtually every compiler and virtually no one uses it

Header **Do's** and **Don'ts**:

Put this in the header	Don't put this in the header
<ul style="list-style-type: none">○ public function prototypes○ structs○ enums○ typedefs○ #defines (constants, macros)	<ul style="list-style-type: none">○ function definitions (code)○ private function prototypes○ variables (ever!) <p>all this stuff goes in the .c file.</p>

Remarks:

- 1) Basically, headers contain all the non-code non-private bits.
- 2) If you don't want someone else to use it, don't put it in the header.

Simple Shell Scripts:

Who's Tired of Typing `gcc -o` blah blah...

- 1) A **shell script** is a file containing a **list of shell commands**
 - In fact, *it's a whole programming language*
- 2) It's a text file whose name ends in **.sh** and contains the following:

```
#!/bin/sh
...commands...
```

- 3) The first line is called the **shebang**
 - **#** is **hash**, **!** is **bang**
 - ⇒ *ssssssshhhhhhhebang*
 - It says which program to execute this script with
- 4) **/bin/sh** is almost always what you want
 - But you could e.g. make a python script and use **/usr/bin/python**

build.sh

- 1) If you've got a very small program, maybe something as simple as this will be sufficient for building stuff:

```
#!/bin/sh
gcc -Wall -Werror -g --std=c99 -o myprogram file1.c file2.c
```

- 2) Once you create a file like this, you have to make it **executable**
 - Use **chmod** to **change** the **mode** of the file:

```
$ chmod +x build.sh
```

- 3) Now you can run it like any other program!

```
$ ./build.sh
```

Remarks:

- 1) If you find yourself typing any complex command over and over, make yourself a **shellscript** like this

Command-Line Arguments:

- 1) Sometimes this is **useful**
- 2) **\$1**, **\$2**, **\$3** etc. are **argv[1]**, **argv[2]**, **argv[3]** etc.
- 3) **\$@** will be replaced by all the command-line arguments
- 4) So, I could write a more flexible **build.sh** like so:

```
#!/bin/sh  
gcc -Wall -Werror -g --std=c99 -o myprogram $@
```

- 5) Now, I can run it like:

```
$ ./build.sh file1.c file2.c
```

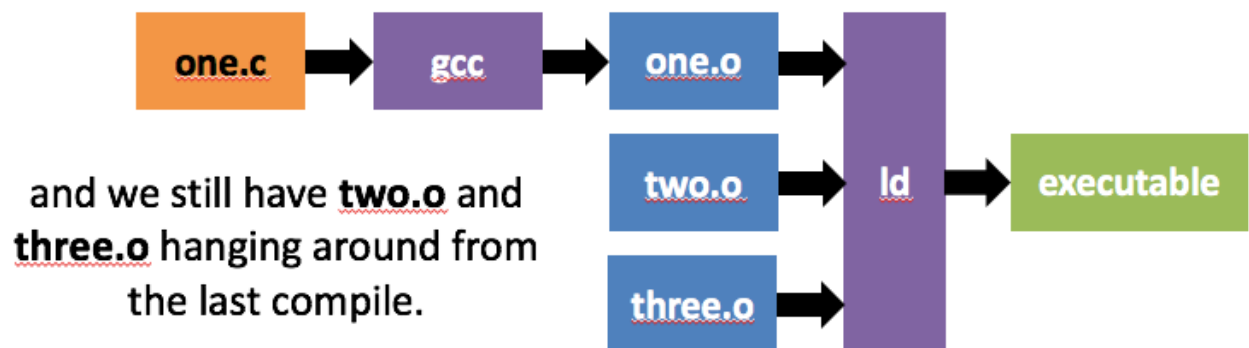
But For Anything More Complex...

- 1) Don't learn bash. **seriously**.
 - It is an awful hack of a programming language.
- 2) Use a real language, like Python or Ruby or whatever.
 - All you have to do is put the interpreter path in the shebang.
 - Use **which programname** to find the path to any program.

make:

Incremental Compilation:

- 1) **Compilation** and **Linking** **actually take time**. sometimes **a lot**.
 - 2) Repeating compilation of *unchanged* files is a waste of time
 - 3) **Incremental compilation** *only recompiles the sources which have changed*, while reusing previously-compiled object files
- Say we only changed **one.c**...



- This is basically a necessity for anything bigger than the tiniest toy programs.

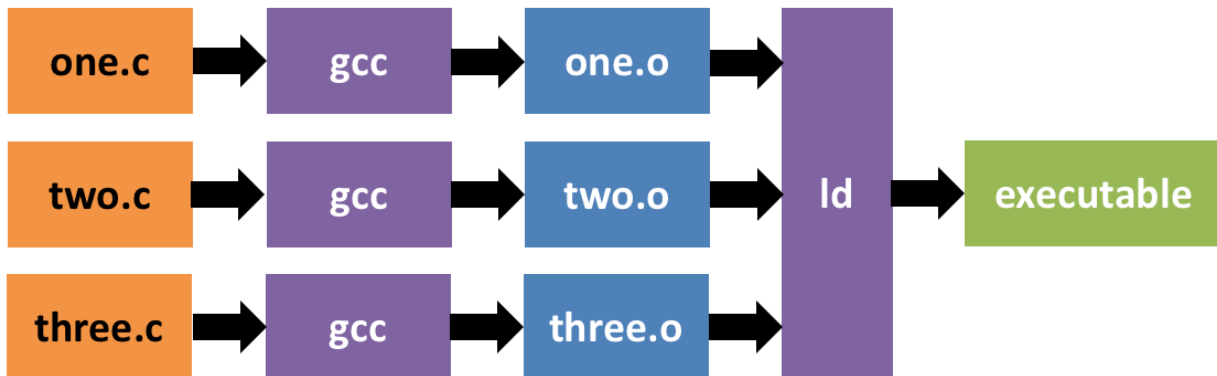
Build Tools:

- 1) A **build tool** is a program that simplifies building programs
 - 2) **make** is the classic build tool
- Other examples are **cmake**, **scons**, **ant**, **cargo**...
 - 3) Build tools can help you compile...
 - **Only the file you changed** (incremental compilation)
 - **Multiple versions** (debug, release, 32-bit, 64-bit...)
 - A **whole directory** without listing every file
 - 4) and they can handle other steps, such as...
 - **Converting** data files between formats
 - Setting up **operating system-specific** files (icons, resources etc.)
 - **Installing** your program

They're **super flexible**, is what I'm saying.

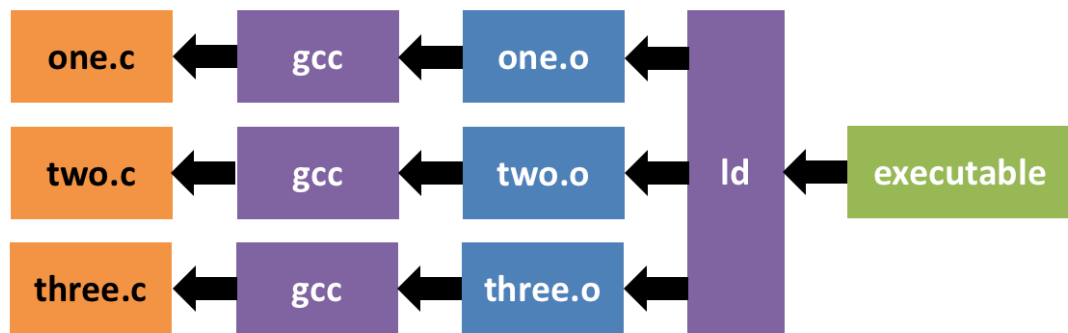
It Depends:

1) Build tools are based on the idea of **dependencies**



OR

● build tools are based on the idea of **dependencies**



each arrow says "the thing on the right *depends on* the thing on the left being done first"

"to build *executable*, we must link it. to link it, we must have these three object files..."

make is a Programming Language:

- 1) I mean, why not, right? :P
- 2) It's a **declarative** language – you describe *what*, not *how*.
- 3) A **makefile** consists of several **rules**
- 4) Each **rule** contains a **recipe** – the commands which satisfy the rule

let's consider this situation:
we want to go from a .c file
to a .o file



so we might make a rule saying: **.o files depend on .c files.**

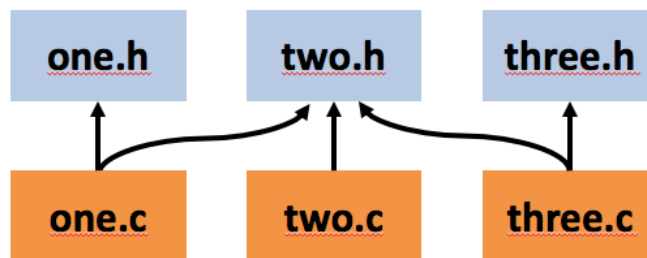
and the **recipe** – what to do
– is to run **gcc**.

Remarks:

- 1) Well, you describe "**how**" but only as parts of "**what**" declarations
 - 2) The counterpart of declarative is **imperative** – you describe how, step-by-step, rather than what
- All the languages you are likely to have encountered so far fall into this category.

More Complex Dependencies:

- 1) Of course, things are usually more complicated



if I change **one.h**, **one.c**
needs to be recompiled.

if I change **two.h**, **one.c**,
two.c, and **three.c** all need to
be recompiled.

fortunately, **make** handles all this crap for us.

- It knows the dependencies because **gcc** has options for **extracting the dependencies**

Dependencies and Targets:

1) The way we indicate dependencies is with this syntax:

target: dependency *dependency dependency...*
→ recipe commands...

make *requires* hard tabs for indentation!

2) The **target** is the thing being built

3) The **dependencies** are what it needs to exist before this rule is run

4) The **commands** are run to create the target from the dependencies

- They are just normal shell commands.

Remarks:

1) Your text editor's syntax highlighter will take care of the tabs for you, probably.

Generic Targets:

1) We can make *all* **.o** files depend on **.c** files with:

%o: %.c

2) Then for the **commands**:

- **\$<** refers to the **dependencies**

- **\$@** refers to the **target**:

%o: %.c

gcc -Wall -Werror --std=c99 -g -c -o \$@ \$<

3) **gcc -c** says "produce an object file instead of an executable."

4) You can remember **\$@** by thinking it looks like **\$☉***

An Example **makefile**:

- 1) The **makefile** has to be named **Makefile**, with a capital M
- 2) You can have many **targets** in one **makefile**
- 3) A very common target is **clean**, which cleans up build results
 - Things like object files, temporary directories, etc.
- 4) To make a target, just run

```
$ make targetname
```

5) Like:

```
$ make clean
```

```
$ make coffee
```

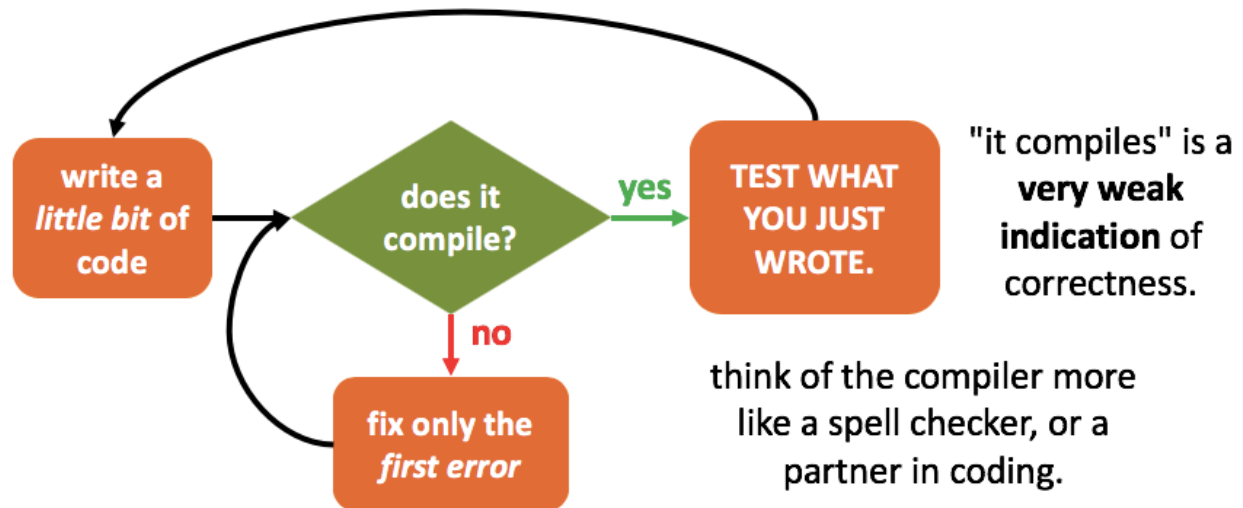
```
$ make my_dreams_come_true
```

Programs - Debugging:

Writing Fewer Bugs in the First Place:

Removing the Training Wheels:

- 1) As your programs get bigger...
 - You **cannot** write them all at once and *then* compile and test.



- 2) The **faster you can make this cycle**, the **faster and more correctly you can write code**.

Writing Smaller Functions:

- 1) Smaller functions have fewer things to go wrong
 - 2) They **have more well-defined purposes**
 - 3) It's **easier to encode assumptions (inputs and outputs)**
 - 4) It's **easier to see the control flow**
 - 5) It's **easy to see all the places they're used**
- It is **totally okay** to write a function that is called exactly once.
 - **Functions label your code**. they **indicate your intent**.

Wall the Werrors:

- 1) *Always try to use -Wall -Werror with gcc.*
- 2) In any other language, turn on **all the error checking you can.**
- 3) While developing, **enable debugging info.** (gcc -g)
- 4) Your brain is squishy and can't handle everything.



WRITE TESTS:

- 1) Ideally, you write a test for each of your small functions

Good things to test
normal conditions
correct outputs
weird inputs
error conditions
"contracts"
effects on state

state is, broadly, "the variables and environment *outside* of the function."

look into **testing frameworks** for your language. most have them.

Using the Right Programming Language:

1) All languages are more robust in some areas than others.

If you're doing...	Then try...
lots of strings!!	<u>Lua</u> , Ruby
daily tasks, file management	Python
databases	C#, Python
GUIs	C#, Ruby?
low-level stuff	Rust, C
multithreading	Rust, Haskell
theoretical math	Haskell, <u>OCaml</u>

these recommendations are based on personal and vicarious experience; there are many languages, and many factors go into language choice.

Inputs:

Good

Bad

Results:

Succeeded

V

X

Failed:

X

V

So, Something Went Wrong:

Step 1: Admitting You Have a Problem:

- 1) A **crash** is an obvious indication that something is wrong.
 - In C, you can use a debugger (**gdb**) to get a stack trace.
- 2) But many problems *don't* lead to a crash...

in that case, you need to **follow
the control flow**

at *some point*, your
program goes from
"working correctly" to
"something is bad."

your job is to find that point; **do
not try to *guess*** based on the
output or the final results.

- 3) Programs are proofs (thanks, Howard-Curry isomorphism!)
 - and like a proof, you can make mistakes in your assumptions or proof rules; you can do great for several steps and then oops.

`printf("x = %d\n", x);`

- 1) You can absolutely use prints or **logging frameworks** (e.g. Log4J in Java) to spam informational messages as your program runs
 - Logging frameworks let you have a "severity" for each message
 - "Debug", "info", "warning", "error", "fatal"
- 2) If you don't *already* have prints, using a debugger to follow control flow is probably easier than writing a bunch in there.
 - Especially since you have to remove/comment them out later.

Step 2: Determining the Cause:

- 1) Once you pinpoint the location...
- The cause may or may not be obvious.

```
Rectangle r = get_bounding_rect();  
int x = do_a_thing(r.x, r.y, r.x + r.width, r.y + r.height);  
int y = smth_else (r.y, r.x, r.y + r.width, r.x + r.height);
```

copy-and-paste errors are often hard to spot.
splitting code into **smaller functions** can help avoid this.

missing cases are another common problem – and you don't see them cause they're *not there*.

```
if(x < 0) {  
    do_one_thing();  
} else if(x > 0) {  
    do_another();  
}
```

- We probably shouldn't be adding the y coordinate to the width... we probably meant to swap width and height too.
- What if $x == 0$?

Check Your Assumptions:

- 1) Assumptions are "shoulds"
 - "When I enter this function, what *should* the parameters be?"
 - "When I return this value, what *should* it look like?"
 - "After this search loop, what *should* be in this variable?"
 - "When this method runs, what state *should* the object be in?"
- 2) **Assertions** are a great way to catch bugs early during development
 - Java has `em`. C has `em`.

```
#include <assert.h>  
...  
assert(x >= 0);  
assert(self.state == READY);
```

assertions should only be used to test for things that should be *absolutely impossible to happen* under any circumstance.

don't e.g. use them to check parameters on a public method.

Working Backwards from Crashes:

1) When a program **crash**...

- Fairly often, that specific line **is not at fault**.
- It's because some *earlier code* broke things, and your program kept running for a while until things finally crashed.
 - ⇒ Like a ticking time bomb.

2) **Look at what happened immediately before.**

- Set breakpoints earlier than the crash and re-run the program.
- Check out what's happening at those breakpoints.
- Then step closer and closer to narrow down the actual error location.

Finding Bugs Faster:

IDEs:

1) IDEs can *continually* compile and find errors

- They can make suggestions to fix them
- or show help on functions while you write so that you don't make mistakes to begin with

2) IDEs *can* be a helpful tool...

- but they can also be "helpy" - "I'M BEING HELPFUL!!!!!!"
- They can also be *so* complex that you just get overwhelmed by all the settings and features.

3) Probably the nicest feature of IDEs is an **integrated debugger**

- No need to use arcane typed commands

Debugger Breakpoints:

1) A **breakpoint** lets you pause the program and look around.

2) In gdb:

- **b func** will pause whenever **func** is called.
 - **b mymalloc.c:45** will pause when line 45 of mymalloc.c is reached.
 - **b *0x8004030** will pause when the PC gets to address 0x8004030.
 - **b location if x == 5** will pause at a location but only if the condition is satisfied.
location can be any of the above.
 - **tb location** is a breakpoint that only happens once – it's deleted after the first time it's hit. (you can make these conditional too.)
- 3) Check out the breakpoint features in your IDEs!

Watchpoints:

- 1) Sometimes a variable gets set to some weird value and you don't know where or when it happens.
- 2) A **watchpoint** is like a breakpoint, but it pauses *whenever a variable is about to be written* (or read, or both).
- 3) In gdb:
 - **watch *globalvar*** will pause when a global variable is changed.
 - **watch *localvar*** will only work when you are paused in a function, and it will last until the local variable goes out of scope.
 - **rwatch** and **awatch** work the same, except they pause when a variable is read (rwatch) or on all accesses (awatch).
- 4) You can't set a condition on these, sadly...

Programs: Calling Conventions:

Calling Conventions:

So, I just met you I'm sorry:

- 1) The stack holds **activation records**; calls push an AR; returns pop
 - but **what about the rest of the machinery?**
- 2) Remember what's in an AR?
 - **return value**, **local variables**, **arguments** (a kind of local variable)

parameters?

stack pointer?

return values?

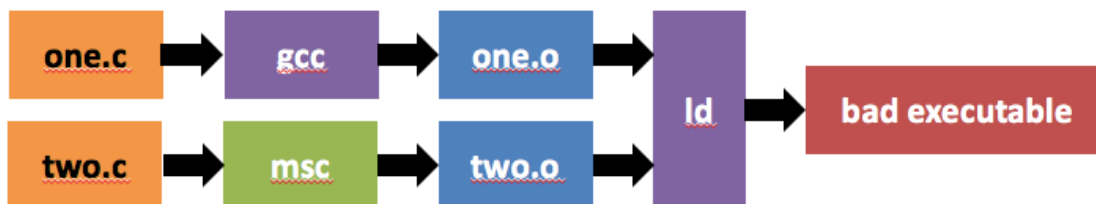
contracts?

control flow?

these and other questions are
answered by **calling conventions**

What is a Calling Convention?

- 1) It's how functions call one another... in **machine language**.
 - They are an **honor system** to make functions work together.
- 2) One of the **reasons** we made **HLLs** was to abstract this machinery
- 3) But when dealing with **low-level software**, it inevitably comes up.



different compilers/languages can have *different calling conventions*, even on the **same CPU architecture!**

Something in `one.c` tries to call something in `two.c`, **but if they use different conventions**, it'll **crash at runtime**.

Universal Truths:

1) You're pretty much always gonna have...

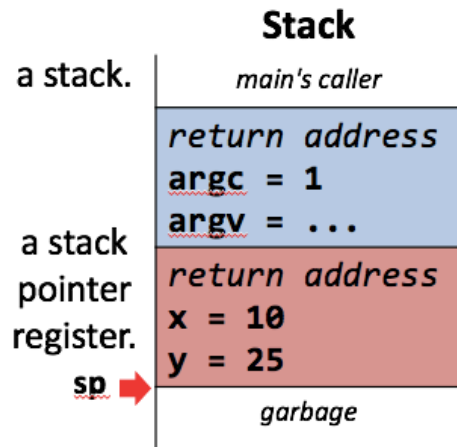
```

int main(...) {
    f(10, 25);
    return 0;
}
int f(int x, int y) {
    return x + y;
}

```

pc →

a program counter register.



other registers to deal with.

rcx ebp h1
t2 g0 r7

The General Idea:

A **function call** consists of...

- 1) Putting the **arguments** and **return address** in the right place(s)
- 2) **Jumping** to the new function
- 3) Setting up the **activation record** on the stack

A **function return** consists of...

- 1) Putting the **return value(s)** somewhere
- 2) Cleaning up the **activation record** from the stack
- 3) Jumping back to the **return address**

How MIPS Does It:

let's translate this call. `f(1, 2, 3, 4, 5, 6)`

the first 4 arguments are put
in the `a0..a3` registers.

any more arguments are pushed onto
the stack in reverse order. (why?)

`jal` both jumps to the new function and puts the
return address in the `ra` register.

```
li a0, 1
li a1, 2
li a2, 3
li a3, 4
add sp, sp, -8
li t0, 6
sw t0, 4(sp)
li t0, 5
sw t0, 0(sp)
jal f
```

Because the stack grows down, pushing them in reverse order means that they will be *in order in memory*.

On the Callee's Side...

1) Once we're inside, we have more to do

push the return address on the stack.

push any *callee-saved registers*
this function needs.

the remaining 2 stack slots are for **locals**.

where are the 5th and 6th arguments
relative to `sp`, now?

f:

```
add sp, sp, -20
sw ra, 16(sp)
sw s1, 12(sp)
sw s0, 8(sp)
sw zero, 4(sp)
sw zero, 0(sp)
```

Function Prologue

all the stuff that
happens *before your*
function even starts
running.

2) 5th and 6th arguments are at `sp+20` and `sp+24`

3) MIPS does have another register – `fp` – which is used to deal with this more
elegantly, but it is not strictly required for proper operation.

– *proper operation. properation.*

Pack it Up:

1) Then, to return...

place the **return value** in **v0**.
pop the previously-pushed registers.
clean up the stack, including the 5th and 6th
arguments that the caller pushed.
jump to the return address.

```
add v0, s0, s1
lw  ra, 16(sp)
lw  s1, 12(sp)
lw  s0, 8(sp)
add sp, sp, 28
jr  ra
```

the caller can inspect the return value
by looking at **v0**.

Function Epilogue
all the stuff that
happens when you
write **return** or get to
the closing brace.

2) The stack must be *balanced* – at the **same position** before and after the call.

Crash Course on IA-32:

x86? IA-32? x86-64? x64? AMD64? EM64T?

- 1) **x86** is currently the most popular architecture in personal computers
- 2) **IA-32** is the 32-bit version released in 1985
- 3) **x86-64** (or **x64** (or AMD64 (or EM64T))) is the 64-bit version released in 2000
 - But x64 is kinda complicated
 - So, we'll stick with IA-32
- 4) It's a **CISC** architecture
 - So, it has **lots and lots of instructions**
 - But those instructions are **flexible** and kinda human-oriented
 - ⇒ It's a little easier to read/write than MIPS in some ways

IA-32 Registers:

- 1) There are... **8**.

		Name	"Meaning"	Small parts
these are "general purpose" registers.	{	<u>eax</u>	accumulator	ax , ah/al
		<u>ebx</u>	base	bx , bh/bl
		<u>ecx</u>	count	cx , ch/cl
		<u>edx</u>	'double'	dx , dh/dl
these are used to manage the stack.	{	<u>esi</u>	source index	ax is the lower 16 bits of <u>eax</u> ; ah is the upper 8 bits of ax; al is the lower 8 bits of ax... etc.
		<u>edi</u>	dest index	
		<u>ebp</u>	base pointer	
		<u>esp</u>	stack pointer	

the program counter is eip (for "instruction pointer")

Very Common Instructions:

1) **mov** copies data around. it's really flexible.

mov eax, 10 eax = 10
mov ebx, eax ebx = eax
mov [esp+4], ebx MEM[esp+4] = ebx
mov ebx, [esp+8] ebx = MEM[esp+8]

instructions with only two operands (most of em) use the first as both a source and a destination.

add eax, 10 eax += 10
sub [esp+4], eax MEM[esp+4] -= eax

2) The first 4 **movs** would be **li**, **move**, **lw**, and **sw** in MIPS

3) Whenever you see [**brackets**], that means "access memory at the address inside the brackets."

4) You can access memory in **MANY** instructions, not just **mov**!

- The only restriction is that you can't access memory in **both** operands.

Accessing the Stack:

1) There are two ways.

1. using the push and pop instructions. easy!

```
push eax
push 10
pop  eax
pop  ebx
```

2. doing it just like in MIPS. wordier, but faster. **compilers** usually use this.

```
sub esp, 8
mov [esp+4], eax
mov [esp], 10
mov eax, [esp]
mov ebx, [esp+4]
add esp, 8
```

2) The push/pop instructions are more "human-oriented" which means they can be less efficient to run than the equivalent sequence of sub/mov/add.

3) push does a sub esp, 4 followed by mov [esp], reg.

4) pop does a mov reg, [esp] followed by add esp, 4.

- So, the code on the left does 2 subs and 2 adds, while the code on the right only does 1 sub and 1 add.

Calling and Returning:

1) The instructions are... call and ret.

```
call f    this pushes the return address and
           then jumps to f.
```

this is a CISC hallmark: **implicit operands.**

esp is changed here, even though we didn't mention it.

```
ret       this pops the return address and jumps to it.
```

```
ret 12    this pops the return address, then adds 12 to esp,
           and finally jumps to the return address.
```

This doesn't return 12!!!

The IA-32 *cdecl* Calling Convention:

Calling a Function:

let's translate this call. `f(1, 2, 3)`

all arguments are pushed in reverse order.

```
sub esp, 12
mov [esp+8], 3
mov [esp+4], 2
mov [esp], 1
call f
```

call both jumps to the new function and pushes the return address.

- 1) Usually no arguments are passed in registers because there are so few registers!
- With a common exception: `ecx` is often used for 'this' in OOP languages like C++.

Function Prologue:

- 1) Here's some weird stuff.

```
f:
do... this. push ebp
mov ebp, esp
make space for locals. sub esp, 0x10
```

we'd also push any *callee-saved* registers here: `ebx`, `edi`, and `esi`. but only if we needed them.

where are the arguments relative to `esp`? `ebp`?

- 2) `ebx/edi/esi` are like the *s registers* in MIPS – push and pop if you need to use them.
- 3) Call pushed the return address, and then we pushed `ebp`.
- 4) Then we made `ebp` point at the value that was just pushed, so the arguments start at `ebp+8`.
- 5) Since we subtracted 16 (0x10) from `esp`, the arguments also start at `esp+0x18 (+24)`.

Function Epilogue:

1) Similarly, strange.

place the return value in `eax`. `mov eax, [ebp-4]`
do... this. `mov esp, ebp`
`pop ebp`
pop and jump to the return address. `ret`

- In the MIPS example, the stack was *balanced* at this point, but *not here*...

Prologue and Epilogue Instructions:

1) x86 has the `enter` and `leave` instructions as shortcuts for... these.

`enter` is the same as `push ebp`
`mov ebp, esp`

`leave` is the same as `mov esp, ebp`
`pop ebp`

`gcc` sometimes uses these and sometimes doesn't...
and I have no idea why `~_(ツ)_/~`

Caller Cleans!

- 1) The last part is making sure the stack is *balanced*.
- 2) In `cdecl`, the caller *cleans* the *arguments off the stack*.
- 3) So, a *full, balanced* call might look like:

```
sub    esp, 12
mov    [esp+8], 3
mov    [esp+4], 2
mov    [esp], 1
call   f
add    esp, 12 ;sweep sweep
```

Peeking Under the Hood:

- 1) Let's compile `12_asm_test.c` with these flags: `-m32 -g`
- 2) Now we can run it in `gdb` and `disassemble main`:
 - `gdb 12_asm_test`
 - `disas main`
- 3) oh god. oh no. oH GOD this looks horrible
- 4) x86 has **two ways of writing assembly**
 - Cause, like, of course it does
- 5) The GNU tools default to **AT&T**

`movl $0x3, 0x8(%esp)`

- 6) The **Much Better One** is... **INTEL SYNTAX**

`mov [esp+0x8], 0x3`

- 7) `gdb` can switch between them with `set disassembly-flavor intel`

Peeking Under the Much-Easier-to-Read Hood:

Now let's see how `main` calls `f` with `disas /m main`

- 1) `/m` shows the C source if it's available
- 2) The **DWORD PTR** just says "this is a 32-bit load/store"
- 3) Apparently `x` is on the stack at `esp+0x1c`
- 4) What register does the return value come out in?

Let's look inside `f` to see how it computes its return value

- 1) When it accesses the arguments, it uses `[ebp+offs]`...
 - What the heck is `ebp`?
 - and what is that stuff at the beginning?
 - and why is the first argument at offset 8 instead of 0?

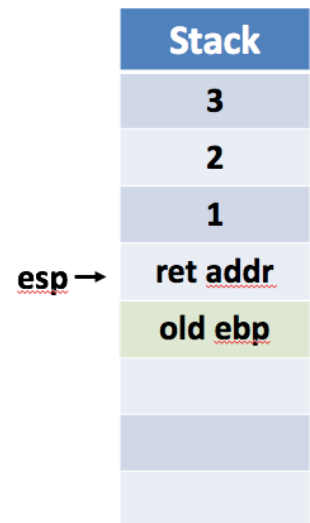
Return value comes out in `eax`.

The Last Piece:

The Base Pointer Register:

- 1) **esp** is the **stack pointer**: it **marks the bottom** of the **AR**
- 2) **ebp**: "**base pointer**" - **marks the top*** of the **AR**
- 3) When we first came into **f**, the **call** instruction just pushed the return address, so the stack looks like:
- 4) Then we have this weird sequence:

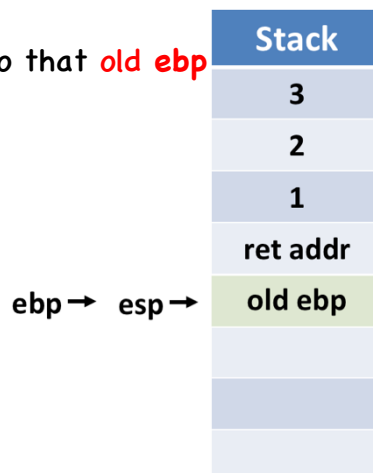
```
push ebp
mov  ebp, esp
```



- 5) **push** saves the **old value** of **ebp**:



- 6) **mov** makes **ebp** point to that **old ebp**

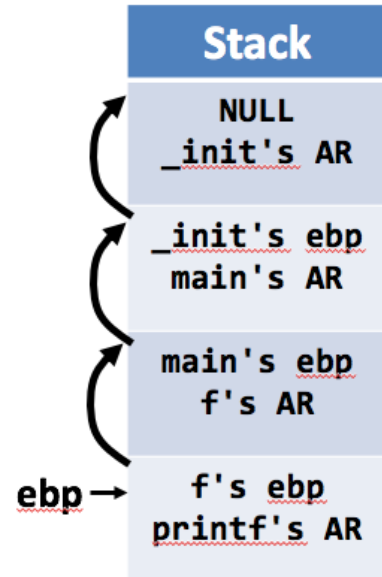


- 7) and now.... uh..... um..... what did that do?

It's a **Linked List** of **ARs**!

- 1) **ebp** is the pointer to the **head of a linked list**
- 2) Every **AR** stores a **pointer to the top** of *the AR of the function that called it*
- 3) When the function is about to return, it does:
mov esp, ebp
pop ebp
- 4) This **"unlinks"** the AR from the list
- 5) This is used for **unwinding**
 - or getting a **stack trace**
 - this is how the **"where"** or **"bt"** commands in **gdb** work: **by following the base pointers!**

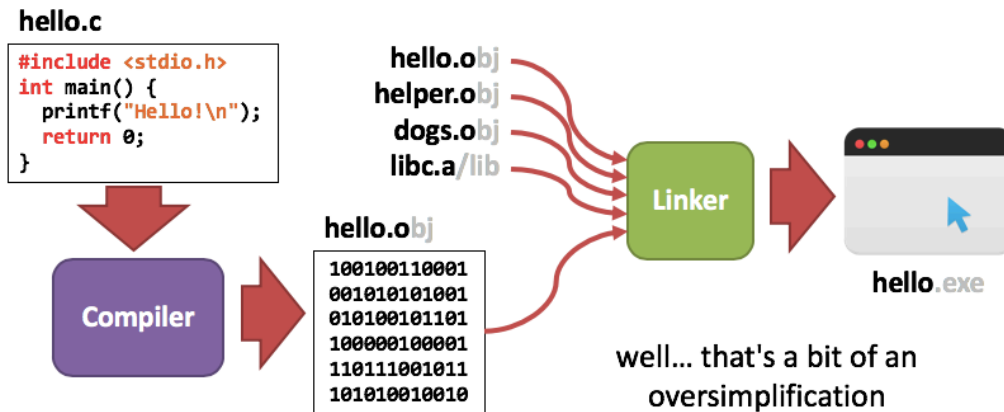
MIPS has an analogous register, **fp**



Programs - Preprocessing, Compilation and Linking:

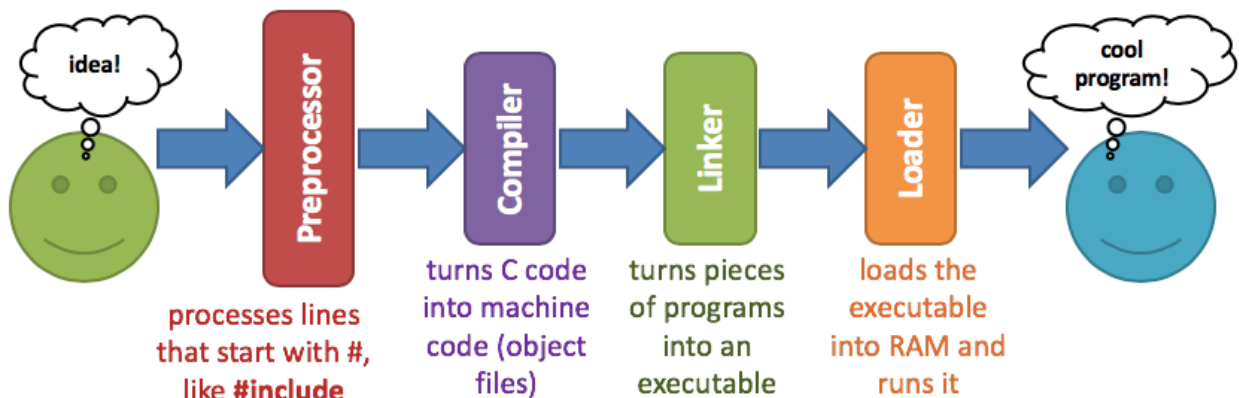
The Compilation Toolchain:

The General Process:



Nothing to Lose but Your Chains:

1) The **compilation toolchain** is the sequence of programs that you use to go from programming-language code to an executable file



(the preprocessor is pretty much unique to C/C++.)

The Preprocessor, Redux:

Header Files (*.h):

- 1) Remember:
- 2) The header file is the **public interface**
 - NO CODE!!!
- 3) The C file contains the **implementation**
 - ALL THE CODE!!!
 - and any private structs, enums etc.



The #define Directive:

- 1) #define is... weird
- 2) You can make constants-ish

```
#define NUM_ITEMS 1000
```

- 3) Whenever you write NUM_ITEMS...
 - The preprocessor will *textually* replace it with 1000
- 4) This is **not** a variable!
- 5) You can make it replace it textually with *anything*

```
#define NUM_ITEMS oh no this is a bunch of crap
```

- 6) You can replace any word-like thing with anything

```
#define int float  
#define true false
```

- 7) Never do this :P

Conditional Compilation:

- 1) We can choose which code to *compile*, based on some condition
- 2) One very common condition is "has this preprocessor name been `#defined`?"

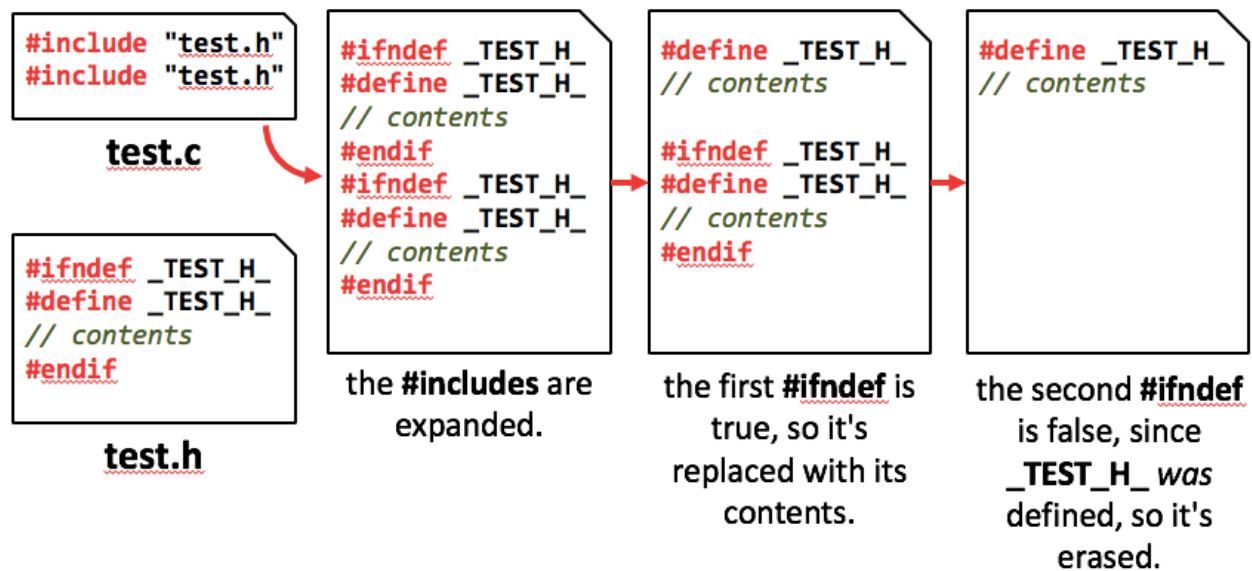
```
#ifdef SOME_OPTION
// code path 1
#else
// code path 2
#endif
```

3) The "not-taken" side of the if-else will literally not even be in the resulting program

- There is also **#ifndef** for "if not defined"
- 4) This is super common in platform-specific code
- or "optional features"

How Include Guards Work:

- 1) These weird things in header files prevent double-inclusion.



#define With Parameters:

```
#define TEST_BIT(v, n) ((v) & (1 << (n)))
```

1) This is a **preprocessor macro**

- It looks like a function
- You can write it like you're calling a function
- But it's **all text replacement**

```
#define streq(a,b) (strcmp((a), (b)) == 0)
```

2) It's easy to get yourself into trouble with macros if you use a parameter more than once...

- Lots of parentheses (b/c reasons)

The Preprocessor is Weird:

1) C/C++ are really the only modern languages which have it

- Cause it's, uh, *suboptimal*

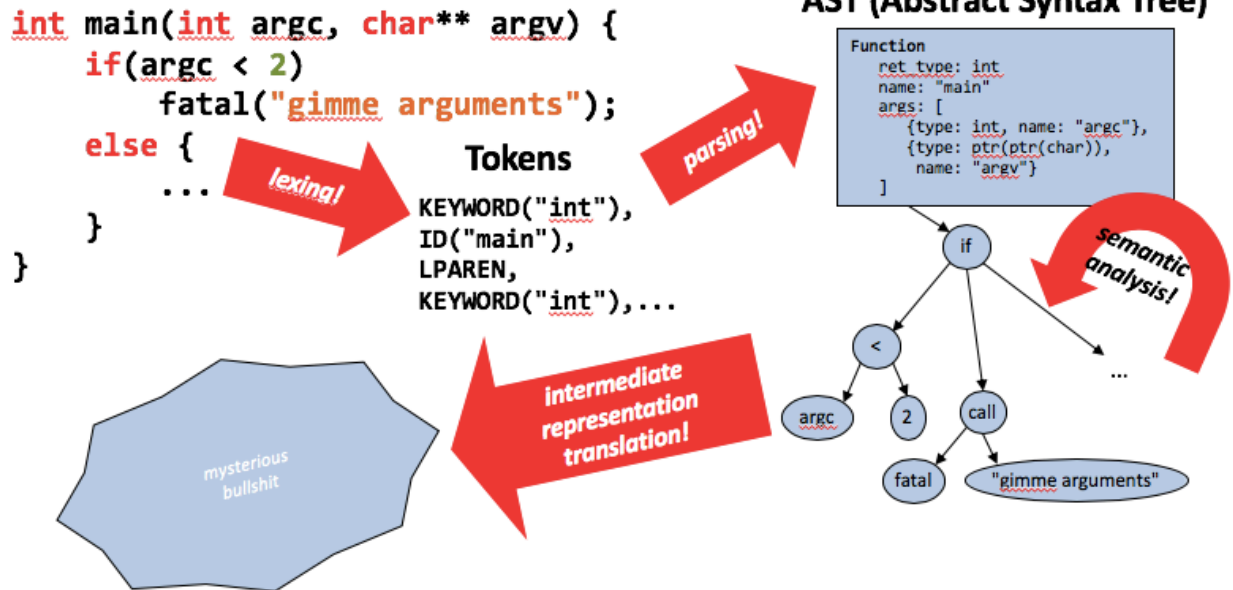
2) These past few slides cover like 95% of what you will ever need

3) I hope you never have to deal with it too much.

The Compiler:

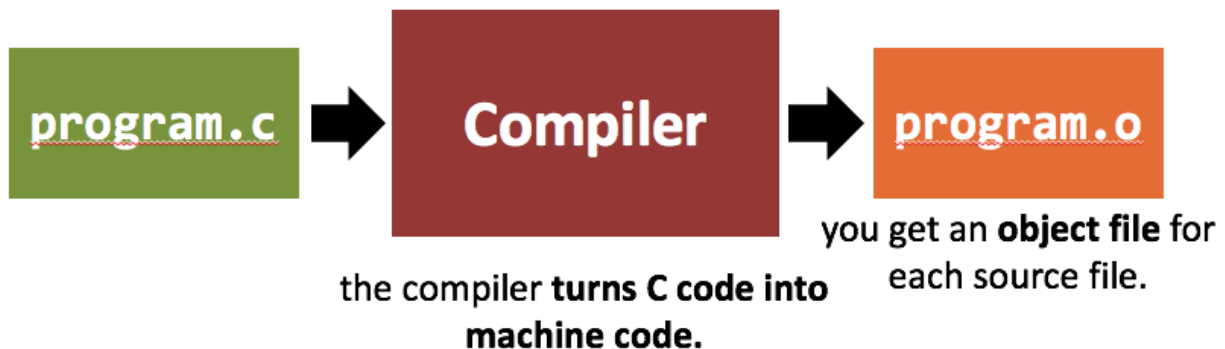
How Does a Compiler Work?

1) It's, uh, complicated.



Take CS1622:

1) All that really matters:



Object Files:

Blob Object Files:

- 1) For every C code file the compiler takes, it produces **one** object file
 - Doesn't matter how many headers it includes
- 2) By default, it hides these files from you, but...
- 3) **gcc -c** will output the object files
- 4) The **file** command in Linux is very useful
- 5) So, what's IN an object file?
 - Let's try **objdump -x**
 - ⇒ WOAHH WHAT

Anatomy of an Object File:

An object file has several **sections** (or **segments**)

- 1) The **.text** segment contains...
 - **The machine code**
- 2) The **.data** segment contains...
 - **Global variables**



Kinds of Data:

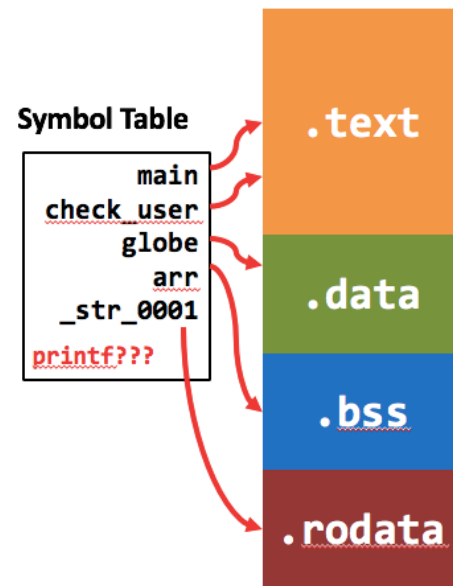
Three kinds of data segments actually

- 1) **.data** is for **globals**
 - **int globe = 100;**
- 2) **.bss** is for **globals initialized to 0**
 - **int arr[100];**
 - There's **no need to store 0s**
 - So, it's a bit of optimization
- 3) **.rodata** is for **read-only data**
 - **"hello there"**
 - If you try to change the values here...
 - ⇒ You get a **segfault!**



A Map! But That Looks Like... a Closeup of an Object File:

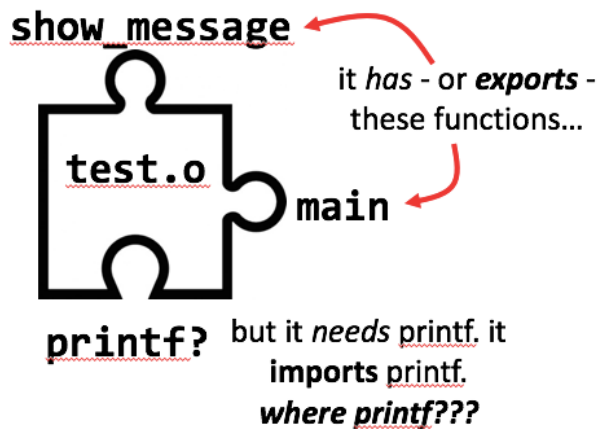
- 1) Then there's the **symbol table**
 - "symbol" means "name"
- 2) This is a list of all the *things* in the file
 - Their **name**
 - **What** they are (function, var, etc.)
 - Which **segment** they're in
 - Their **address**
 - and some other stuff
- 3) But it also lists some things **NOT** in the file



Linking:

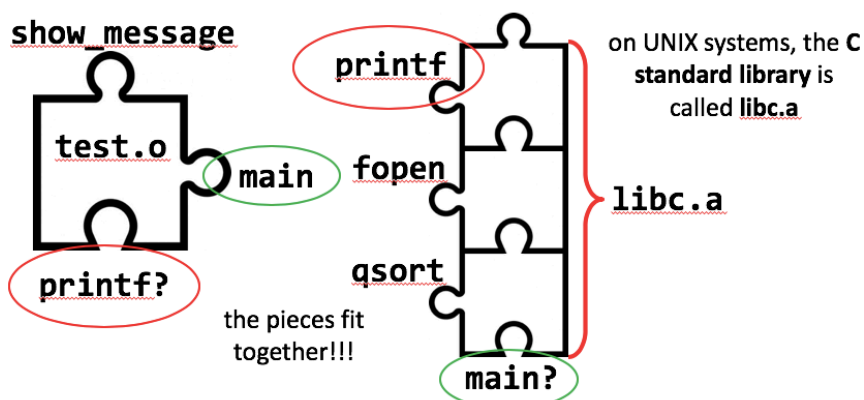
Puzzle Pieces from the Clay:

- 1) Object files are an **incomplete part** of a **whole**, like a puzzle piece



The Adventure of Link:

- 1) A **library** (or **archive**) is just a collection of object files.



A Link to the Past:

- 1) The **linker** takes all these pieces and **links them together**.
 - The result is... an **executable**!
- 2) Let's use `objdump -x` on an executable
 - There's **not really much difference**
- 3) The only real difference between an object file and an executable is **"does it have everything it needs to be run?"**



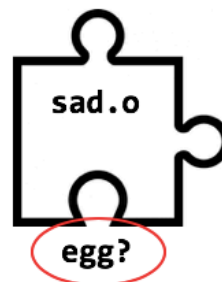
Linker Errors:

- 1) When the linker is putting your puzzle together, things can go wrong.



multiple definition errors happen when the **same name** is defined in **two or more** places.

if an object *imports* a symbol, but nothing else *exports* it, we can't finish linking cause... well, it's not there.



Static:

- 1) On functions, **static** is very similar to **private** in Java
 - **Static** means "do not export this to other compilation units"
- 2) Let's put **static** before **print_message** in **sub_island.c**, compile, and see what happens when we try to link
- 3) Let's see what **nm sub_island.o** prints
 - **nm** lists *names*.
 - Lowercase **t** means it's a **local symbol**- It's contained within **sub_island.o** and *no one else can see it*



extern:

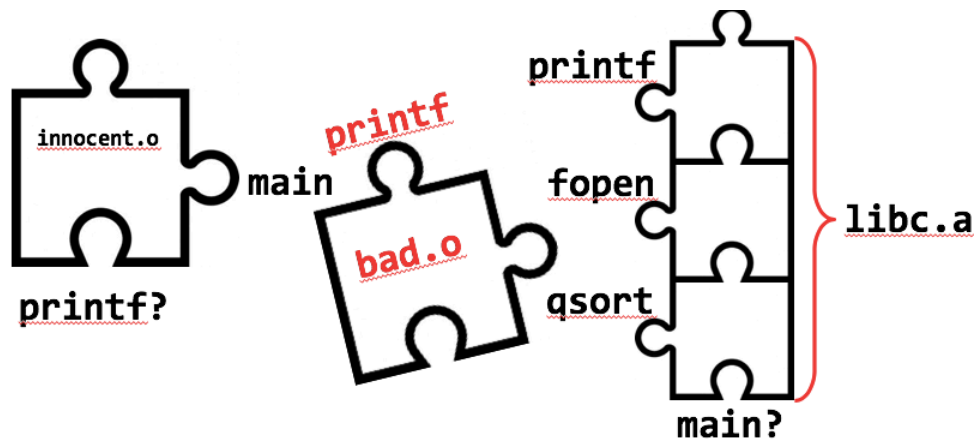
- 1) *leaving static off a function* makes a "bump"
- 2) **extern** makes a "hole"
 - It has *no effect on functions at all*
- 3) The only time you need to use **extern** is on **global variables that are shared across files**.
 - Global variables are bad enough, but shared globals?
 - NEVER.
 - EVER.
 - DO.
 - THIS.
 - OKAY?

Programs – Dynamic Linking and Loading:

More about Linking:

Will the Real printf Please Stand Up?

1) Linking is (weirdly enough) done by *name*



2) Here we can have the "wrong" `printf` linked into our program...

Every File's an Island:

1) Source files in C don't actually know anything about each other...

– So, it's the *linker's* job to match up the names between files.

2) Here we have two C files which we can compile with `-c`

– `14_innocent.c`

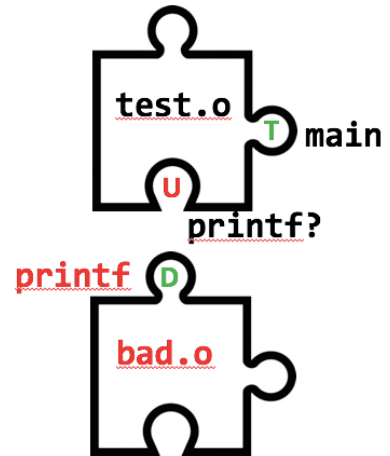
– `14_bad.c`

– It whines a bit but does work

3) You can use the `nm` program (name) to see the symbol tables

The Output of nm:

- 1) Each symbol has a type
- 2) There are many but:
 - **T** is a **bump**: an exported **T**ext symbol
 - **U** is a **hole**: an **U**ndefined symbol
 - ⇒ It needs to be imported
 - **D** is a **bump**: an exported **D**ata symbol
- 3) You might see where this is going
- 4) ...yes, the linker really is that dumb
- 5) Lowercase **t** and **d** are **local (static)** symbols contained within the object file but **neither imported nor exported**



Just Mash it in There:

- 1) name == name?? OKAY!!
- 2) you can link with gcc too
 - **gcc -o bad bad.o test.o**
- 3) aaaand now we have a program that crashes
- 4) Cause it's trying to execute an integer in the data segment



Function Pointers:

Point to Anything:

- 1) We can have pointers to data anywhere in memory
- 2) **Why not to functions, too?**
- 3) A **function pointer** is..... gee, I dunno. what do you think
- 4) in C, it looks... terrifying

```
int (*fp)(int); // WHAT
```

return type **parameter**
 types

this is actually a variable declaration.
it makes a variable **fp** that *points to a function* that takes an int and returns an int.

Typedef is Your Friend:

- 1) What is this?

```
float(**(*fp)(const char*))(float, float);
```

- 2) uhhhhhhh

- THIS IS A REAL TYPE.

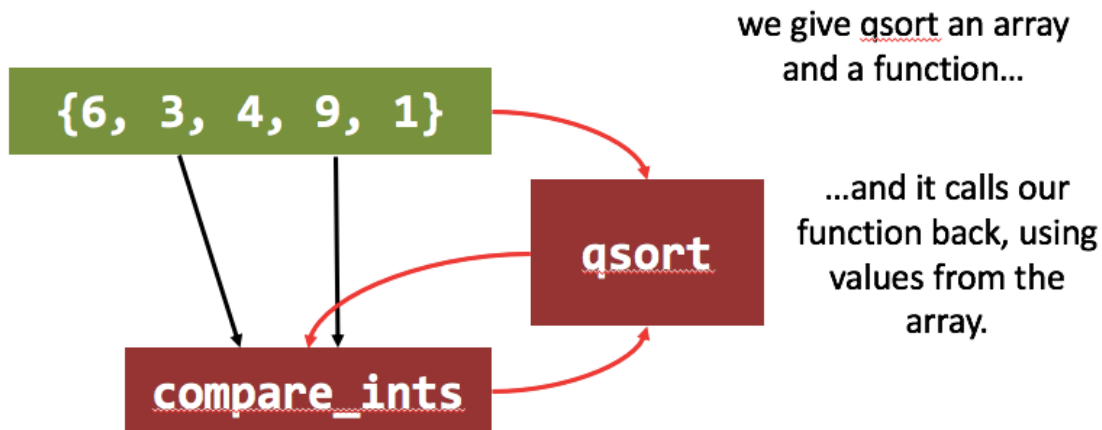
- Pointer to a function which takes a **const char***, and returns an *array of function pointers*, each of which takes two floats and returns a float

- 3) **typedef** is nice to use when making function pointer types

```
typedef float(*OPERATOR)(float, float);  
typedef OPERATOR* (*OPERATOR_GETTER)(const char*);  
OPERATOR_GETTER get_operators;
```

Why Function Pointers?

- 1) You can **pass functions as arguments** to other functions
 - This is a **very powerful technique**
 - You can **parameterize actions** just like you can with **values**
- 2) Let's look at `14_qsort.c` and `14_qsort_structs.c`
 - This is what your lab 5 is about!



These are Not New to You, Actually:

- 1) Java implements function pointers **indirectly** by having you implement **interfaces**
 - Like **Comparable**!
- 2) But the interface solution is a bit limiting...
 - What if you want to be able to sort an array of objects in five different ways? you only have one interface method.
- 3) So more modern versions of Java give you more freedom.

Dynamic Linking:

There's a Hole in my Heart:

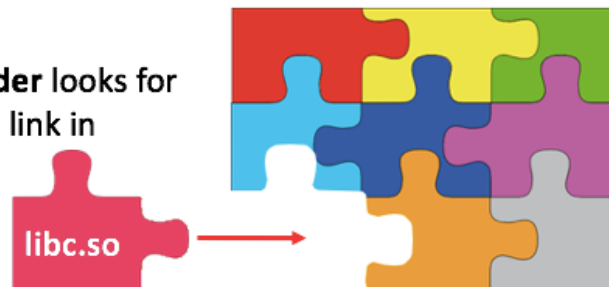
- 1) What if we left the holes in the executable?
 - Like leaving out a piece of a puzzle
- 2) This is a technique called **dynamic linking**
- 3) Basically, we **leave the last step of linking unfinished**
- 4) When we **run the program**, **then** we find that last piece(s)



Dynamic Linking:

- 1) Many programs need printf. **why duplicate it?**
- 2) So, we put e.g. the C standard library (libc) into a special object file
 - A **shared object (*.so) file**
- 3) The **loader** is responsible for doing this **final linking step**
 - In fact, on Linux, the linker and the loader are **the same program**

when the program is run, the **loader** looks for the right shared object(s) to link in



Static Linking:

- 1) If you make a **whole puzzle with no holes** – that's **static linking**
- 2) There are **two main downsides** of **static linking**:
 - **Bigger executables**
 - It can **embed bugs** into your **executables**
 - ⇒ libc ain't perfect
 - ⇒ If it has a serious bug, the only way to fix your program is...
 - ⇒ **Recompile** and **redistribute**.
- 3) Statically linked executables can be loaded more quickly and have no dependencies, so they're more self-contained and easier to distribute



Pros and Cons:

1) With dynamic linking...

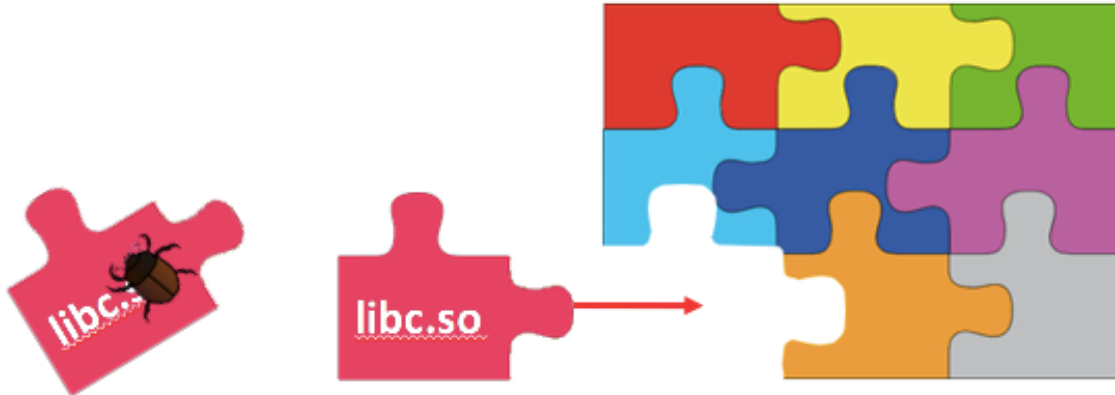
- We can just **fix** **libc.so** and now any program that uses it is automatically "fixed"

2) But...

- Fixing bugs can **break** programs.

- Shared libraries can **have multiple versions**

- If the shared library can't be found, the program **won't run**



Suppose a program erroneously depends on a buggy library function

1) You fix that function in the shared library

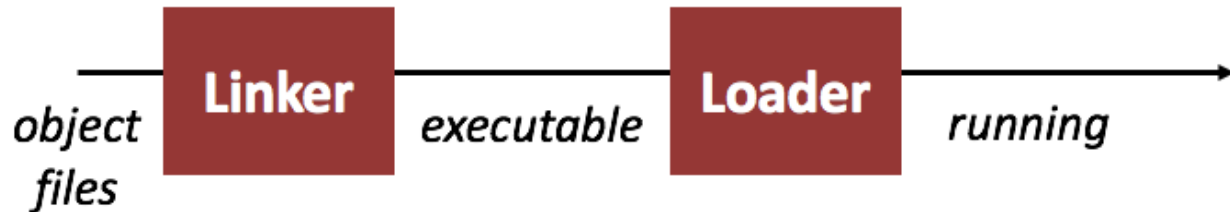
2) And now the program crashes cause now that function correctly returns NULL instead of an invalid-but-it-never-crashed pointer

Dynamic Loading:

Time, time, time:

There are three times when we can put a library into an executable

- 1) **Link-time** (**static linking**)
- 2) **Load-time** (**dynamic linking**)
- 3) **Run-time** (**dynamic loading**)



Dynamically Loaded Libraries:

- 1) A dynamically **loaded** library is just like a shared library...
 - But the **application** decides **which** shared objects to load
 - And **when** to load them!
 - **While it's running!!**
- 2) This is often used to load **optional** functionality
 - Usually called **plugins**

MPEG
Decoder



Program



H.264
Decoder



Asking the OS:

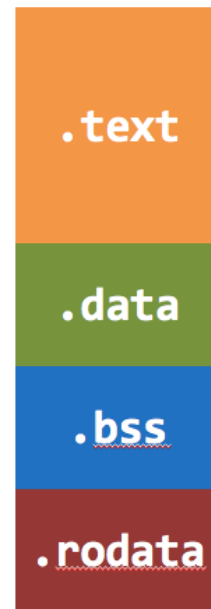
- 1) To dynamically load a library, we have to **ask the operating system**
 - It will invoke the loader for us
 - Once it's loaded, we can **get function pointers** to the functions inside.
- 2) What interface (or "API") a plugin uses are defined either by the host program or by some standard

Programs – Loading and Running an Executable:

Executables and Loading:

What the Heck is in an Executable File?

- 1) Pretty much the same stuff as an object file
 - Machine code ("text")
 - Data
 - Symbol table(s)
 - OS-specific info
- 2) `objdump` can inspect executables as well!
 - This is because they use the **same file format**
 - ⇒ Well, on most flavors of UNIX anyway...



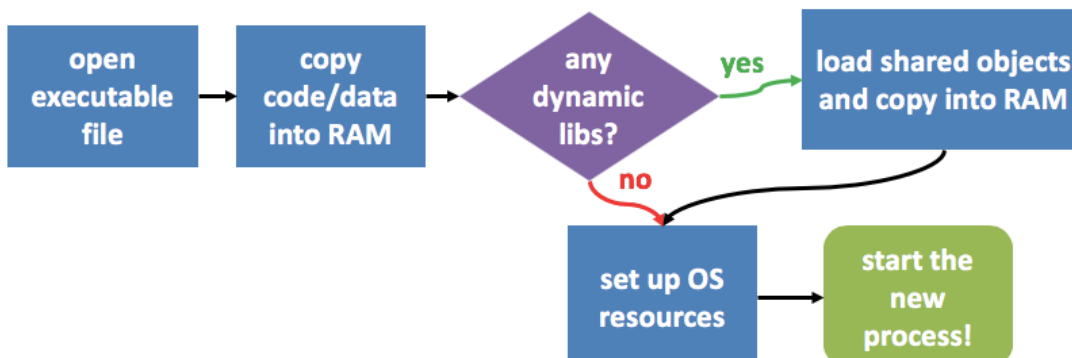
Executable Formats:

- 1) Different OSes use different formats for their executable... things
 - 2) Each of these formats is **broadly similar**, but each operating system works differently and so can require different information
- Don't remember this table, it's just here for demonstration

OS	Format	Executables	Static Libraries	Objects	Shared Objects
<u>UNIXes</u>	ELF	<nothing>	*.a	*.o	*.so
<u>macOS</u>	Mach-O	<nothing>	*.a	*.o	*.dylib
<u>Windows</u>	PE/COFF	*.exe	*.lib	*.obj	*.dll

The Final Countdown:

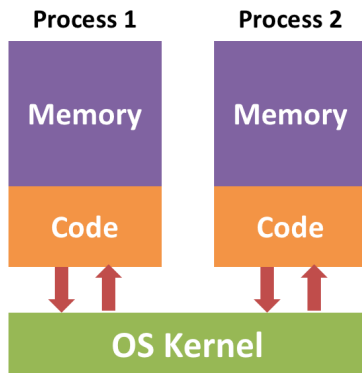
- 1) The last step is **loading**
- 2) The **loader** is the part of the OS that, uh, **loads the executable**



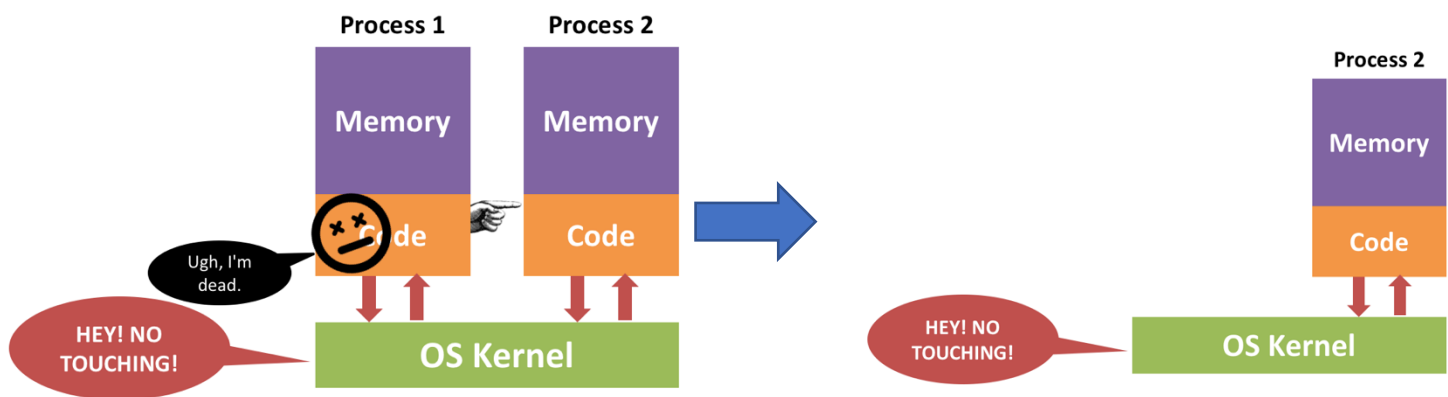
Processes:

All My Children:

- 1) A **process** is the in-memory representation of a program, all its data, its resources (like open files), etc.
- 2) The OS's main responsibility is **controlling access to resources**
- 3) The OS wants everyone to **play nicely**



- 4) and if they don't...
 - It's lights out for the process



- 5) **ps** and **pstree** let you see what processes are running
- 6) OS controls resources
 - Abstracts
 - Protects
 - Allocates

Stuck in Your Teeth:

- 1) The **kernel** is a process too, but...
- 2) It's *special*
- 3) Basically, this *is* the program that makes the OS work
- 4) It makes **every other process** believe that they are the **only program running on the whole computer. (it lies)**
- 5) It's also allowed to do **anything it wants**.
 - ...which means a malicious program gaining access to the kernel is *a very bad situation*
 - *Malware: A program taking control of the kernel*

OS Kernel

Address Spaces:

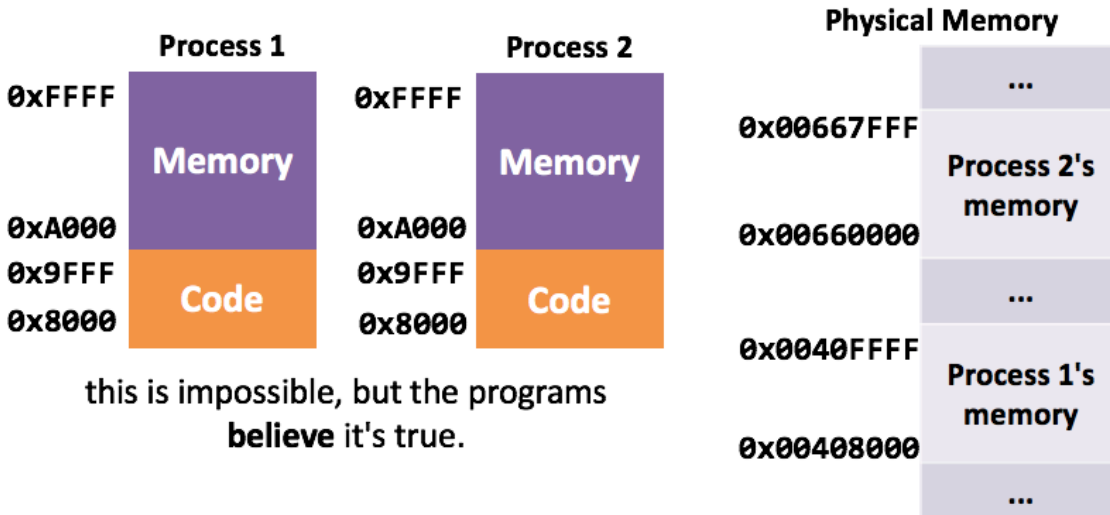
- 1) Each process gets its own **address space**
 - Its own little memory compartment
 - Processes can't access each other's address spaces or the OS's
- 2) Old CPU architectures did this with *memory segmentation*
 - This is where **"segfault"** comes from
 - They literally sliced up the memory and gave each process a piece
- 3) But these days we do something a little more complicated (and flexible)
- 4) Kernel puts each process in its own address space



Virtual Memory:

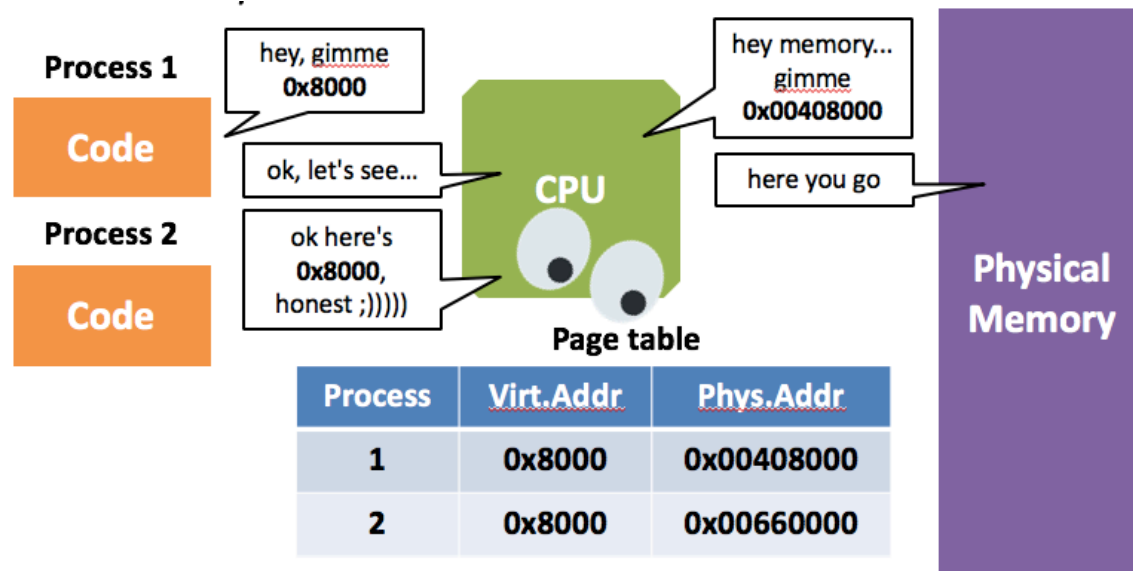
Virtual Memory:

- 1) **Virtual memory** abstracts memory addresses.
- The addresses each process accesses are *not the real addresses sent to the memory*.
- Virtual memory is the basis for so many things



How?

- 1) Virtual memory is a feature of the **CPU hardware**



🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍

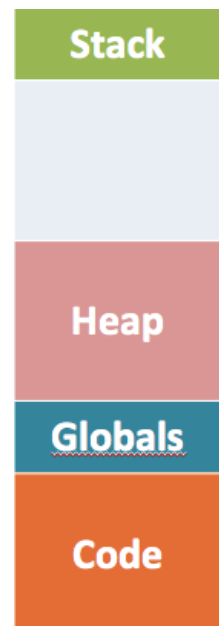
Shattered Memories:

- 1) Physical memory is a **scarce resource**
- 2) the OS splits memory into **pages**: small chunks that can be given to processes
 - Usually these days, a page is 4KiB
- 3) The OS keeps track of which parts of memory belong to which process
 - Since we have *equally-sized chunks...*
 - **How** do you think the OS keeps track of which pages are **used/free**? ;)
 - ⇒ a **bitmap**!

Page	Physical Address	Owning Process
...	...	
10	0000A000	<OS>
9	00009000	<OS>
8	00008000	bash
7	00007000	bash
6	00006000	bash
5	00005000	ls
4	00004000	ps
3	00003000	id3edit
2	00002000	<OS>
1	00001000	<OS>
0	00000000	<OS>

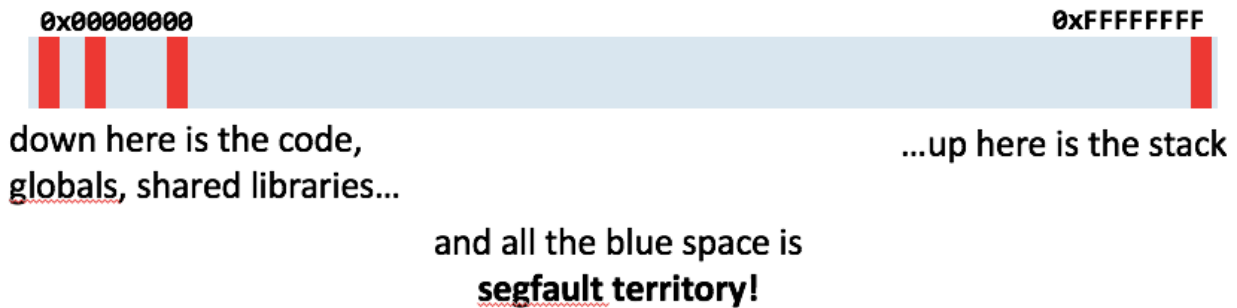
One Process's Address Space:

- 1) Here's an address space that a process sees
- 2) The code (or the **text** segment) usually resides at the **bottom** of memory (the lowest addresses).
 - (not usually at address 0, though)
 - ⇒ (0 is **NULL** and we want that to be invalid)
- 3) Then comes the global (static) data segment
 - The code and globals are basically copied directly from the executable file
- 4) But the stack and heap weren't in the executable
- 5) Where'd they come from?
 - The **loader** also set those up for us!



Looking at a Process's Memory Map:

- 1) The *memory map* is the arrangement of the address space.
- 2) You can get a *process's pid* with `ps...`
- 3) ...and then use this to print its memory map: `pmap -x pid`



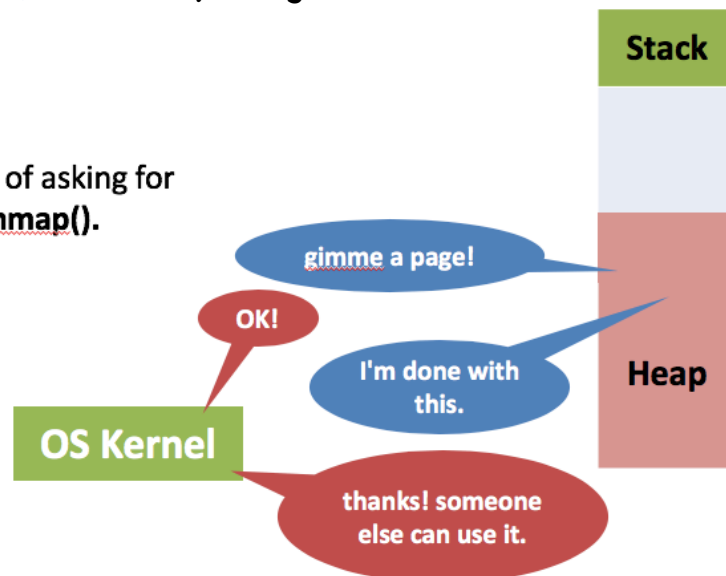
memory maps are usually pretty sparse. *rarely* will a program have most/all of the memory used.

Segfault -> accessing a piece of virtual memory that does not map to a real piece of memory

Honey, I Broke the Heap:

- 1) Processes can *ask the OS for new pages*
- 2) When they're done with them, they can *give pages back*
- 3) This is what `[s]brk()` is actually doing!

a more modern way of asking for pages is with `mmap()`.



OS - System Calls:

The OS:

The Responsibilities of the OS:

1) Resource Control.

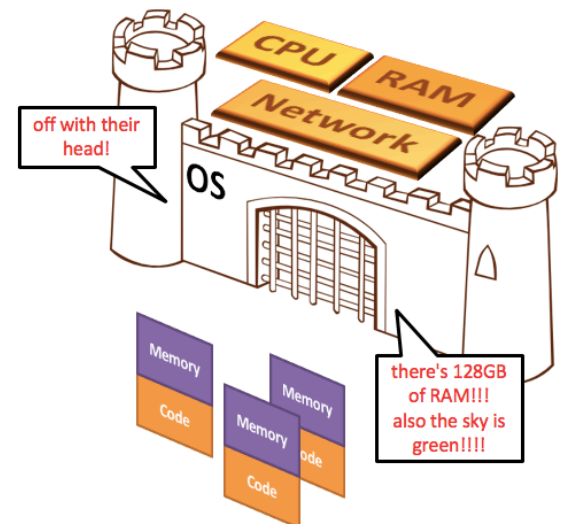
- It's the gatekeeper to all the computer's resources

2) Process Control.

- It makes everyone play nice
- It can kill broken/evil processes

3) Hardware Abstraction.

- Since processes must ask OS to access resources on their behalf...
- The OS can make processes believe things that aren't real



this isn't really painting a flattering picture of the OS, is it

Walled Cities:

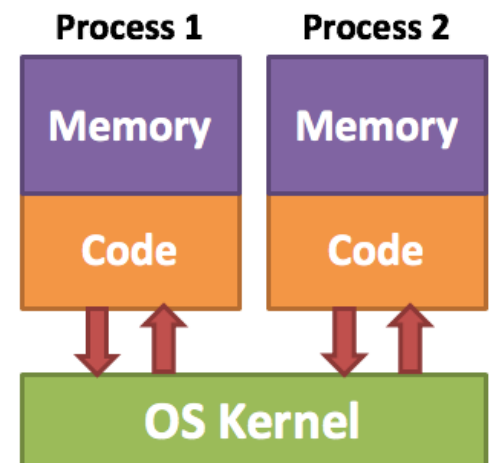
1) The OS kernel is a program – a *process* – that manages all the rest

2) It can **access any resource** and **control any other process**

- How do we keep that power from falling into the wrong hands?
- And how do we do it on a computer with only a single CPU??

⇒ *How do we even run multiple processes on a single CPU*

Abstraction is fancy lying.



Building the Castle:

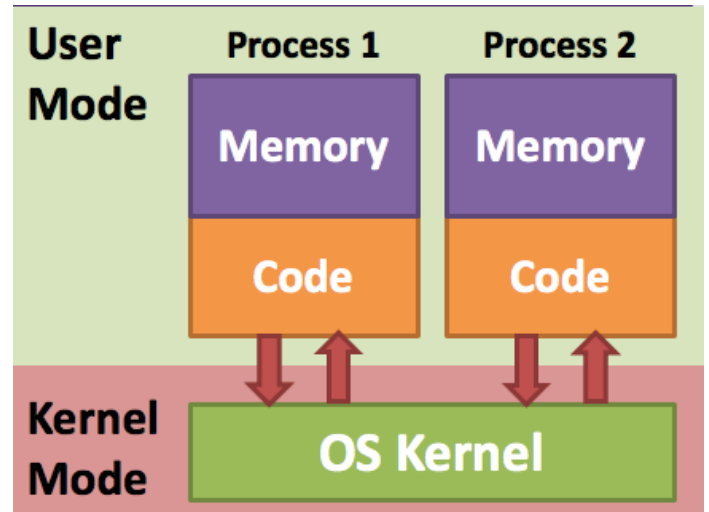
1) CPUs can run in (at least) **two modes**:

- **User Mode**, for running "untrusted" processes
- **Kernel Mode**, for running the kernel (duh)

2) **User Mode** is very restricted:

- Many instructions are illegal
- Most of the memory is off-limits
- Hardware can't be accessed

3) So... how do we **switch modes**?



System Calls:

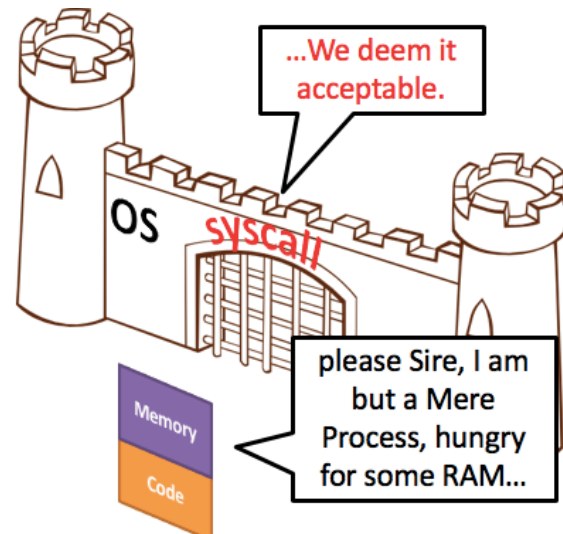
System Calls:

1) A **system call** is a special kind of function call that user-mode processes use to **ask the OS to do something for them**.

- If the OS is the castle wall...
- Then **syscalls** are a **gate**

2) System calls are a **CPU feature**

- Since it must switch modes



System Call CPU Mechanisms:

1) You learned how to do them in MIPS

- It's.... **syscall**

2) In IA-32, it's **int** or **sysenter**

- **int** doesn't mean integer!
- It means **interrupt** (we'll come back to those)
- Older versions of Linux use **int 0x80**
- Newer versions use **sysenter**

2) On both, you put the **syscall number** in a certain register

- This identifies **which operation** the process wants to perform

3) The arguments get passed as usual

4) Then you perform the system call instruction

5) and then...

MIPS	IA-32
syscall	sysenter int n

Opening the Portal:

1) Switching modes causes a few things to happen

User Space

```
push 45  
mov eax, 17  
sysenter
```

Stack
3
2
1
45

the CPU switches **memory spaces**:

the kernel has its own memory

execution jumps to the **system call handler** in the kernel

the kernel also has its own **stack**

and of course, the CPU is now in **kernel mode**

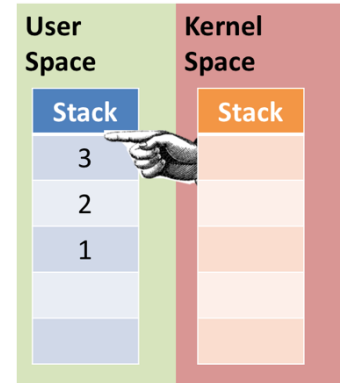
Kernel Space

```
mov esp, kstack  
push ds
```

Stack
0
78
0

The Upside-Down:

- 1) Kernel code is pretty similar to user code
 - There are some restrictions, like no libc
 - But you can do **special things**
 - Like **accessing other processes' memory spaces**
- 2) This is how it sees the parameters to the system call - they're on the *user* stack



Switching Tracks:

- 1) When the kernel is done handling a syscall...
 - It could **return to the user process**
 - Or it could **switch to another user process!**
- 2) This is how **one CPU can run many processes**
- 3) At any given moment, it's only running *one*
 - But over time, they *all* get some attention
- 4) The OS is like a juggler



Context Switches:

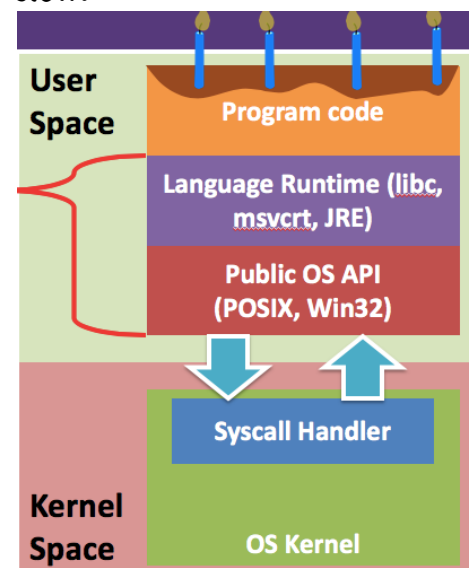
Barreling Down the Track:

- 1) A **context switch** is when the CPU changes modes
- 2) But the CPU is like a train:
 - High speed in a **straight line**
 - But slow to **change directions**.
- 3) So, we have this annoying problem:
 - We need the OS to do *anything*
 - But system calls require a **context switch**, which is **slow**.



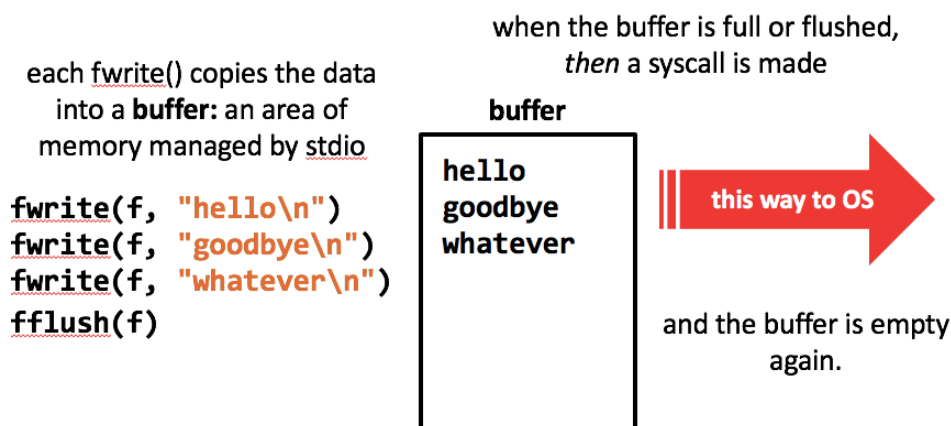
Layer Cake:

- 1) User programs usually look like this:
- 2) The layers in the middle do a good bit of work trying to make **as few system calls as possible**.
- 3) As an example, let's look at a common one: **buffered I/O**



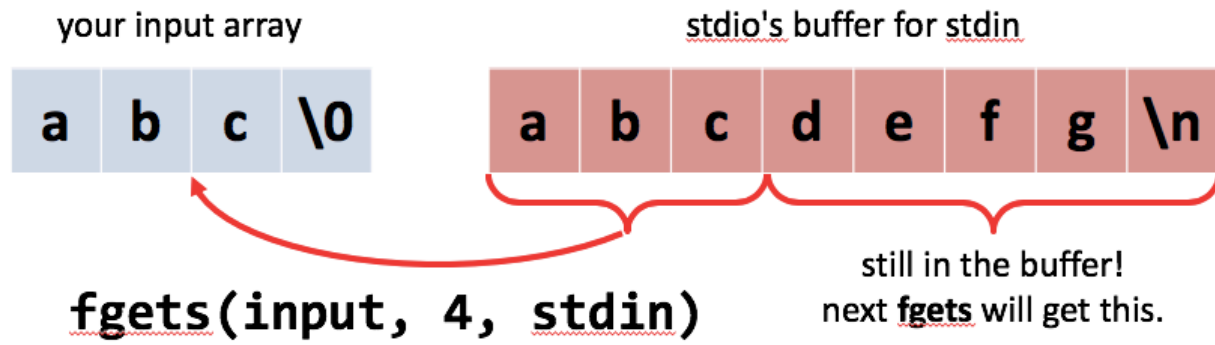
Don't Do It All At Once:

- 1) There is a **fixed overhead** for context switches
- 2) So, if we can do *more work per switch*, we can reduce their impact
- 3) Let's say we're writing to a file.



An Example with `fgets()`:

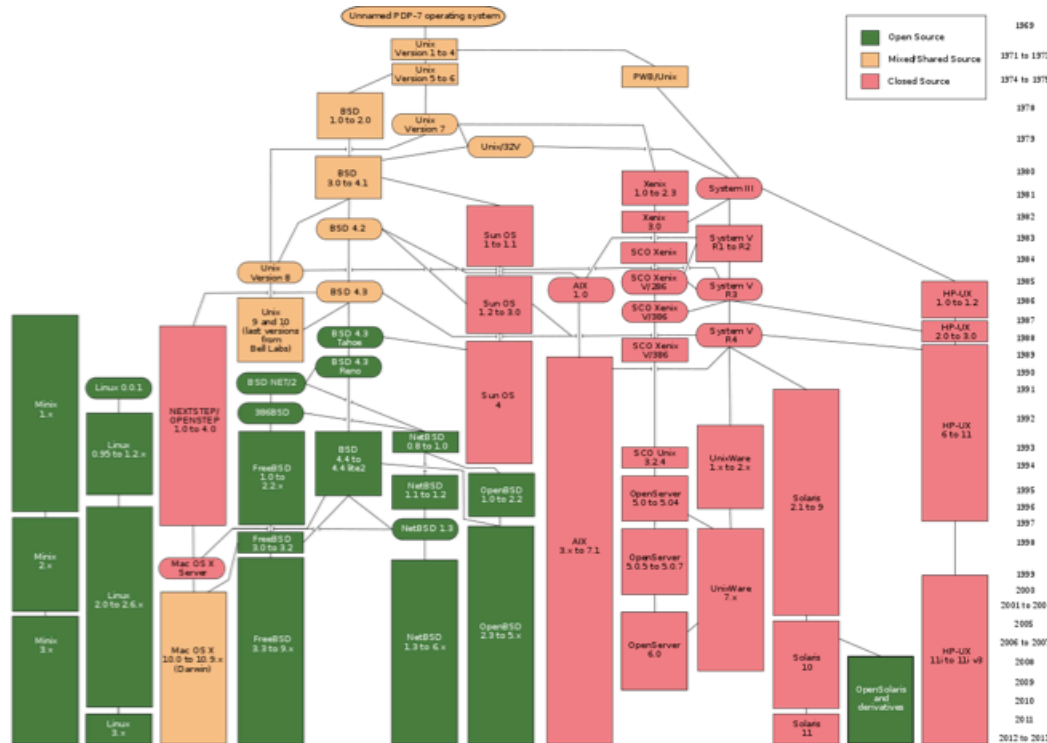
- 1) If the user types in **more text** than there's room for in your array...
 - 2) The rest of the line is still in **the input buffer**, and the next call to `fgets` will get that buffered data
- It will **not** wait for you to type more!



The POSIX API:

The POSIX API:

- 1) POSIX stands for **Portable Operating System Interface**.
- 2) (the X just looks cool.)
- 3) POSIX was defined as a **universal API** for all the flavors of UNIX, which Linux (mostly) implements as well.
- 4) This is why it was needed.

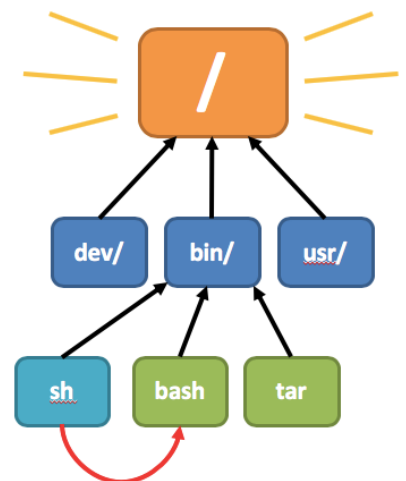


The POSIX API:

- 1) Linux and macOS (mostly) implement POSIX
 - But historically not Windows, cause Windows
 - ⇒ But I guess that's changing-ish.
- 2) It's based on this idea: **what if everything were a file**
 - Hard drives? **Files**.
 - Displays? **Files**.
 - Keyboards? **Files**.
 - Processes? **Files**.
 - Files?.....
 - ⇒ **Files**.

The Big Unifying Concept:

- 1) Everything on the system is represented in this big tree
- 2) The root of this tree is is /, which is called... the **root** directory
- 3) The internal nodes are **directories**, and the leaves are **files**
- 4) Directory names **always** end in a /
- 5) We can also have **symbolic links**
 - They're basically pointers.
 - ⇒ When we access /bin/sh on tho, it's really accessing /bin/bash.



Accessing the "File" System:

1) There are four main system calls, and they look familiar-ish...

```
int open(const char* filename, int flags);  
int read(int fd, void* buffer, size_t size);  
int write(int fd, const void* buffer, size_t size);  
int close(int fd);
```

2) These are the **real deal** – when you call these functions, *you are talking directly to the OS!*

3) There is no **FILE*** here. that is a `<stdio.h>` abstraction.

4) **open()** returns a *file descriptor* instead.

– This is just an integer which uniquely identifies the open file.

The Standard Input/Output/Error Files:

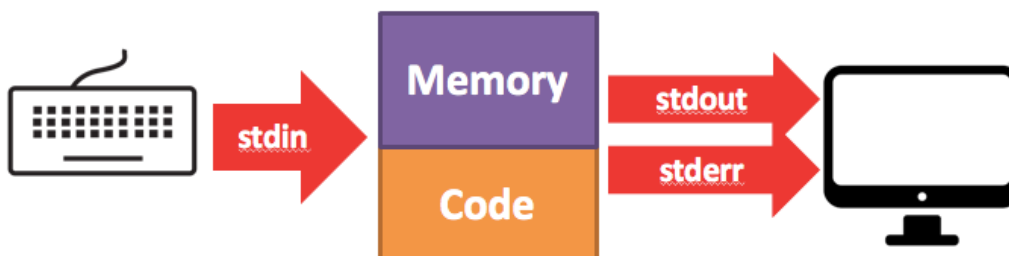
1) Every process on a POSIX system is given these three files when it starts.

2) Java names these **System.in**, **System.out**, and **System.err**.

3) Notice that their file descriptors are 0, 1, 2.

– Starting at 0 and going in order...

Name	Descriptor
<u>stdin</u>	0
<u>stdout</u>	1
<u>stderr</u>	2



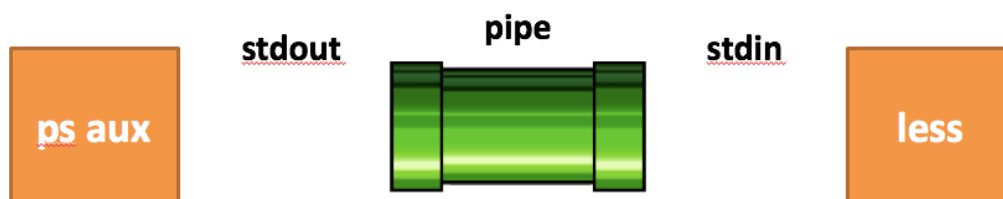
this is the default arrangement, but it can be changed easily.

World 7:

1) A **pipe** connects the output of one process to the input of another.

2) Let's try **ps aux | less**

3) Here's what's happening:



Watching it Go:

1) We can see *all* the system calls a program makes as it runs using the **strace** command.

2) Let's **strace** a simple "hello world" program.

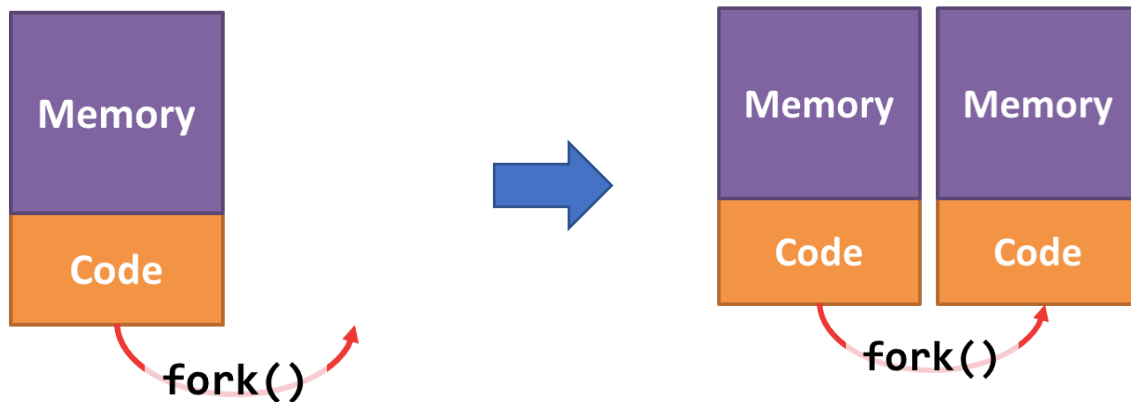
- **execve** is the call that actually runs the program. (sorta.)
- **brk** is used to see where the heap begins. (hey, that's familiar!)
- **mmap** (memory map) is a more modern way to dynamically allocate memory, and can also be used to read/write files.
 - ⇒ It's part of the **virtual memory** system.
 - ⇒ **mprotect** lets you change the access rights on memory areas.
 - ⇒ **munmap** gives the mapped memory pages back to the OS.
- **fstat** gets information about an open file.

OS: Process Creation and Deletion:

POSIX Process Creation:

Mitosis:

- 1) In the beginning
 - When a UNIX/Linux system first starts, the only process running is **init**
- 2) The way **every** process starts in POSIX is by **splitting off from another process**
- 3) The POSIX function is **fork()**
- 4) The original process is the **parent**, and the newly-forked process is the **child**
- 5) Just like when cells split:
 - **Both processes are completely identical**
 - Same code, same data, same everything
 - ⇒ Even *where they're executing* is the same...



The Process Tree:

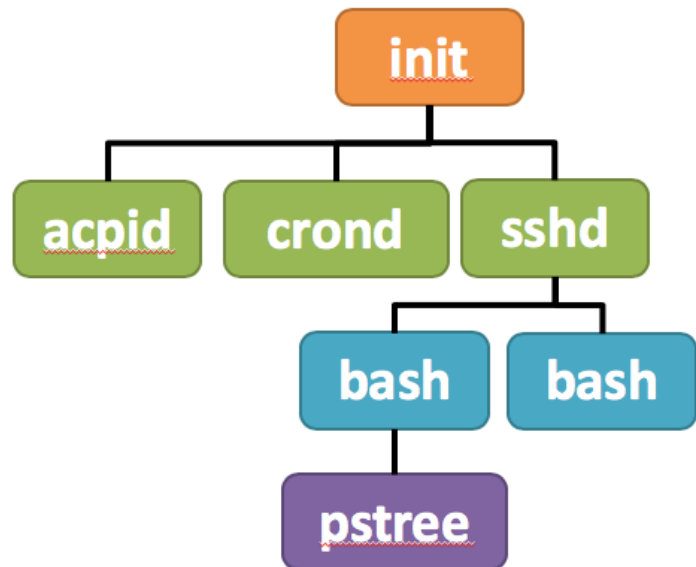
- 1) Since each process is started by another...
- 2) We kinda end up with a process "family tree"

init is the ancestor.

it spawns several **daemons**:
processes which handle
important background tasks.

when you ssh into thoth, sshd
gives you a bash shell.

and then you can run pstree to
see this tree!



Process Identifiers (pids):

- 1) Every process has a unique numeric identifier, its **pid**
 - **init** has a pid of 1, always.
 - You can see the pids with **pstree -p**
 - The POSIX function **getpid()** gets the pid of the current process
 - **getppid()** gets the pid of the current process's *parent*
- 2) You use pids as arguments to process-management syscalls
 - Like **waitpid()**
 - and **kill()**

fork() is weird:

- 1) Let's look at an example of `fork()`
- 2) We can use `strace -f` to watch the syscalls from *both* processes
- 3) When you do a `fork()`...

Parent (pid=1234)	Child (pid=331)
➡ <code>int cpid = fork();</code>	➡ <code>int cpid = fork();</code>
<code>if(cpid == 0) {</code>	<code>if(cpid == 0) {</code>
<code> // child</code>	<code> // child</code>
<code>} else {</code>	<code>} else {</code>
<code> // parent</code>	<code> // parent</code>
<code>}</code>	<code>}</code>

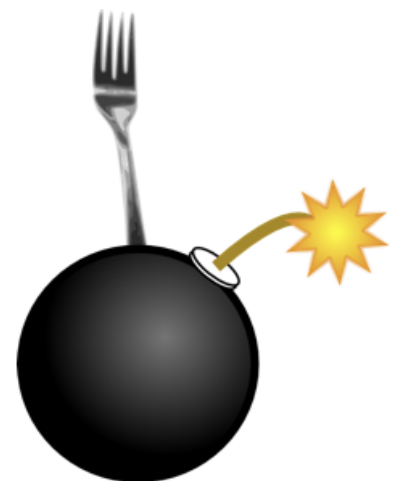
fork() returns the child's pid in the parent... and 0 in the child.

These are Happening "At the Same Time!!":

- 1) I think what really hangs people up on this "it returns different values in each process" thing is that *these processes are running in parallel*
 - 2) BOTH the parent AND the child will start running the SAME code, starting on the line after the `fork()`.
- The only difference is what the return value was.

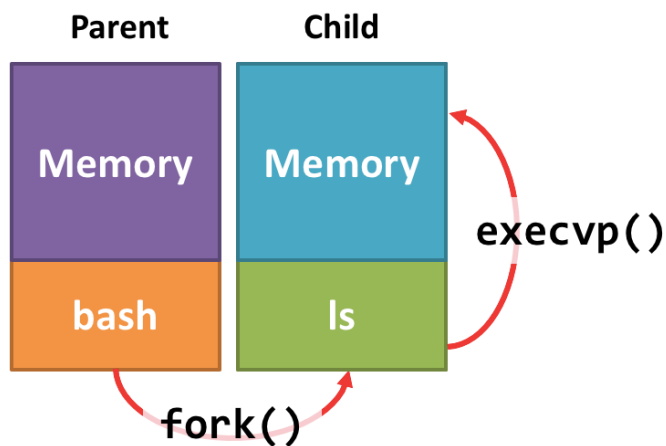
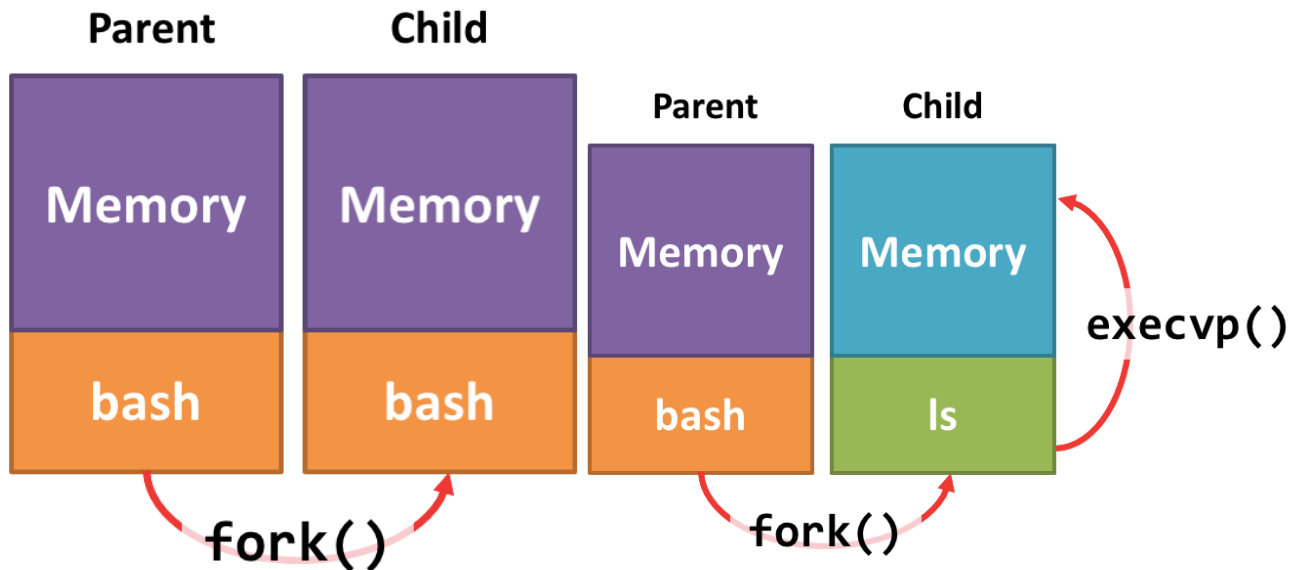
WHAT IF YOU CALLED FORK() IN A LOOP:

- 1) DON'T,
 - lol
- 2) this is a **forkbomb** - so many processes spawn that the computer **grinds to a halt**
- 3) It's *usually* accidental, but can be used as a form of DoS (denial-of-service) attack
- 4) Don't do it on tho
- 5) *A recursive function is still a loop.*
 - I can kill forkbombs but it will make tho unusable for you and everyone else.



Changing Identities:

- 1) Making clones of the *same* process isn't that useful
- 2) We transform a child process with one of the **exec*()** function



it's like sailor moon

when you use an `exec()` function, *the current process's memory is wiped and completely replaced*

if this seems like a weird and roundabout way to start a new process, it kinda is.

but... what if **execvp()** fails?

Error Handling in C:

What Does it Mean:

- 1) If you look at the manpages for `execvp`, it says this:
 - If any of the `exec()` functions returns, an error occurred.
 - ⇒ The return value is `-1`, and `errno` will be set to indicate the error.
- 2) You'll see similar descriptions for `fork`, `open`, `close`, `read`, `write`...
 - A return value of `-1`
 - Something called "`errno`"
- 3) In Java, if you try to open a file that doesn't exist, what happens?
 - `FileNotFoundException`
- 4) But C has no exceptions

In-Band Signaling:

- 1) The C Way™ to report errors is to return some impossible value
 - For *many** POSIX functions, this is a negative number
- 2) In addition, syscalls (and some C library calls!) can put a value in a global variable (!) called `errno` (short for "error number")
- 3) The Proper Way to Handle Errors™ looks like:

```
int fd = open("myfile.txt", O_RDONLY);  
if(fd < 0) {  
    int error = errno; // save errno  
    if(error == EEXIST)  
        // file already exists...  
    else if(error == ... // etc  
}
```

"You Mean We Have to Do That On EVERY syscall?"

1) *YEP.*

- I mean, unless you want a crashy, insecure program ͇_(\ツ)_/͇

2) Be sure to read the errors that a syscall can produce, then:

- Handle the ones you care about; and

⇒ Fail gracefully for the errors you don't handle

3) The simplest method is "whine and exit." use `strerror` or `perror`:

```
int fd = open("myfile.txt", O_RDONLY);
if(fd < 0) {
    perror("couldn't open myfile.txt");
    exit(1);
}
```

Don't Forget to Exit After `exec*`():

1) One of the easiest ways to forkbomb is to have a loop where you fork and `exec`...

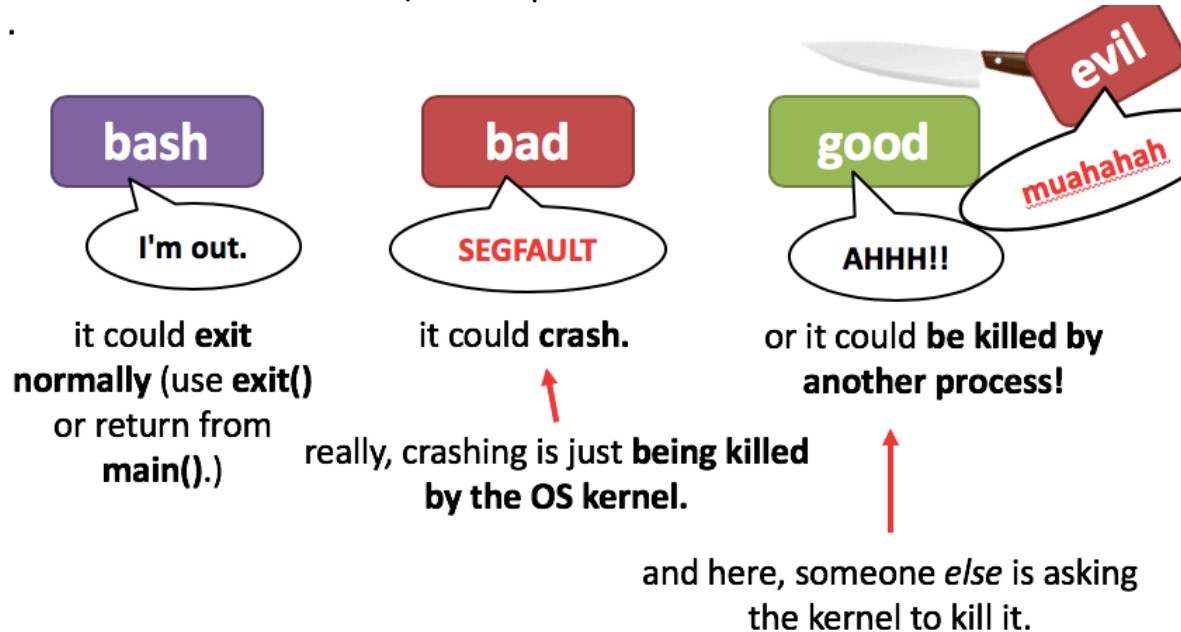
- But you don't `exit()` if `exec` fails.

- SO, DO THAT. ALWAYS.

POSIX Process Destruction:

Apoptosis, Necrosis, and Lysis:

1) There are three main ways for a process to end:



2) "Apoptosis" means "programmed cell death" – a cell decides "it's time for me to die."

3) "Necrosis" means "a cell dying due to some kind of malfunction" – it got too old, or it ingested some poison, etc.

4) "Lysis" means "a cell being killed by another cell" – like a white blood cell or a predatory cell.

Exit Status:

- 1) When your process terminates normally, you can give an **exit status**
 - Return it from main, or pass it as a parameter to `exit()`.
- 2) Just like you can pass parameters on the command line...
 - You can get the exit status with the `$?` variable.

```
$ ls
blah blah blah
$ echo $?
0
$ ls /bogus
No such file or directory
$ echo $?
2
```

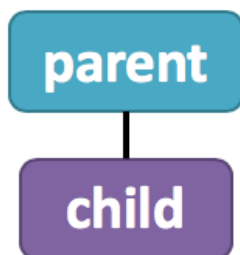
echo is like bash's "printf".

what exit status a program gives is **up to the program** – you have to look at the program's documentation to know what it means.

`./prog a b c -> $?prog(a,b,c)`

What Happens After a Child Process is Spawned:

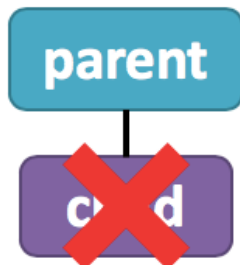
- 1) You've got the parent and the child processes, and...



they could both run forever.



the parent could terminate before the child.



the child could terminate before the parent.

in that case, the child is **orphaned** – and its parent process becomes **init**.



If the Child Terminates Before the Parent:

- 1) The parent can find out *how* the child terminated with `waitpid()`
- 2) Let's see an example of a child process exiting two ways...
 - 17_waitpid.c shows this
 - and your next lab will have you practice with it
 - What are these mentions of "signals"?

POSIX Signals:

Polling vs. Asynchronous:

- 1) Polling means “asking over and over if something happened”
- 2) Asynchronous means “being notified when something happens”

Polling



Asynchronous notification



What's a Signal?

- 1) A **signal** is an **asynchronous** way the OS notifies your program of certain special events
- 2) This happens *outside the normal flow of execution*.
 - Signals are *kind of* like Java exceptions...
- 3) When a signal is sent, the program runs a **signal handler**

```
➔ normal_code();  
nothing_unusual(); AAAA  
whatever(); SIGINT!!!  
if(blah == blah) void handler(int sig) {  
    printf("banana");    printf("O NO");  
                        }
```

but when the handler is done, **execution resumes as normal.**

Good Signals to Know:

- 1) **SIGSEGV** is your best friend, the segvfaul!
- 2) **SIGFPE** is when a mathematical error happens.
- 3) **SIGINT** is the ctrl+C interrupt.
 - By default, it stops the program.
 - But you can make it do whatever you want!
- 4) **SIGTERM**: "hey, time to quit, just letting you know ☺"
- This is clicking the X button on the window
- 5) **SIGKILL**: お前はもう死んでいる
 - The process is instantly killed.
 - This is force closing from the task manager.
 - This signal **cannot be handled or ignored**.
 - ⇒ (the others above can be!)
- 6) "Segmentation Violation" is why it's **SIGSEGV**
- 7) **SIGFPE** = Floating Point Exception, but it happens for integer divides by 0 too...



Handling Signals:

- 1) You can *catch* signals by setting up a **signal handler** of your own
- 2) The **signal()** function is a little old-fashioned but can do this:
 - The first argument is the signal to handle
 - The second is one of these things:
 - ⇒ **SIG_IGN** to ignore the signal;
 - ⇒ **SIG_DFL** to perform the default action;
 - ⇒ or a **function pointer** to a handler function.
 - » This will be called when that signal is sent.
- 3) There is a much more powerful function, **sigaction()**, for setting up custom handlers
 - but my god it has a lot of options.

Sending Your Own Signals:

- 1) You can send signals from your program with the `kill()` function
 - Despite its name, it can send *any* signal
 - You give it a pid and a signal
- 2) You can send signals from the shell with the `kill` command
 - A common useful invocation is:

`kill -SIGKILL pid`

This sends the SIGKILL signal to a process

- This is the command-line "force close" or "end process"!

Kernel - Device Drivers (Part 1):

Resource Allocation:

Who Took My Remote:

- 1) The OS, the overbearing micromanager that it is, **keeps track** of which process is using every resource on the system
- 2) Each process has a **list of resources** that it's using

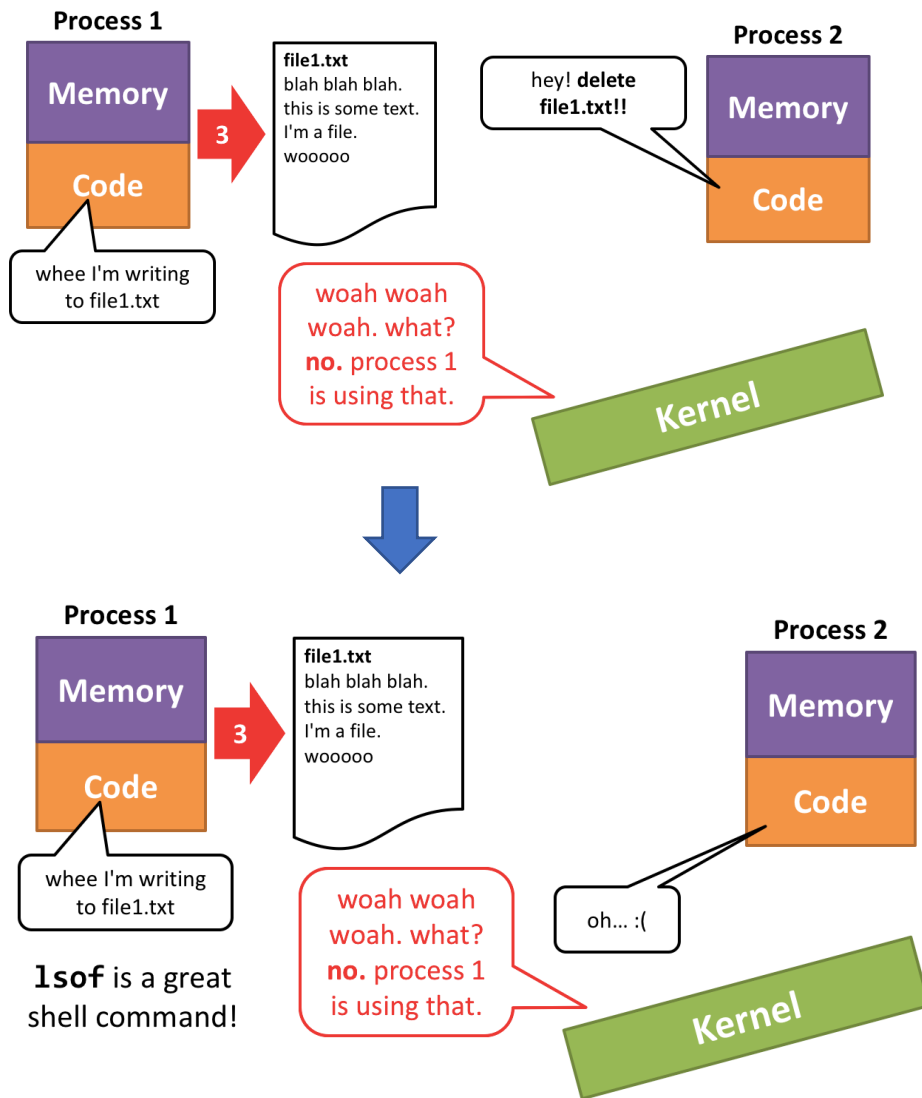
Process 1	index	Resource	
Memory	0	/dev/pts/1	let's look at <u>/dev/fd</u> stdin, stdout, stderr...
	1	/dev/pts/1	
	2	/dev/pts/1	
Code	3	file1.txt	opened another file

↑
this is what the **open()** syscall really returns!

The indexes are like **array indices**!

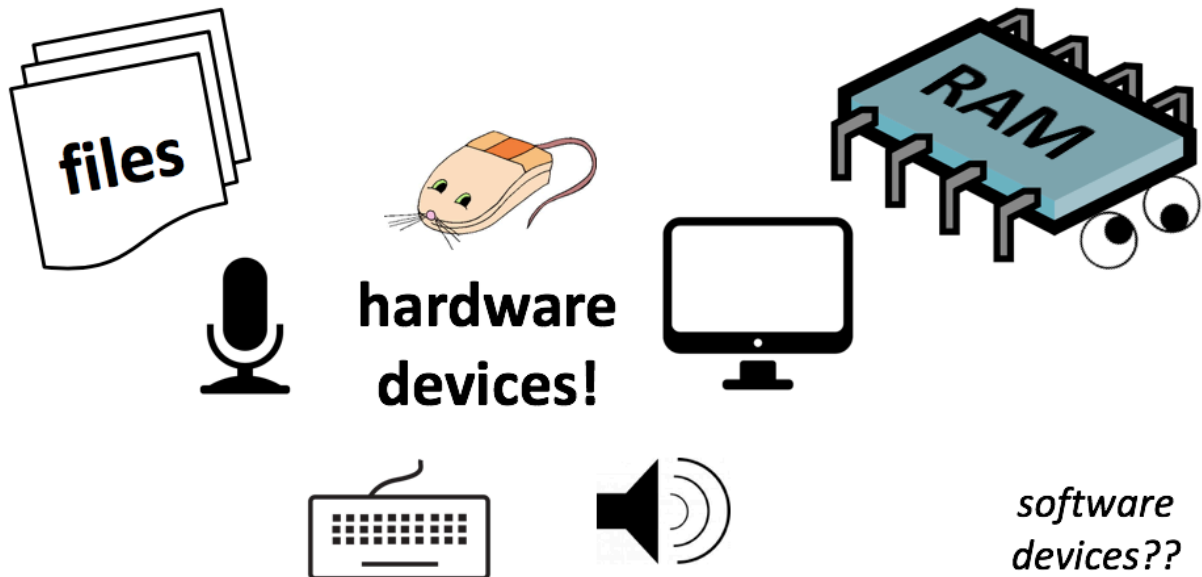
No Touch-a My Remote!

1) The OS is the **arbiter** of who gets to access what things, and when



It's Not Just Files:

1) The OS keeps track of every resource like this



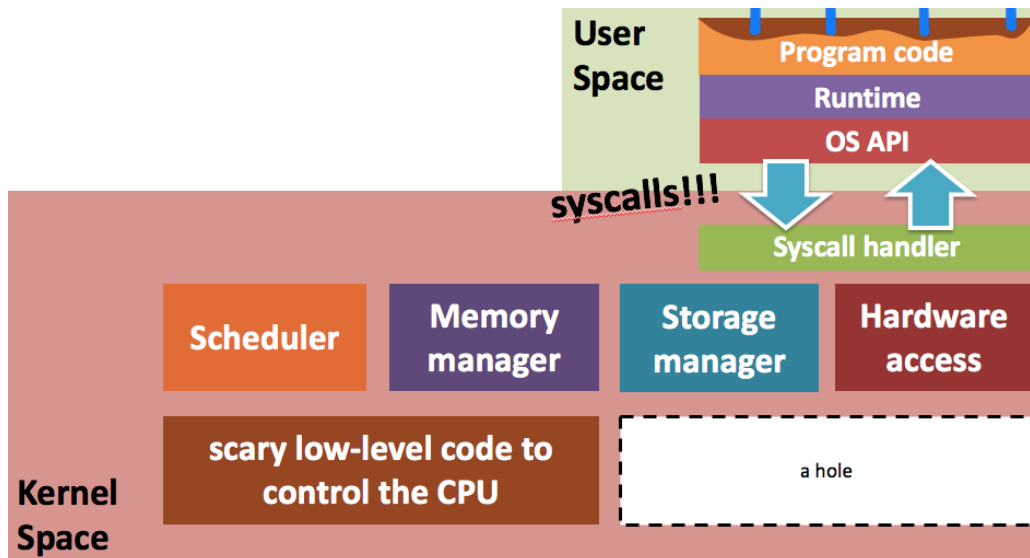
Limits:

- 1) The **sysadmin** (system administrator) has the ability to **put limits** on resources that users/processes can have
- 2) The **ulimit** command lets you see and set those limits
 - `ulimit -a` shows your limits
- 3) **ulimit -u n** is your best defense against forkbombing tho!th!
 - By default, you can run up to 512 processes
 - ⇒ this is absurd
 - When you work on project 4, use e.g. `ulimit -u 15`
 - It only lasts for this login; if you log into tho!th again, you're back to 512 processes

Kernel Modules:

Cakes on Cakes:

- 1) Remember the cake??
- 2) Well the kernel has a whole buncha parts of its own



- 2) Scheduler is responsible for deciding which process gets a turn next
- 3) Memory and storage managers are... pretty self explanatory
- 4) Low-level CPU control code is complicated stuff to control caches, hardware threading support, interrupts etc
- 5) Hardware access is all the stuff that talks to other hardware devices besides the CPU and memory

Plug N' Play:

1) How many kinds of hardware can you think of?



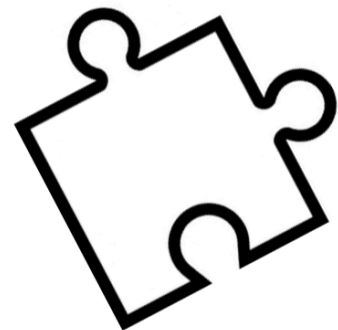
I didn't wanna put a thousand images on here but you get the idea

Intractable:

- 1) There are **thousands upon thousands** of pieces of hardware
- 2) Each one works differently
- 3) It's completely impractical to "bake" code to control that hardware *into the kernel itself*
- 4) The kernel would be hundreds of terabytes
 - and most of it would go unused

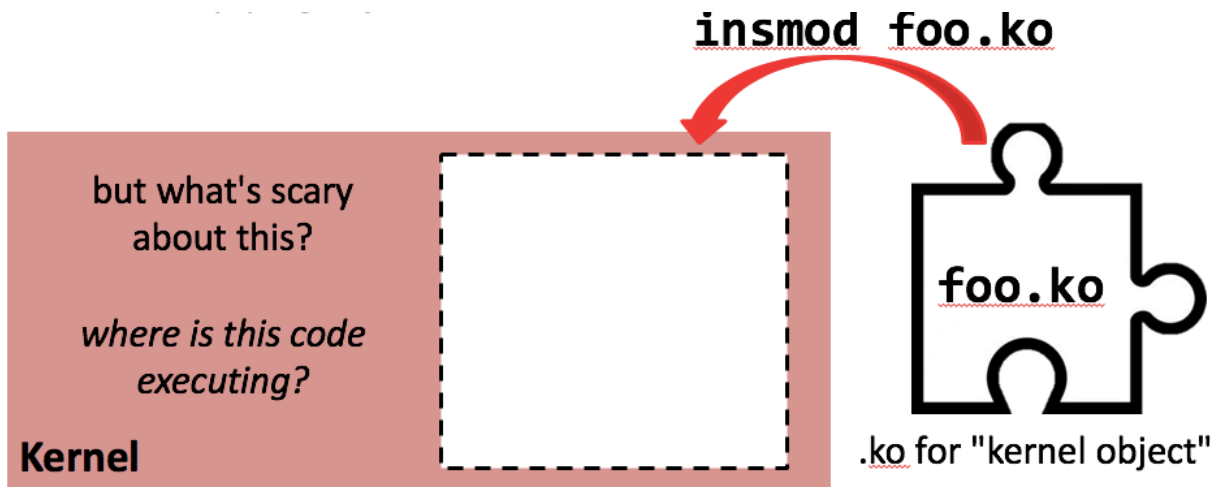
Kernel

well... what did we do
when we wanted to **add
functionality** to a user-
mode program?



Like Plugins, but Way Scariar:

- 1) Many OS kernels support **dynamic loading** of kernel module
 - Essentially, plugins *for the kernel*.

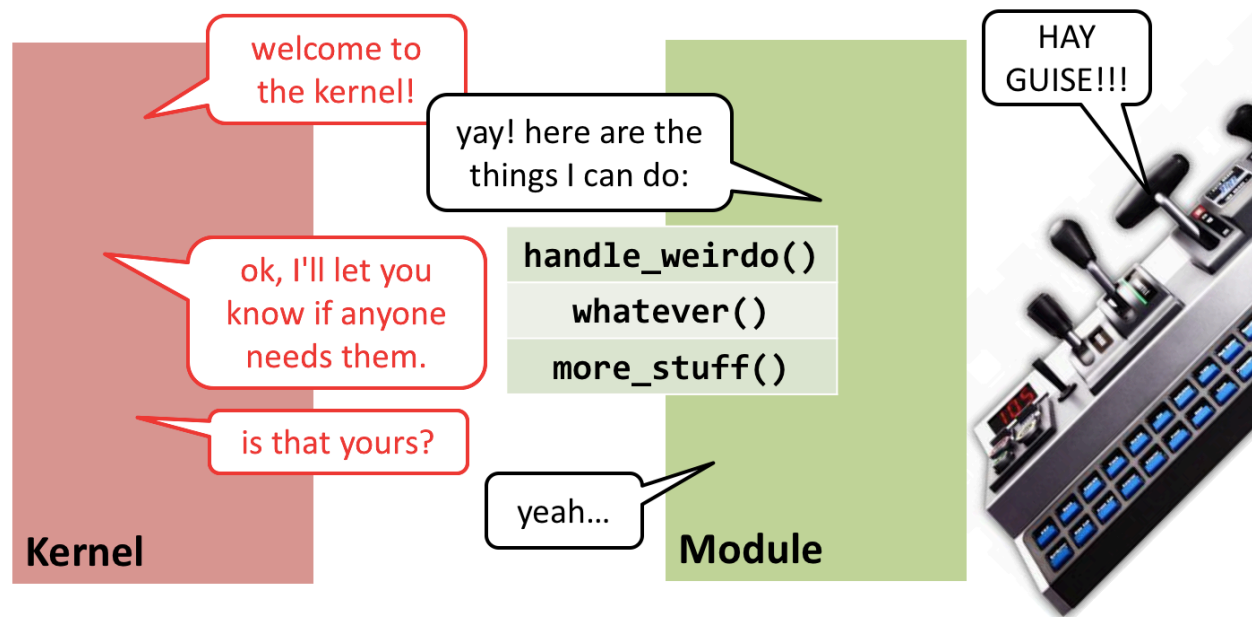


if a kernel module is **evil** or **broken**, you're gonna have a bad time.

- 2) The kernel module IS EXECUTING IN THE KERNEL, IN KERNEL MODE.

How it Works:

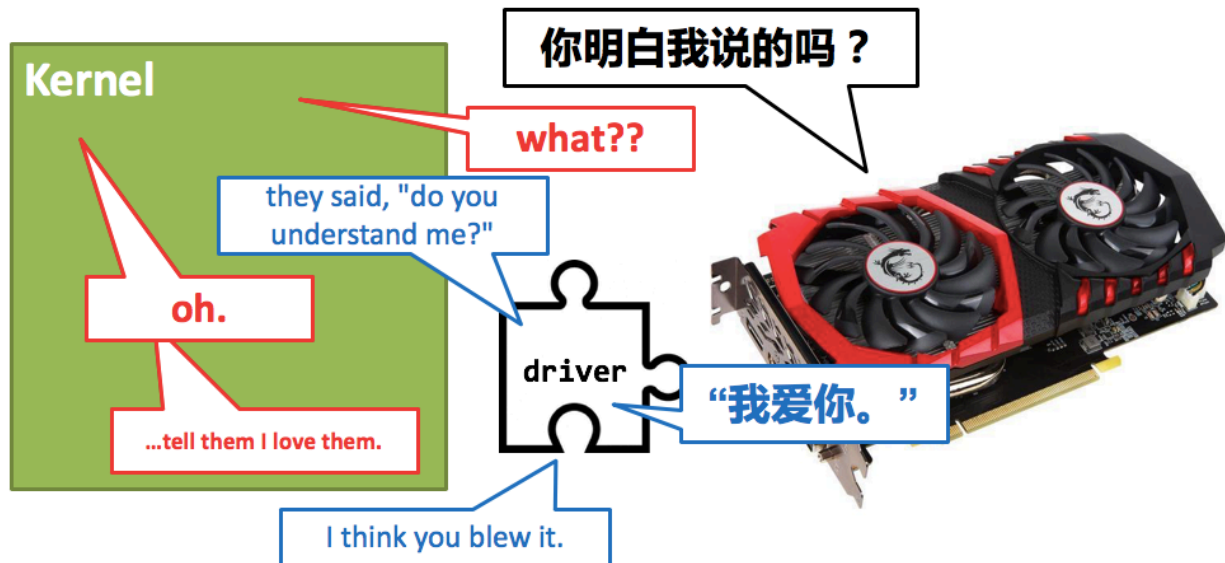
- 1) When a kernel module is loaded...



Device Drivers:

Beep beep honk honk:

1) A device driver *is a kind of kernel module* which allows the kernel to communicate with a piece of hardware. like a translator.

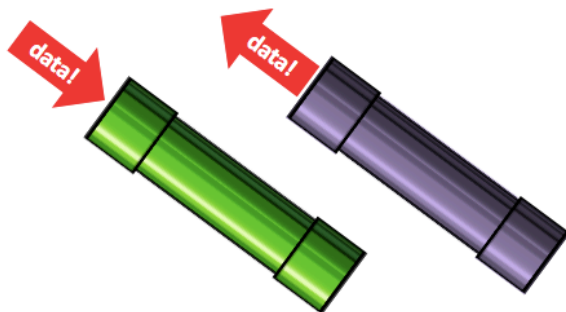


Kinds of Devices:

1) There are two broad classes of devices: **pipes** and **notebooks**.
- Uh, I mean **character** and **block** devices.

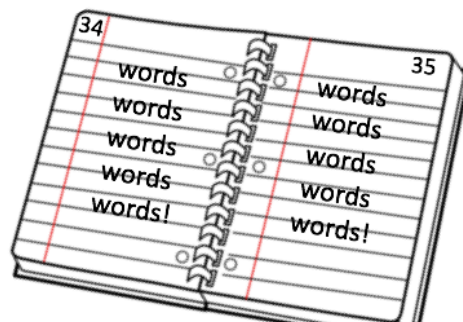
Character

they're like a pipe: **one way**,
and the data goes through **one**
byte at a time.



Block

they're like a notebook: you can go
to any **location** and **access a bunch**
of data at once



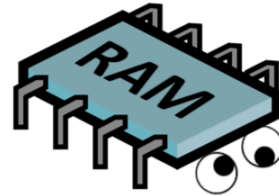
What Do You Think?

1) Which kind of device do you think each of these is?

Character



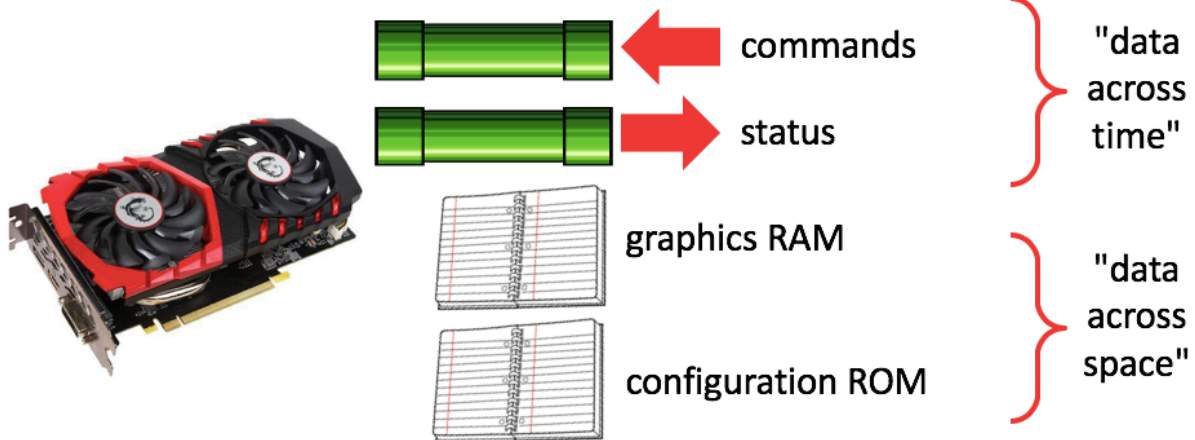
Block



If You Have Trouble Classifying Something...

...either your classification scheme needs to be expanded

...or you should *split up* the complex thing into simpler parts.



one *hardware* device, multiple *software* devices!

Software Devices:

1) It's also useful to have "devices" which aren't really "devices" at all



now we use **terminal emulators** (Terminal, PuTTY)

teletype machines let you log into remote computers then came **terminals**

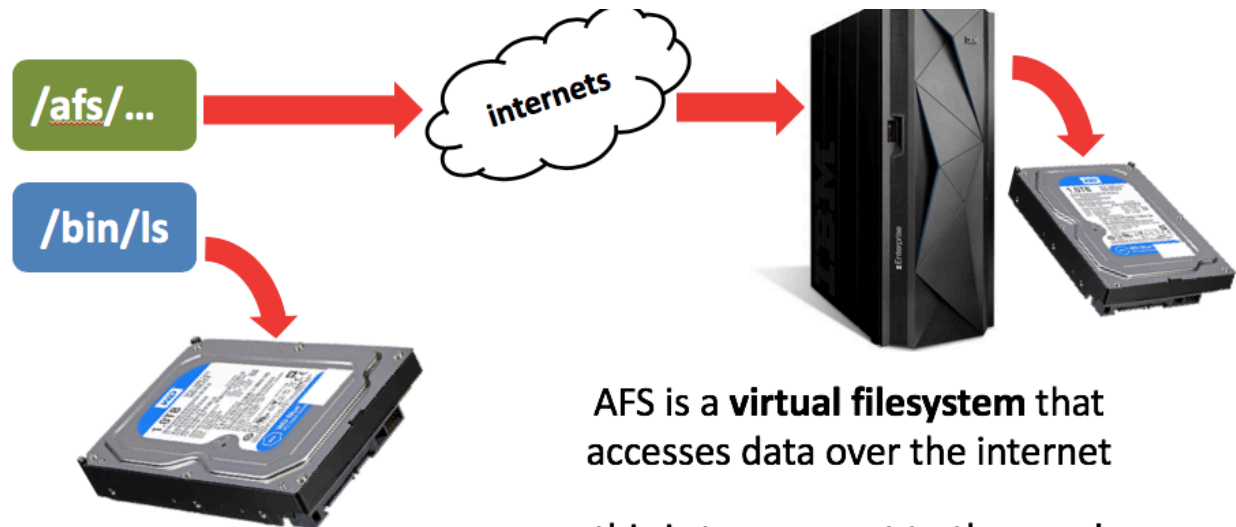
when you log into thoth, it gives you a tty: a **teletype device**

teletypes haven't existed for decades, but the OS doesn't care!

The Kernel's Virtual File Systems:

"File" System:

- 1) The POSIX filesystem is just a way of organizing names.
- 2) Virtual filesystems are directories which *aren't on the hard drive*.



AFS is a **virtual filesystem** that accesses data over the internet

this is transparent to the user!
they're "just" files!

- 3) in the case of AFS, you're still accessing *files*, they just happen to be located elsewhere.
- But the abstraction of the filesystem lets us pretend like they're local files.

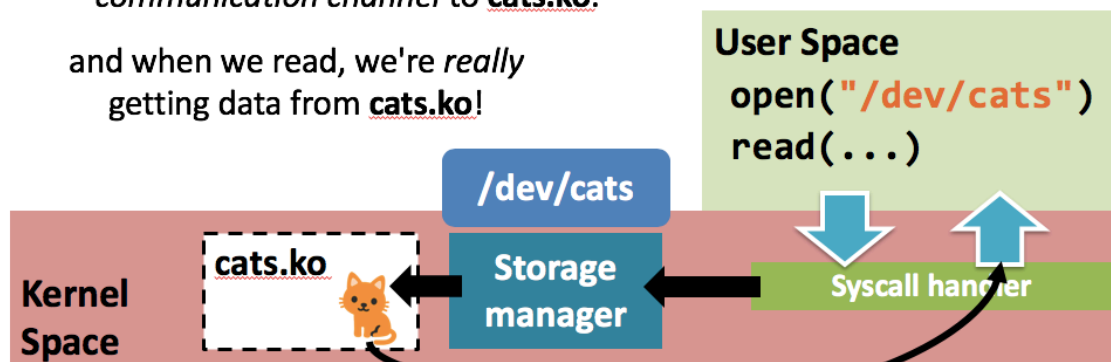
Opening New Doors:

- 1) Syscalls are **fixed**: you *can't add new ones*. but there's a loophole...

when we load cats.ko, it creates a device named `/dev/cats`.

when we open `/dev/cats`, we're opening a *communication channel* to cats.ko.

and when we read, we're *really* getting data from cats.ko!



/dev/

1) /dev/ is the directory of devices, hardware and software.

this column says **block** or character these are the **device numbers**

brw-rw----	1	root	disk	8,	0	May 24 18:30	sda
brw-rw----	1	root	disk	8,	1	May 24 18:30	sda1
brw-rw----	1	root	disk	8,	2	May 24 18:30	sda2
cw-rw----	1	root	disk	21,	1	May 24 18:30	sg1
cw-rw----	1	root	cdrom	21,	2	May 24 18:30	sg2
cw-rw----	1	root	root	10,	231	May 24 18:30	snapshot

the **major** number says
"which **kernel module**
created this device?"

sort of

the **minor** number is used by
the kernel module to identify all
the devices it made

/proc/

1) Oh boy. this has a lot of crap in it

2) The **numbered directories** are processes. they're the pids

- If you list one, you'll see all sorts of stuff...

- You can **cat** or **less** many of these "files" to get info

- The **fd/** directory shows all open file descriptors – there's 0, 1, 2!

3) But then... they started dumping a bunch of **other** stuff in **/proc/**

- You can see the CPU info with **/proc/cpuinfo**

- You can see the OS version with **/proc/version**

- **/proc/modules** shows all the currently-loaded kernel modules

4) Then they were like "maybe we should stop throwing things in /proc

/sys/

1) A more modern way of accessing kernel features

2) It's much more hierarchical.

3) **/sys/module/*/parameters** are module "options"

- These wouldn't make sense as a device

- But these are how your control panel might do things like set how long your screen stays on until it sleeps

4) Most of the stuff in **/sys/** is pretty opaque to a normal user...

In Summary:

- 1) When a kernel module is loaded...
 - It can create one or more **devices** which appear in **/dev/**
 - It can create **parameters** which appear in **/sys/module/**
 - You can **interact** with it through those files
- 2) The kernel itself also places things in **/dev/**, **/proc/**, and **/sys/**
- 3) This is all super flexible because **everything is a file!**
 - No need to add a million syscalls!

Kernel - Device Drivers (Part 2):

Programming and Loading a Kernel Module:

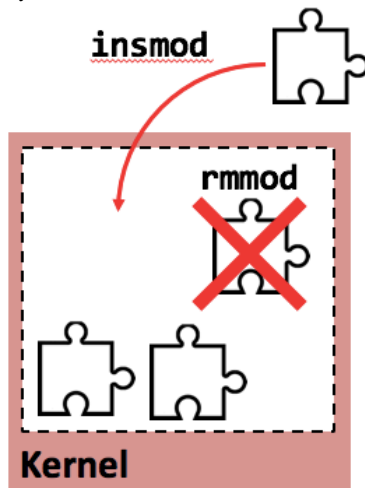
Like User Mode Programs, But Not:

1) Inside the kernel, there are a lot of little differences.

Stack the stack is tiny. alloca() and VLAs are a bad idea. so you <u>wanna</u> use the heap... but there's no <u>malloc/free</u> .	there's no C <u>stdlib</u> at all! #include <stdio.h> there are only the functions that the kernel provides. printf() <u>kmalloc()</u> <u>kfree()</u> <u>copy_to_user()</u>	you also <u>can't</u> use floats. printf("%f", 1.0) that seems like a weird restriction, but it's to make context switches faster. float registers are huge, and <i>not</i> saving them saves time.
--	---	--

The Module Lifecycle:

1) The two commands insmod and rmmod load and unload modules.



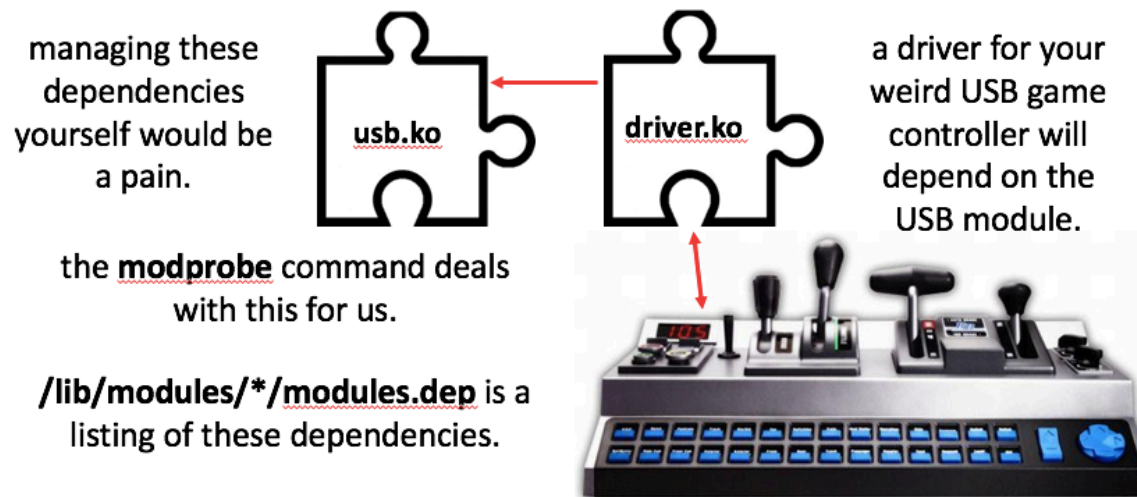
after the module is loaded, the
kernel calls its init function.

rmmod call's the module's exit
function and unloads the module.

we can use lsmod (/sbin/lsmod on thoth) to
get a listing of loaded modules.

Module Dependencies:

1) Just like user mode libraries, modules can depend on *other modules*.



Hello Kernel!

1) Let's look at a very simple module: the hello world module.

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL"); ← Linux is picky about licensing...
static int hello_init(void) {
    printk(KERN_ALERT "Hello, world!\n");
    return 0; ← 0 means success.
}
static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, world!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

these lines tell the kernel what the **init** and **exit** functions are.

Compiling It:

1) The Linux kernel comes with all the include files, libraries, linker scripts etc. needed to compile kernel modules.

2) We just need to use a little Makefile (inside **hellomod.tar**)...

3) and use this command:

```
make ARCH=i386
```

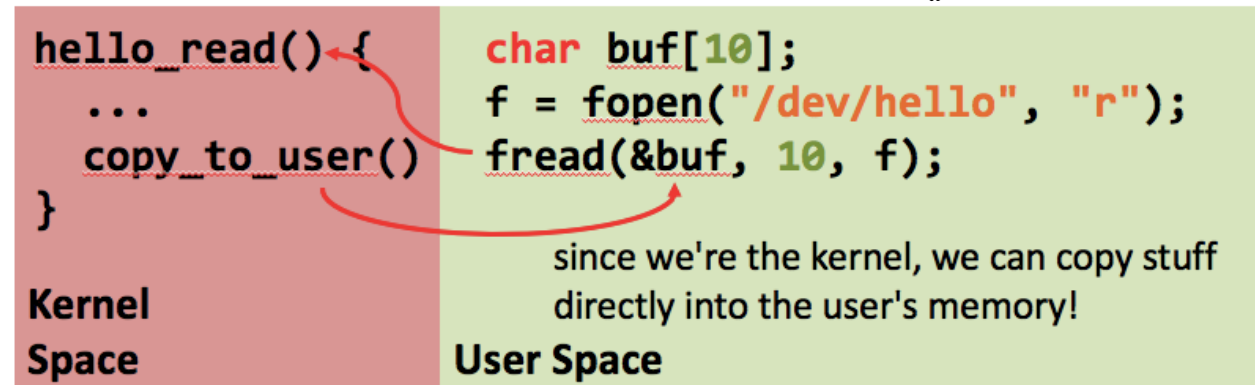
- If you forget the ARCH, it'll make a 64-bit module...

4) Now I can load it into my little Linux VM!

A Software Device Driver:

A More Substantial Example:

- 1) The `hello_dev` example shows a very simple *driver*.
 - 2) In its `init` function, it calls `misc_register()`, a kernel function to say "hey, I offer you a gift of this device."
- The `hello_fops` struct tells the kernel about the device.
 - The `hello_read` function... well, it's the other side of `read()`!



Things to Watch Out For:

- 1) Say we had *two* devices in this module.

```
init() {  
    misc_register(&hello_dev);  
    misc_register(&cats_dev);  
}  
hello_read() {...}  
cats_read() {...}
```

← the *instant* this is loaded...

← this function might be called!!!
even before *init()* finishes!!!

your module is also responsible for
managing its own memory.

if your module **leaks memory**...
well, what's the only way to clean up
a memory leak in C?

- 2) Restart entire computer to clean up memory leak.

Demonstration:

- 1) Let's look at the **hello_dev** example.
- 2) When loaded, it creates a device...
 - Which doesn't actually show up in **/dev/** *yet*.
 - It *does* show up in **/sys/class/misc**.
- 3) If we **cat /sys/class/misc/hello/dev** we get
10:63
- 4) These are the **major and minor device numbers** the OS gave it.
- 5) Now we run this command:
mknod /dev/hello c 10 63
- 6) and if we do this:
cat /dev/hello
- 7) It prints our message!!!

User-Mode Drivers:

Wait, what?

- 1) Doesn't that seem like a contradiction?
- 2) User-mode processes **can't access hardware**.
 - ...*directly*. 🤔
- 3) Remember the **virtual filesystems**?
 - We can read/write the "files" in /dev and /sys to control kernel modules
 - What if we took this further?

Privileges:

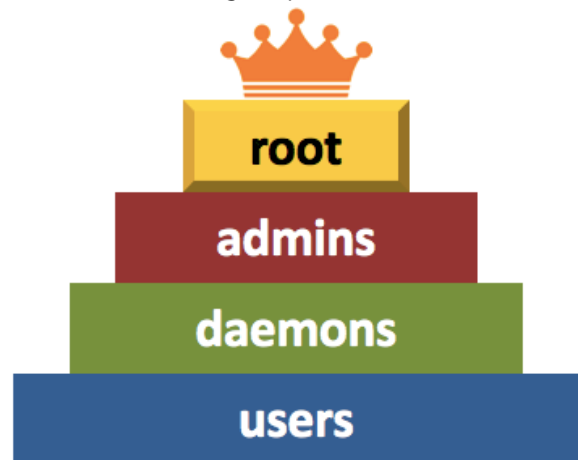
- 1) On a POSIX system, there are many users, and groups of users.
- 2) Each **user group** gets **privileges**: capabilities that other groups may not have.

root can do almost anything.

admins have fewer privileges than root, but more than most users.

daemons are "special" processes that run *in the background* who have higher-than-usual privileges.

privileges like... reading and writing to the files in /dev and /sys!



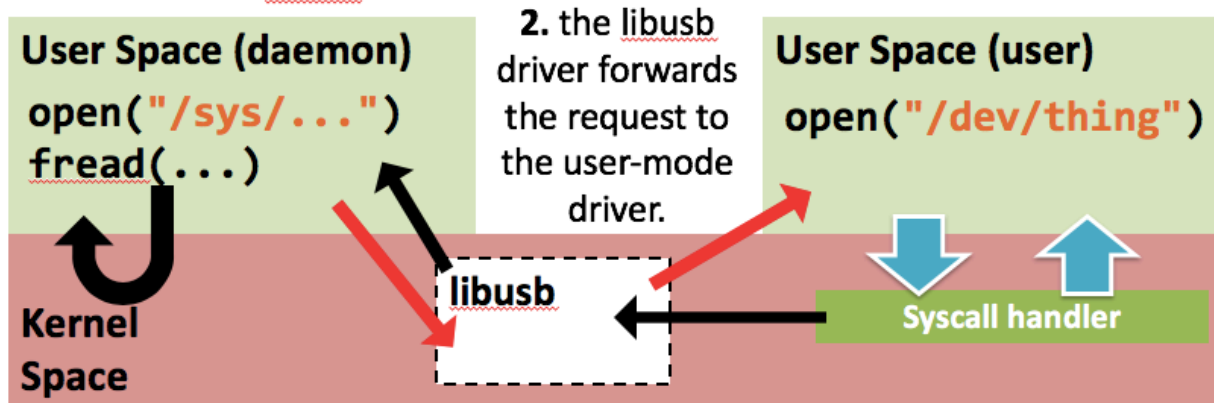
- 3) "Privilege" -> "the things that you never have to worry about because of who you are."

How User-Mode Drivers Work:

- 1) **libusb** is a common user-mode driver framework.
- 2) With it, you can control USB devices *from user-mode daemons*.

3. the user-mode driver handles the request and sends the data back to **libusb**.

1. a program accesses a software device.



4. **libusb** sends the data back to the original process.

Why would we do this?

- 1) Remember what was scary about kernel modules?
- 2) Well, if our user-mode driver is **evil or broken**...
 - Big deal! so what!
 - It'll crash, because **it's a user-mode process**.
 - Your device might stop working, but oh well. Restart the driver!
- 3) They're much **easier to develop**.
 - You just saw how many steps there are to making a kernel module.
 - and you have to have the privileges to install them.
 - In user mode, you can use **any language**, use **any user-mode library**, you can **debug them in gdb**, and **compile and run them much more easily**.

But Of Course There Are Downsides:

- 1) What'd you notice about that diagram?
- 2) *How many times did we cross the user/kernel mode barrier?*

- Once to do the original syscall...
- Once to get up into the driver...
- Once for the *driver* to do a syscall...
- Once for the OS to return to the driver...
- Once for the driver to return data to **libusb**...
- and once for libusb to return to the original process.

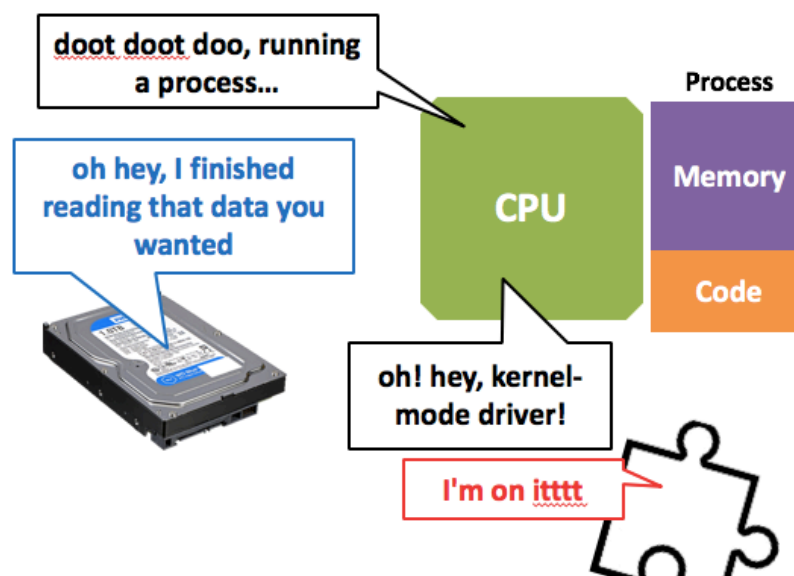
3) CONTEXT SWITCHES GALORE

- This makes them much slower than kernel-mode drivers
- Buuut sometimes it's worth it.

⇒ You don't need crazy speed for a thing that blinks a light when you get an email.

Excuse Me...

- 1) One more thing user-mode drivers *can't* do:



this is a **hardware interrupt**.

they're used for very low-latency asynchronous notifications.

only the kernel is allowed to respond to them!