RECITATION 9

INFO

- Jon Rutkauskas
- Recitation: Tue 12-12:50
- Office Hours: Tue 11-11:50
  Thur 11-12:50
  SENSQ 5806
  (additional hours by appointment if needed)
- On discord: @jrutkauskas
- By email: jsr68@pitt.edu
- Website:
  https://github.com/jrutkauskas/spring2019-449-rec
- Ask me any questions you have!!!

WARMUP POLL

WELCOME BACK

- Hope you all had a good spring break
- Project 3 is done! Lab 5 is done!
- Great job! Project 3 is one of the hardest projects in this class, it's smoother sailing from here.

WHICH OF THESE IS -NOT- TRUE OF DYNAMIC LINKING?

Wastes less space storing the same code in ALL of our executables.

Can patch/fix the code later without updating all the executables we used it in

Allows us to only import the shared objects the user wants or add in optional functionality

Different versions of the code we link can have compatibility issues (e.g., we expect a recent version, but a user only has a .so from 2006 so there might be changed functionality)

Users might not have the shared library at all

I don't know... please explain the answer to me.

WHICH OF THESE IS -NOT- TRUE OF DYNAMIC LINKING?

1. Dynamic linking doesn't copy any of the code of the data we want to link in, essentially it just tells the loader what functions to link in. E.g., we don't have the code for fread in our executables, libc has it, we just link to it
2. When we link to a library, we can update that library without having to change a thing in the executables that link to it, when they are loaded again, the updated version of that library will be used.
3. Importing different SOs at runtime is a property of dynamic *loading*, not linking.
4. There absolutely may be compatibility issues with different versions, or different version requirements (e.g., we have 2 programs that each demand different, incompatible versions of a library)
5. Some libraries have to be installed separately.
   - Windows Users: Ever had to install dozens of "Visual C++ 200X Redistributable"s? – That's this happening. (a necessary evil)

Wastes less space storing the same code in ALL of our executables.

Can patch/fix the code later without updating all the executables we used it in

Allows us to only import the shared objects the user wants or add in optional functionality

Different versions of the code we link can have compatibility issues (e.g., we expect a recent version, but a user only has a .so from 2006 so there might be changed functionality)

Users might not have the shared library at all

I don't know... please explain the answer to me.

RANK THE FOLLOWING OPERATIONS IN CHRONOLOGICAL ORDER (FARTHEST IN THE PAST FIRST)

loading

compilation

running

linking

RANK THE FOLLOWING OPERATIONS IN CHRONOLOGICAL ORDER (FARTHEST IN THE PAST FIRST)

1. Compilation
2. Linking – Statically linked code is copied into your executable
3. Loading – Dynamically linked code is copied into memory with your program
4. Running – Any dynamically loaded code is loaded as requested

HOW DO YOU DYNAMICALLY LOAD LIBRARIES?

No need to! The code is already copied into your executable

It's not possible in C, only higher level languages like Python can do it.

The loader will copy it into your memory when your program starts

Ask the operating system to load the library for you

I don't know... please explain the answer to me.

HOW DO YOU DYNAMICALLY LOAD LIBRARIES?

A. This is Static Linking, not Dynamic Loading
B. C absolutely can do this, we just learned it.
C. This is Dynamic Linking, not Dynamic Loading
D. The only way to dynamically load a library is by asking the operating system to do it for you, you can't just execute any data you want without the OS's permission

No need to! The code is already copied into your executable

It's not possible in C, only higher level languages like Python can do it.

The loader will copy it into your memory when your program starts

Ask the operating system to load the library for you

I don't know... please explain the answer to me.

What type of function is this function pointer defining: `char* (*to_print)(int, int);`

A function pointer called char that takes a pointer to a to_print function and returns an int

A function pointer called to_print that takes 2 ints as arguments and returns a char*

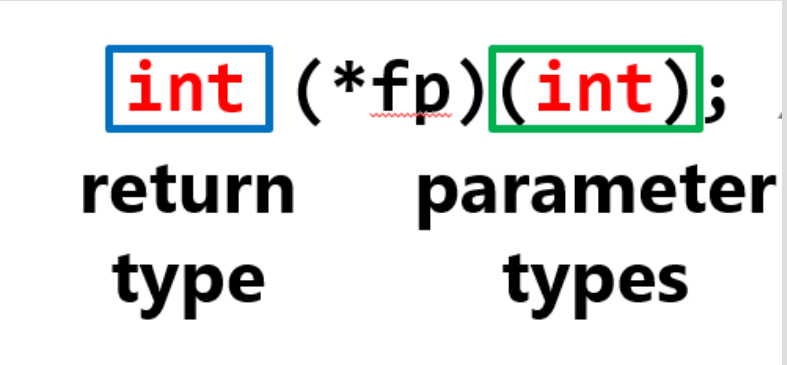
A function pointer that takes a char* argument and an int and returns an int

A function pointer called to_print that takes 2 ints as arguments and returns a char

I don't know... please explain the answer to me.

What type of function is this function pointer defining: `char* (*to_print)(int, int);`

- Recall:



- So, we have the char* as our return type, our pointer is called `to_print`, and we want that pointer to take 2 int arguments

A function pointer called char that takes a pointer to a to_print function and returns an int

A function pointer called to_print that takes 2 ints as arguments and returns a char*

A function pointer that takes a char* argument and an int and returns an int

A function pointer called to_print that takes 2 ints as arguments and returns a char

I don't know... please explain the answer to me.

USING THIS FUNCTION POINTER

```c
#include <stdio.h>

char* yes = "yes";
char* no = "no";

char* all_greater_than_zero_string(int x, int y)
{
    if(x > 0 && y > 0)
        return yes;
    else
        return no;
}
int main()
{
    char* (*to_print)(int, int);
    to_print = &all_greater_than_zero_string;
    printf(to_print(3,3));
    return 0;
}
```

Kind of a -meh- use case, but perfectly good sample code.

YEAH, YOU TOTALLY SHOULD USE TYPEDEFS IF YOU'RE DOING FUNCTION POINTERS FREQUENTLY

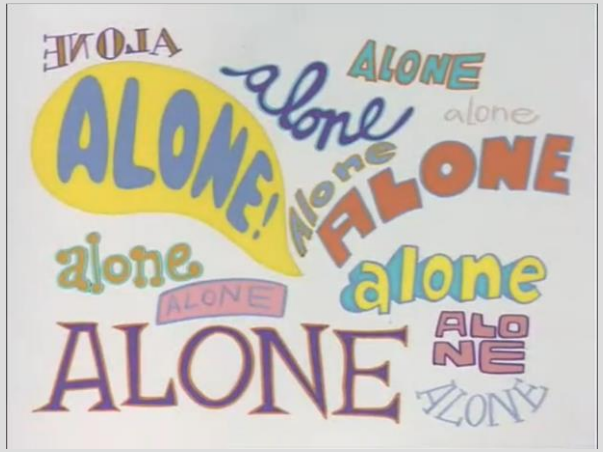
- Or at least try to avoid writing that complicated type all over the place
  - It's a mess to read
- While you might not need to memorize how a bunch of function pointers work, this *is* a useful skill, because they're commonly used and look so dang scary.

WHAT'S A PROCESS?

- No poll for this one. Good old-fashioned discussion?

WHAT'S A PROCESS?

- A process is just an "in-memory representation of a program"
- It's an instance of a running executable
  - All of the code, data in memory (virtual memory), files open, etc.
  - A process thinks it's the only thing running on your computer. It never knows if it's paused for a bit to let other programs run, or where it is in real memory (not virtual), or what else is on the system. It's just the process, and the operating system kernel… all *alone*



WHAT'S VIRTUAL MEMORY?

- A Process's Address space: The area within the realm of all possible 32 (or 64) bit addresses in which your process is allowed to access.
- These addresses aren't actually the physical memory addresses in your RAM
- The OS and your CPU translate those fake 'virtual' addresses into the real memory addresses (using a *page table*)
- Why? Protection. Additional features. Moving memory around. Placing processes in memory onto the hard drive when you're running out of RAM. So much more.
- Lots of 1550 fun stuff here. Feel free to ask me about it but you don't need to know *tooooo* much about how it works for this class.

LAB 6?

- ~~Out sometime? It's on *something*. Don't let it slip past you. I'll be available in OH on Thursday and on Discord anytime, just send a message if I'm offline and I'll get back to you.~~

Last Updated: 12:40am

- Actually it's out now! Surprise!

Last Updated: 2:11am

LAB 6

- We'll be using a compression and decompression library: zlib.
- Compression is the process of taking data, usually a file, and shrinking it down in size to save space. .zip, .gz, and .rar files are *compressed*.
- You'll learn some actual compression algorithms in 1501. We'll just be using someone else's code here.
- How? Dynamic loading!!!!!
- We'll load the zlib library (libz.so) in our code, then extract the functions out.
- It's a really fun lab. Now you know how to add in additional functionality into your programs!

LAB 6

- Recommendations:
  - Take your time, follow along with the lab instructions
  - Break up your code into smaller functions, like one for the dynamic loading, one for calling the compress function, one for calling the decompress function, etc. Do it in parts
  - You'll be writing the compressed files out to stdout. Remember that you can do that with fwrite, and remember that `stdout` is a file that's always available to you.
  - Take it through step-by-step, and don't get discouraged.