CS4414 Recitation 3

A bit about Linux. And a bit about Classes.

02/10/2023

Ricky Takkar

Part 1/2

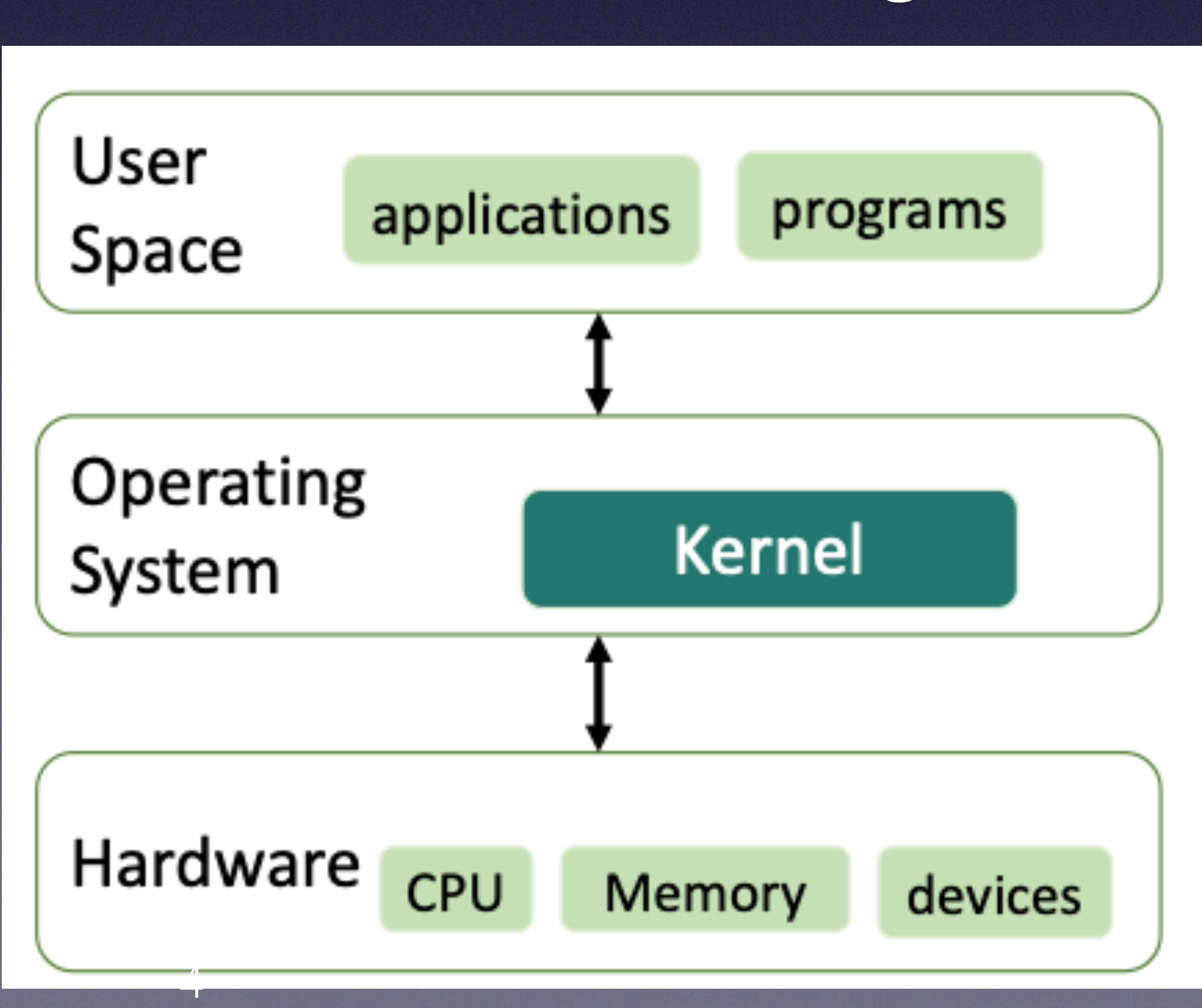
A bit about Linux

What is Linux?

- But first, what’s an operating system (OS)?
  - A system software that **manages** computer hardware, software resources, and **provides common services** for computer programs
  - *Analogy*: If hardware = back-end, then OS = API, and user space = front-end

What is Linux?

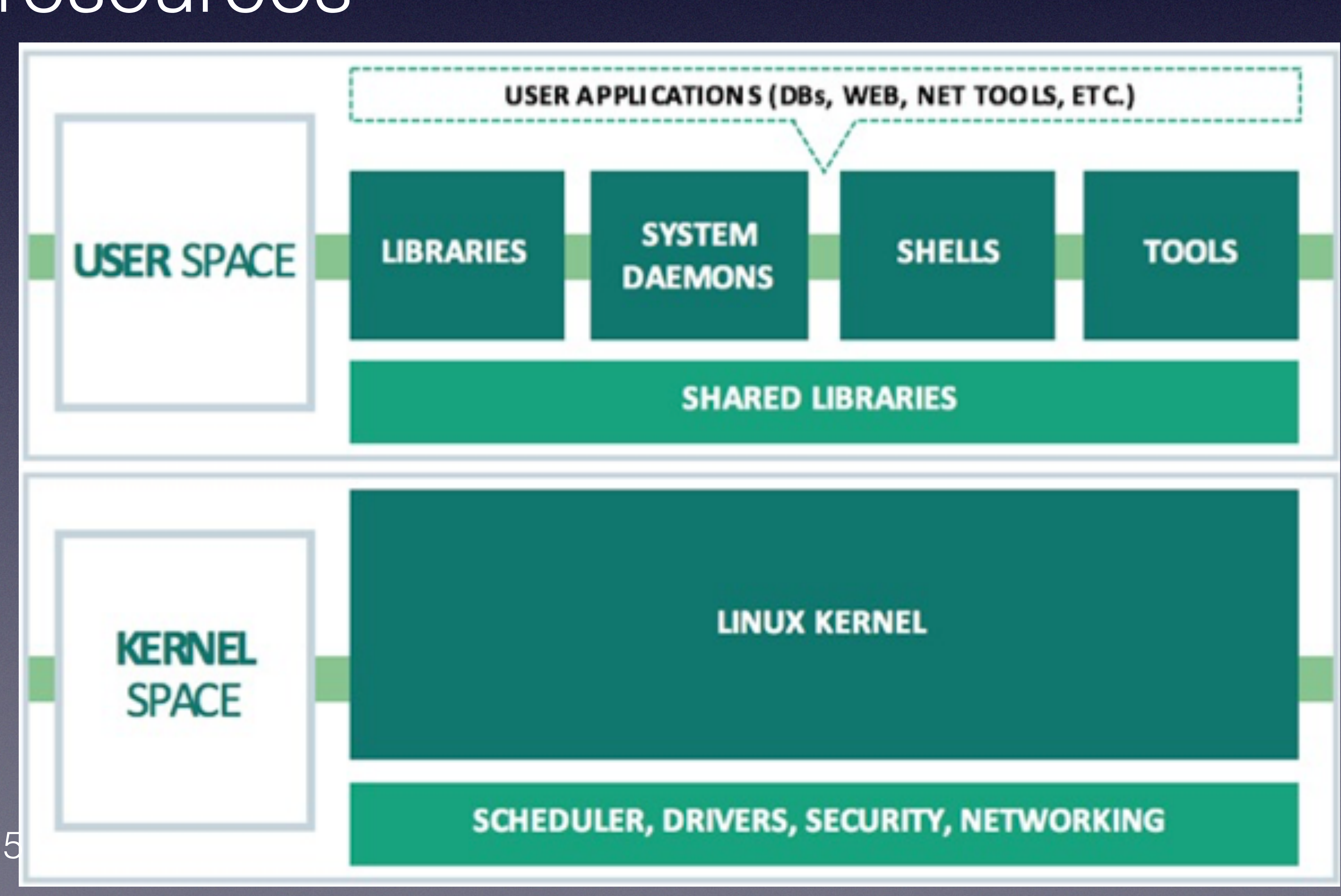
- *If hardware = back-end, then OS = API, and user space = front-end*
- <u>Which component within OS serves as this “bridge” between hardware and user space?</u>
  - Answer: **Kernel**



What is Linux?

- Ok. I understand what an OS is, and I see how it relates to the user space and hardware. But what inside the OS does a kernel do?

1. Access computer hardware resources
2. Resource management
3. Memory management
4. Device management

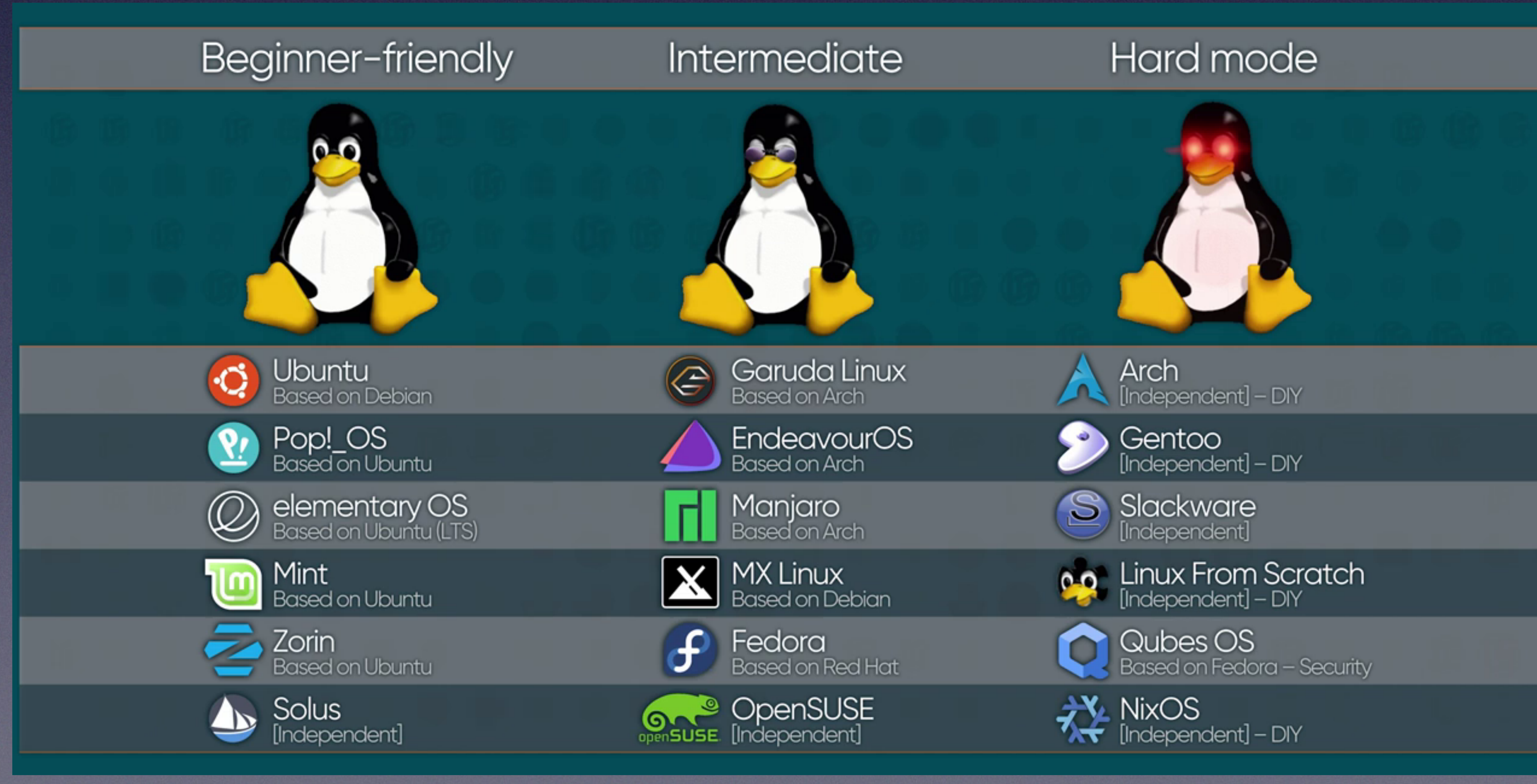


What is Linux?

- Ok, I now understand: (1) what an OS is, (2) how it relates to the user space and hardware, and (3) what a kernel does. But I still don’t know…
  - What on earth Linux is!
- What people actually mean when they say “I run a Linux machine” —> “My machine runs a Linux kernel” or “my server runs Linux” —> “my server runs a Linux kernel”
- But not all Linux machines “look”/“feel” the same…

What is Linux?

- Remember: the kernel is **invisible** to the user. So what are they seeing when they use Linux?
- Various distributions of Linux exist and are used widely, e.g., Linux Mint, Debian, Ubuntu, Fedora, etc…
    - Distro is based on user preference

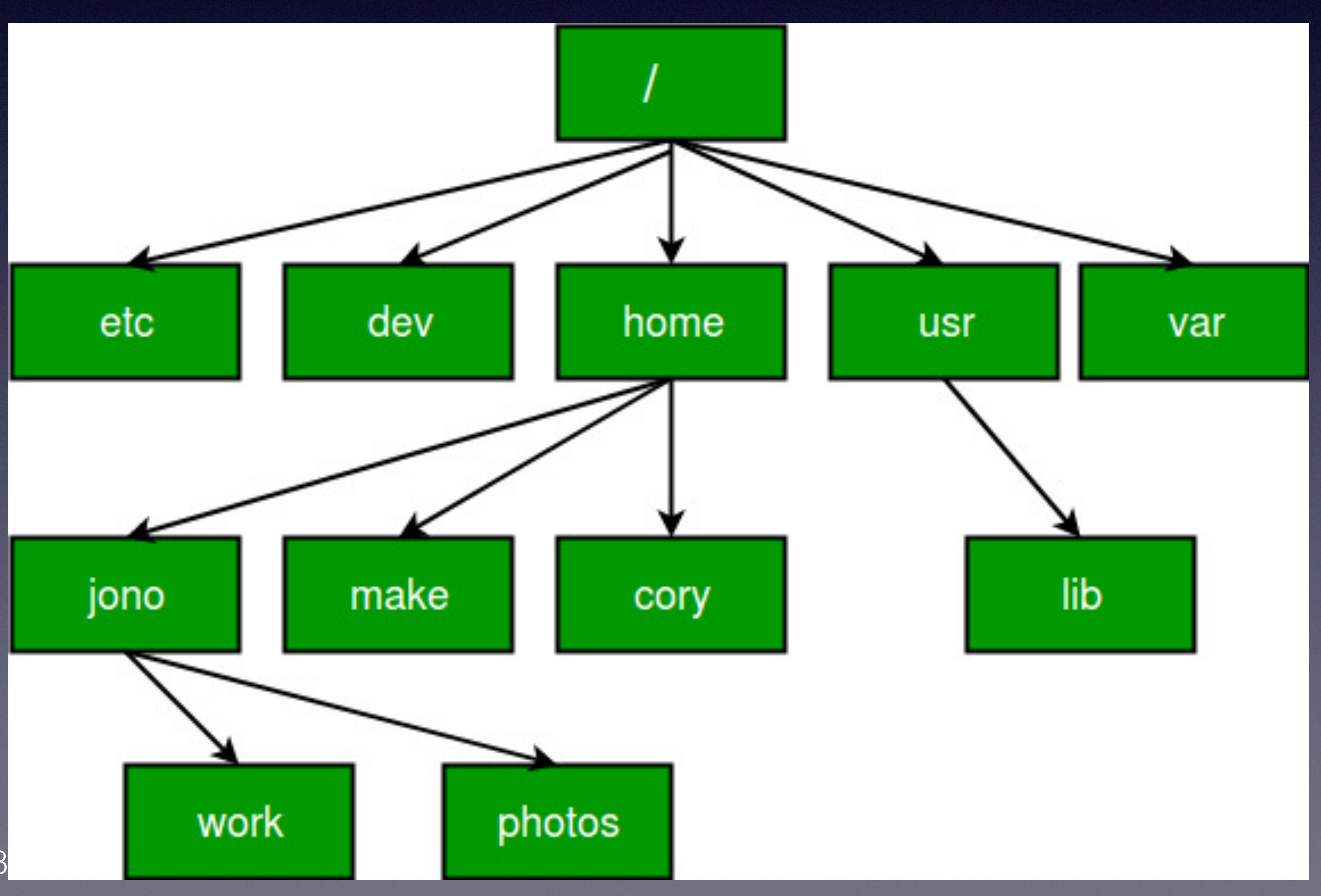


Linux File Structure

Absolute path: Location of file/folder from root directory /

What's the absolute path of **work** folder

/home/jono/work/

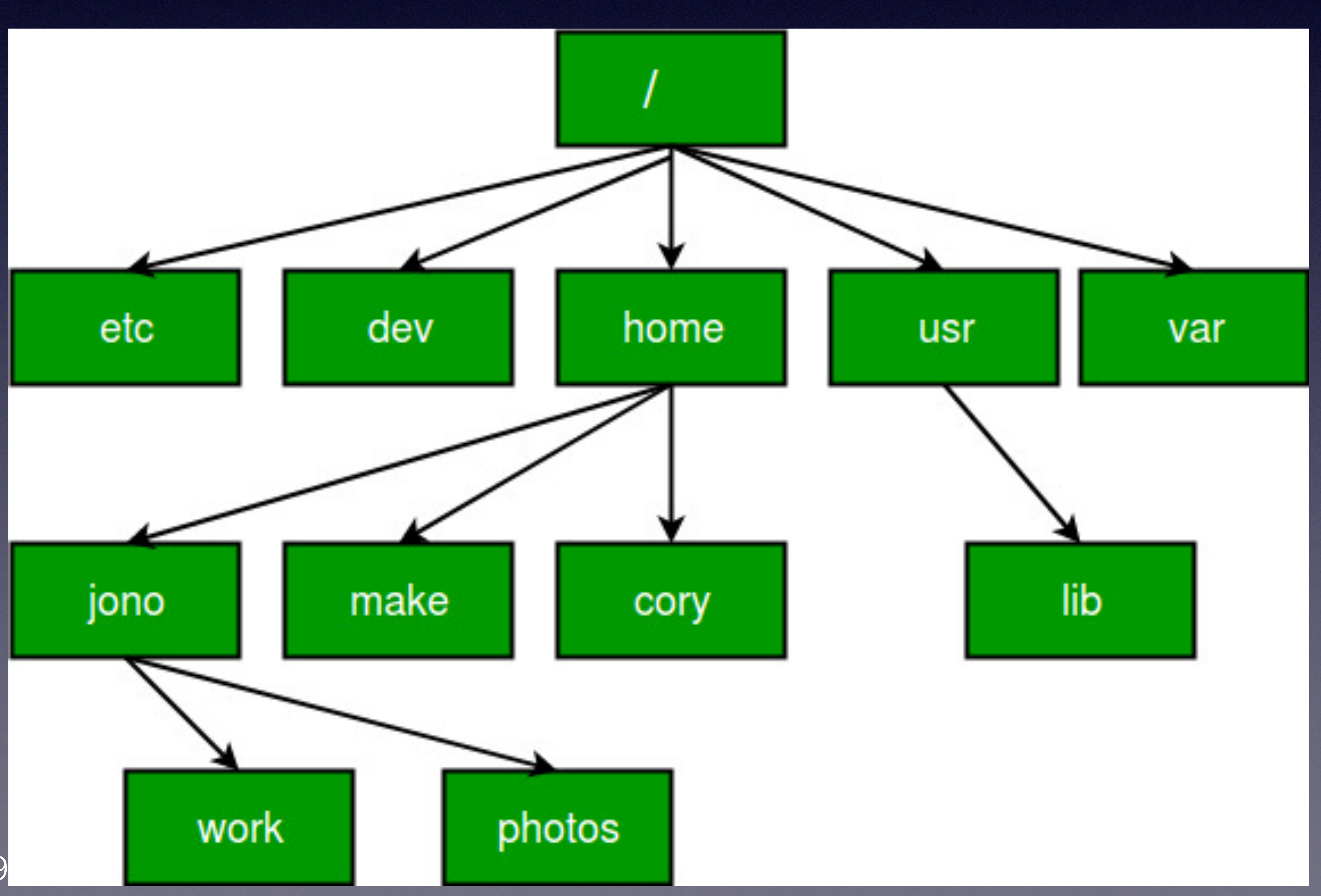


Linux File Structure

Relative path: Location of file/folder from present working directory (pwd)

What's the relative path of **work** folder assuming pwd is /home/

./jono/work/



Directory and Navigation Commands

- pwd get present working directory (pwd)
- ls show what's in current directory
- ls <directory> show what's in specific <directory>
- ls -l '-' is argument pass to command, <l> command indicates long listing
- cd <directory> move to another directory (change directory)
  - cd / change to root directory from anywhere
- mkdir <directory> create a directory

Directory and Files Commands

- echo "This is a test” ‘echo’ prints its arguments back out again
- mv [file1] [directory1] move file1 to directory1
- rm [file1] remove file1
- rmdir [directory] remove empty directory
- rm –r [directory] remove [directory] and all files in the [directory]

Command Line I/O Redirection

- echo "This is a test" > test_1.txt ‘>’ redirect the content to the file
- cat < test_1.txt ‘<’ display the content in file
- cat test_1.txt test_2.txt ‘cat’ can concatenate/link the [file2] and [file1], then display
- ./helloworld > test_1.txt write output from ‘helloworld’ program to file

Basic Commands

- echo $SHELL
  - Within a terminal, there's a shell.
  - Shell is a part of the operating system, defines how the terminal behaves after a command.
  - Examples: bash, zsh (~/.bash_profile set the environment for shell, same for ~/.zsh_profile)
- lsb_release –a 		Display **Linux** distribution
- free -g 		Display how much **space freed/used**

Basic Commands

- which g++ shows which compiler is running
- uname basic info about OS name + system hardware
  - uname -s print kernel name
  - uname -a print all info
  - …
- man uname ‘man’(manual) command like [help] can print details of cmd’s optional argument

Wildcard and alias

- ? Wildcard: matches a single character.
- * Wildcard: matches any character or set of characters
- *Alias*
    - alias clean=‘rm -f *~’ Defile alias of clean
    - touch a~ b~ x~ Create some files with ~ ending

Recap of Lecture Slides

PROGRAMS CONTROLLED BY CONFIGURATION FILES

In Linux, *many* programs use some sort of configuration file, just like cron is doing. Some of those files are hidden but you can see them if you know to ask.

- In any directory, hidden files will simply be files that start with a name like ".bashrc". The dot at the start says "invisible"
- If you use "ls –a" to list a directory, it will show these files. You can also use "echo .*" to do this, or find, or ....

Recap of Lecture Slides

A FEW COMMON HIDDEN FILES

Bash replaces “~” with the pathname to your home directory

~/.bashrc – The Bourne shell (bash) initialization script

~/.vimrc – A file used to initialize the vim visual editor

~/.emacs – A file used to initialize the emacs visual editor

/etc/init.d – When Linux starts up, the files here tell it how to configure the entire computer

/etc/init.d/cron – Used by cron to track periodic jobs

Recap of Lecture Slides

ENVIRONMENT VARIABLES

The bash configuration file is used t[...]bles.

Examples of environment variables

- HOME: my "home directory"
- USER: my login user-name
- PATH: A list of places Ubuntu searches for programs when I run a command
- PYTHONPATH: Where my version of Python was built

Other versions of Linux, like CentOS, RTOS, etc might have different environment variables, or additional ones. And different shells could use different variables too!

EXAMPLE, FROM KEN'S LOGIN

HOSTTYPE=x86_64

USER=ken

HOME=/home/ken

SHELL=/bin/bash

PYTHONPATH=/home/ken/z3/build/python/

PATH=/home/ken/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games

Recap of Lecture Slides

WHEN YOU LOG IN

The login process sees that "ken" is logging in.

It checks the secure table of permitted users and makes sure I am a user listed for this machine – if not, "goodbye"!

In fact I am, and I prefer the bash shell. So it launches the bash shell, and configures it to take command-line input from my console. Now when I type commands, bash sees the string as input.

BASH INITIALIZES ITSELF

The .bashrc file is "executed" by bash to configure itself for me

I can customize this (and many people do!), to set environment variables, run programs, etc – it is actually a script of bash commands, just like the ones I can type on the command line.

By the time my command prompt appears, bash is configured.

Permission

- sudo command for super user to execute (be careful)
- ls -l file shows permission of [file]
- chmod [who][+,-,=][permissions] filename change the permissions
    - chmod u-r filename remove read permission from [file]
    - chmod a-x filename add execute permission to [file]
    - chmod 750 ~/example.txt is equivalent to chmod u=rwx,g=rx,o= ~/example.txt

Permission details

https://en.wikipedia.org/wiki/Chmod

Reference	Class	Description
u	user	file owner
g	group	members of the file's group
o	others	users who are neither the file's owner nor members of the file's group
a	all	all three of the above, same as ugo

Operator	Description
+	adds the specified modes to the specified classes
−	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes

Mode	Name	Description
r	read	read a file or list a directory's contents
w	write	write to a file or directory
x	execute	execute a file or recurse a directory tree

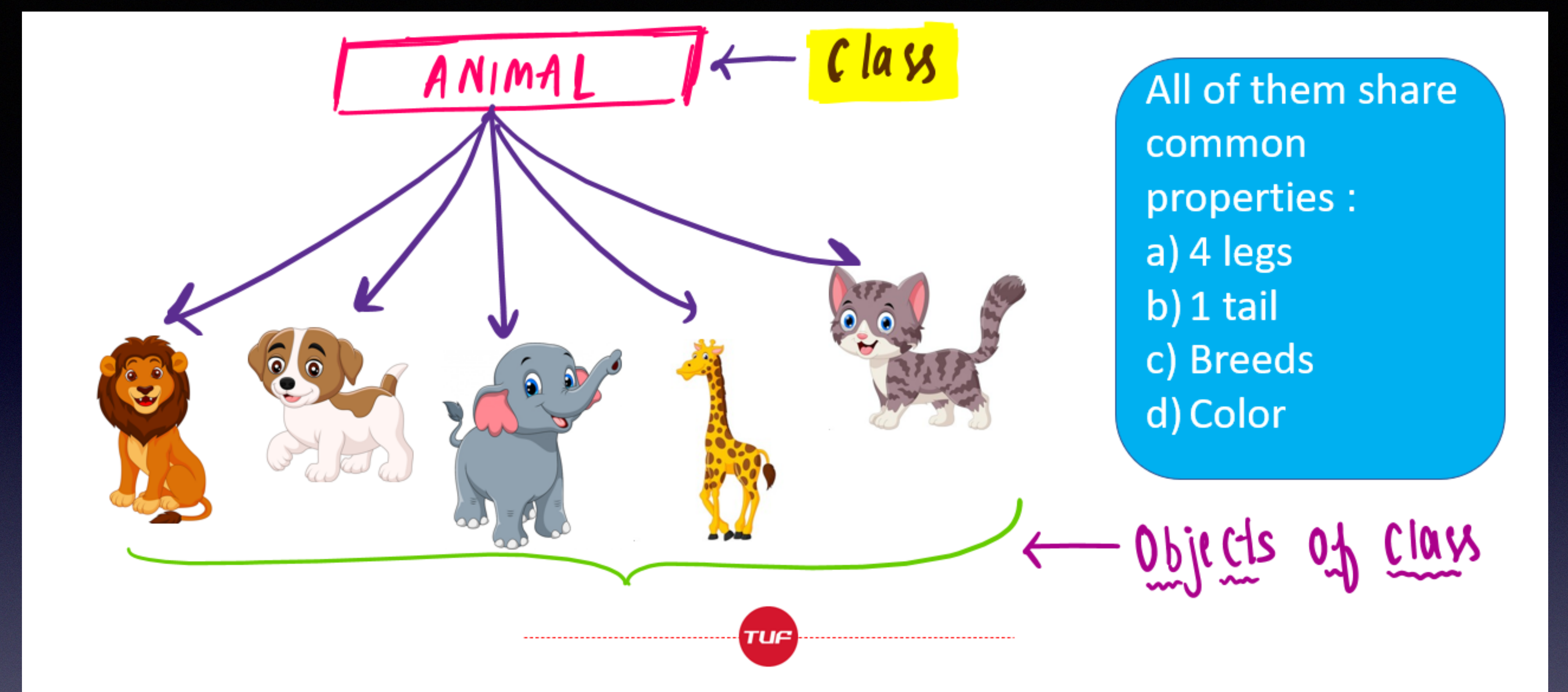
Processes

- ps aux — Show **all** processes
  - ps aux | grep — Grep (search output within input)
- sleep 10 — Sleep for 10 seconds
  - sleep 10 & — Sleep for 10 seconds (in background)
- Ctr+ c — Send signal to terminate process
- ps — Show **only** current user’s running processes

g++ Compilation

- -g — turn on debugging (so GDB gives more friendly output)
- -Wall — turns on most warnings
- -O or -O2 — turn on optimizations
- -o <name> — name of the output file
- -c — output an object file (.o)
- -I<include path> — specify an include directory
- -L<library path> — specify a lib directory
- -l<library> — link with library lib<library>.a

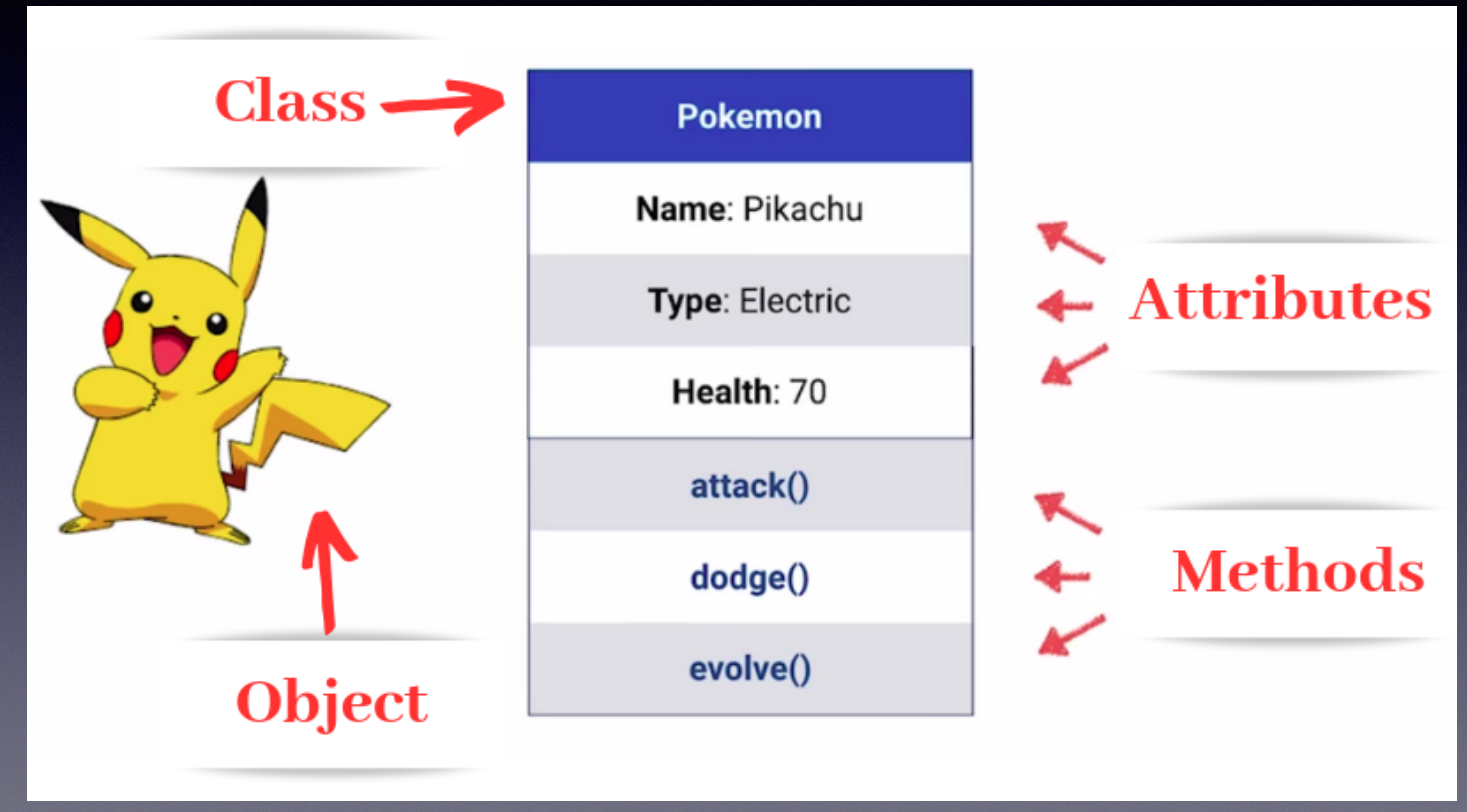
Demo (*optional*, if time permits)



Part 2/2

A bit about Classes

What is a class?



Best practices for classes in C++

- Define class, e.g., MyClass, inside header file with same name as the class (MyClass.hpp)
- Implement class' non-access member functions ("getters") and constructor(s) inside a .cpp file with the same name as the class
- To use MyClass in your program, #include "MyClass.hpp" at the top and compile MyClass.cpp into the project
- Question: Since the class is defined in a header file of the same name, what's the use of another .cpp file with the same name? Why not just implement all class attributes and methods inside the header?

C++'s One-Definition Rule (ODR)

- Only one definition of any variable, function, class type, enumeration type, concept (since C++20) or template is allowed in any one **translation unit**
- But what about this case:

one.hpp — recitation > 3 > pragma > one.hpp > ...

```cpp
struct foo {
    int member;
};
```

two.hpp — recitation > 3 > pragma > two.hpp

```cpp
#include "one.hpp"
```

main.cpp — recitation > 3 > pragma > main.cpp

```cpp
#include "one.hpp"
#include "two.hpp"

int main() {
    return 0;
}
```

What do you think will happen during compilation?

C++'s One-Definition Rule (ODR) Cont'd

Compilation result:

```
rt398@en-ci-cisugcl20:~/recitation/3/pragma$ g++ -std=c++2a -Wall main.cpp
In file included from two.hpp:1,
                 from main.cpp:2:
one.hpp:3:8: error: redefinition of 'struct foo'
    3 | struct foo {
      |        ^~~
In file included from main.cpp:1:
one.hpp:3:8: note: previous definition of 'struct foo'
    3 | struct foo {
      |        ^~~
```

The fix:

recitation > 3 > pragma > one.hpp

```
1  #pragma once
2
3  struct foo {
4      int member;
5  };
```

New compilation result:

```
rt398@en-ci-cisugcl20:~/recitation/3/pragma$ g++ -std=c++2a -Wall main.cpp
rt398@en-ci-cisugcl20:~/recitation/3/pragma$
```

Compiling Classes

- Run “**g++ -o exec_name main.cpp rest.cpp …**”
- Include all the cpp files in the g++ command
- Ignore header files in compilation command as they should be included in the cpp files
- Only one program should contain the main function (in the above example, main.cpp)

Using Classes

- A class is the *blueprint*. Its instance, called an “object” is the *real thing.*
- Objects have their own state, but share class methods and attributes

Classes: C++ vs Java

- Unlike Java, class objects are **NOT** null references in C++!

- This means that when you create an object, all of its internal fields must be initialized (constructed). When the object goes out of scope, its allocated memory must be deallocated. But this **isn't** always handled done automatically.
    - Dynamically allocated memory or use of pointer in class necessitates user-defined destructor

- Each class has at least one constructor and only one destructor (preceded by ~ and without parameters or return type)

Default Initialization in C++

- Example: class myClass { int x; std::string str; };
- Note:
  - Constructor undefined
  - No initialization
- Compiler provides **default constructor** which **default initializes** fields

More on Constructors

- A constructor has the same name as the class and no return type. It can have as many arguments as needed (just like a regular function)
- You can write as many constructors as you need
- E.g.,
  - myClass();
  - myClass(int x, std::string str);
  - myClass(someOtherClass otherClassObject) and so on

(Even) More on Constructors

- Special constructors:
  - Default constructor – takes no arguments
  - Copy constructor (careful with this!) – myClass(const myClass& other);
  - Move constructor – myClass(myClass&& other);
- The compiler provides a default constructor (public) when no constructors are defined
- It also provides a default copy and a default move constructor unless the user defines them

(Just a bit) More on Constructors

- Using the keywords **default** and **delete**, you can enable or disable a constructor
- What if you want to disable the copy constructor? For e.g., you want unique ownership of a resource and don't want it duplicated.
  - myClass(const myClass& other) = delete;
- What if you write a custom constructor that takes some arguments, but still want to keep a default constructor?
  - myClass() = default;

Constructors and Destructor: Creation and Use

TA.hpp

```cpp
#include <string>
#include <iostream>

class TA {                      // class
    public:                     // access specifier
        // attributes
        std::string course;
        std::string name;
        int experience;
        // method
        void printTA() {
            std::cout << "TA " << name << " teaches " << course << " with "
            << experience << " semesters of experience." << std::endl;
        }

        // constructor with parameters
        TA(std::string x, std::string y, int z) {
            course = x;
            name = y;
            experience = z;
            std::cout << "Constructor executed for object with name: "
            << name << std::endl;
        }

        ~ TA() { // destructor
            std::cout << "Destructor executed for object with name: "
            << name << std::endl;
        }
};
```

main.cpp

```cpp
#include "TA.hpp"

int main() {
    // create TA objects and call constructor
    TA ricky("CS4414", "Ricky", 0);
    TA alicia("CS4414", "Alicia", 2);

    // call class method printTA() on objects
    ricky.printTA();
    alicia.printTA();

    return 0;
}
```

Question: What will the output be?

Answer:

```
rt398@en-ci-cisugcl20:~/recitation/3$ g++ -std=c++2a -Wall main.cpp -o TA
rt398@en-ci-cisugcl20:~/recitation/3$ ./TA
Constructor executed for object with name: Ricky
Constructor executed for object with name: Alicia
TA Ricky teaches CS4414 with 0 semesters of experience.
TA Alicia teaches CS4414 with 2 semesters of experience.
Destructor executed for object with name: Alicia
Destructor executed for object with name: Ricky
```

Static Members

Static members of a class are shared by all objects. Similarly, objects declared as static live until the program lives.

Question: Can static variables be initialized using constructors?

Exercise: Create a simple class and create multiple objects of that class in your main function. Utilize a static class member to get the count of objects created.

Access Specifiers

- 3 access specifiers for class variables and methods in C++:
  - **public** - accessible outside the class
  - **private** (default) - inaccessible outside the class
  - **protected** - only accessible to inherited classes outside the class itself. More on Inheritance later…

Let’s code!

References

1. https://data-flair.training/blogs/kernel-in-operating-system/
2. https://www.linux.com/what-is-linux/
3. https://www.geeksforgeeks.org/absolute-relative-pathnames-unix/
4. https://www.learncpp.com/cpp-tutorial/class-code-and-header-files/
5. https://en.wikipedia.org/wiki/Pragma_once
6. https://en.cppreference.com/w/cpp/language/definition