FUNCTION POINTERS, FUNCTORS, THREADING

FUNCTION POINTERS

- What do they look like?
- What are they used for?
- You're already using them and you don't know it.

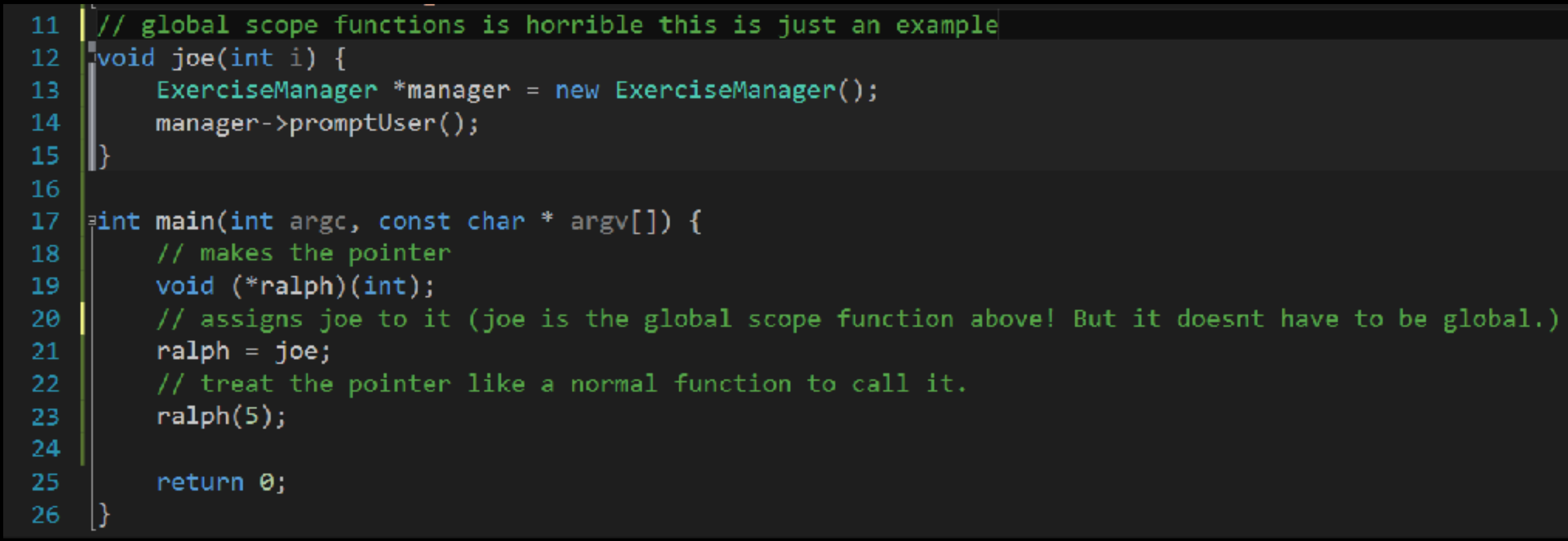
DECLARING A FUNCTION POINTER (TRADITIONAL)

- return type
- arbitrary name
- arguments if any

<u>RETURN TYPE</u> (*NAME) (ARG1, ARG2)

The star is the key.

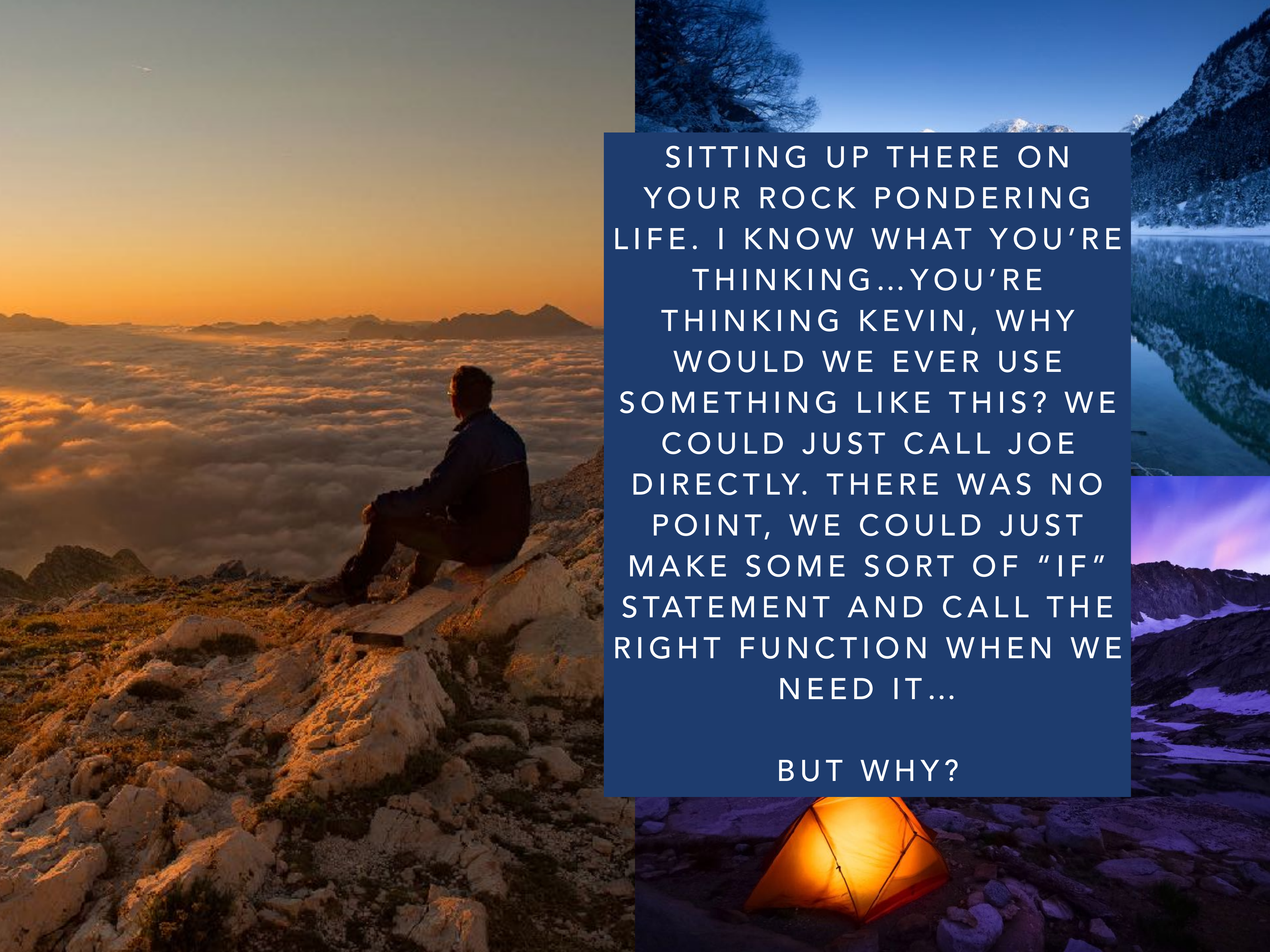
EXAMPLE



```
// global scope functions is horrible this is just an example
void joe(int i) {
    ExerciseManager *manager = new ExerciseManager();
    manager->promptUser();
}

int main(int argc, const char * argv[]) {
    // makes the pointer
    void (*ralph)(int);
    // assigns joe to it (joe is the global scope function above! But it doesnt have to be global.)
    ralph = joe;
    // treat the pointer like a normal function to call it.
    ralph(5);

    return 0;
}
```



SITTING UP THERE ON YOUR ROCK PONDERING LIFE. I KNOW WHAT YOU'RE THINKING…YOU'RE THINKING KEVIN, WHY WOULD WE EVER USE SOMETHING LIKE THIS? WE COULD JUST CALL JOE DIRECTLY. THERE WAS NO POINT, WE COULD JUST MAKE SOME SORT OF "IF" STATEMENT AND CALL THE RIGHT FUNCTION WHEN WE NEED IT…

BUT WHY?

I DELIBERATELY CODED IT LIKE THAT TO MAKE A POINT

FUNCTION POINTERS IN MOST CASES DON'T MAKE SENSE.

USE OBJECT ORIENTED CODING AS YOU'VE LEARNED.

HOWEVER, THERE IS A COUPLE THINGS THAT THEIR GOOD AT...

CALLBACKS

Look at this constructor

```
BUTTON(CHAR * TEXT, INT X, INT Y, VOID (*CLICK)());
```

This way the Button class doesn't need to be subclassed. Every time you want to add a slightly different action, you can just make a new button and give it a different function. Logically, you can write the code for the custom action where it belongs.

SaveManager can have the save() not some custom class SaveButton: public Button {} abstract class shit code.

BUT WAIT, I THOUGHT THAT WAS GOOD CODE!
INHERITANCE AND SUBCLASSING, YOU TOLD US THAT WAS OK!
I'M SO CONFUSED!!!

DON’T WORRY,
THIS IS THE DIFFERENCE BETWEEN KNOWLEDGE AND WISDOM. FIGURING OUT THE CASES WHERE ONE IS SUPERIOR AND WHEN TO USE IT.

Short aside, I mentioned never needing to subclass button… This probably should of been part of the inheritance slides but…

If you use the keyword “final” the class cannot be subclassed.

SO - ALL THAT COOL STUFF YOU JUST LEARNED, THERE ARE PROBLEMS WITH IT.

- They don’t support arguments or return values that use generics (template T thing, aka std::vector, or modern pointers.)
- They are stateless, must be managed elsewhere. (aka no use of the “this” keyword).
- Memory management and capture is complicated.

SO WHERE DO WE GO FROM HERE?

STD:: FUNCTION

Think of it like an improved function pointer.

Or alternatively templates. But that’s a whole different lesson. We’ll get there, just not today…

<T>

IN CLASS EXERCISE

Let's say in your dream job you were tasked with creating a system for a turn based strategy game. Where the user can cancel a chain of actions 1 by 1 until they are back at the beginning of the game.

Get into groups 4-5

PSUEDO code me some ideas on how you would tackle this problem? Try not to google it; see how inventive you guys can be.

STD::FUNCTION

std::function<void (int)> someName = some method that matches the signature of void (int)

like..
void print(int){}
defined elsewhere.

You can specify whatever you like the for the signature.

- So, std::function
- works a lot like function pointers
- but they have this added feature
- called bind
- bind lets you how do you say, inject the values into the function you are calling

SO YOU TAKE A VECTOR (ARRAY) Vector<std::function> v

THEN YOU ADD TO YOUR VECTOR THE REVERSE COMMAND

```
v.push_back( bind(reverse of whatever the user did))
```

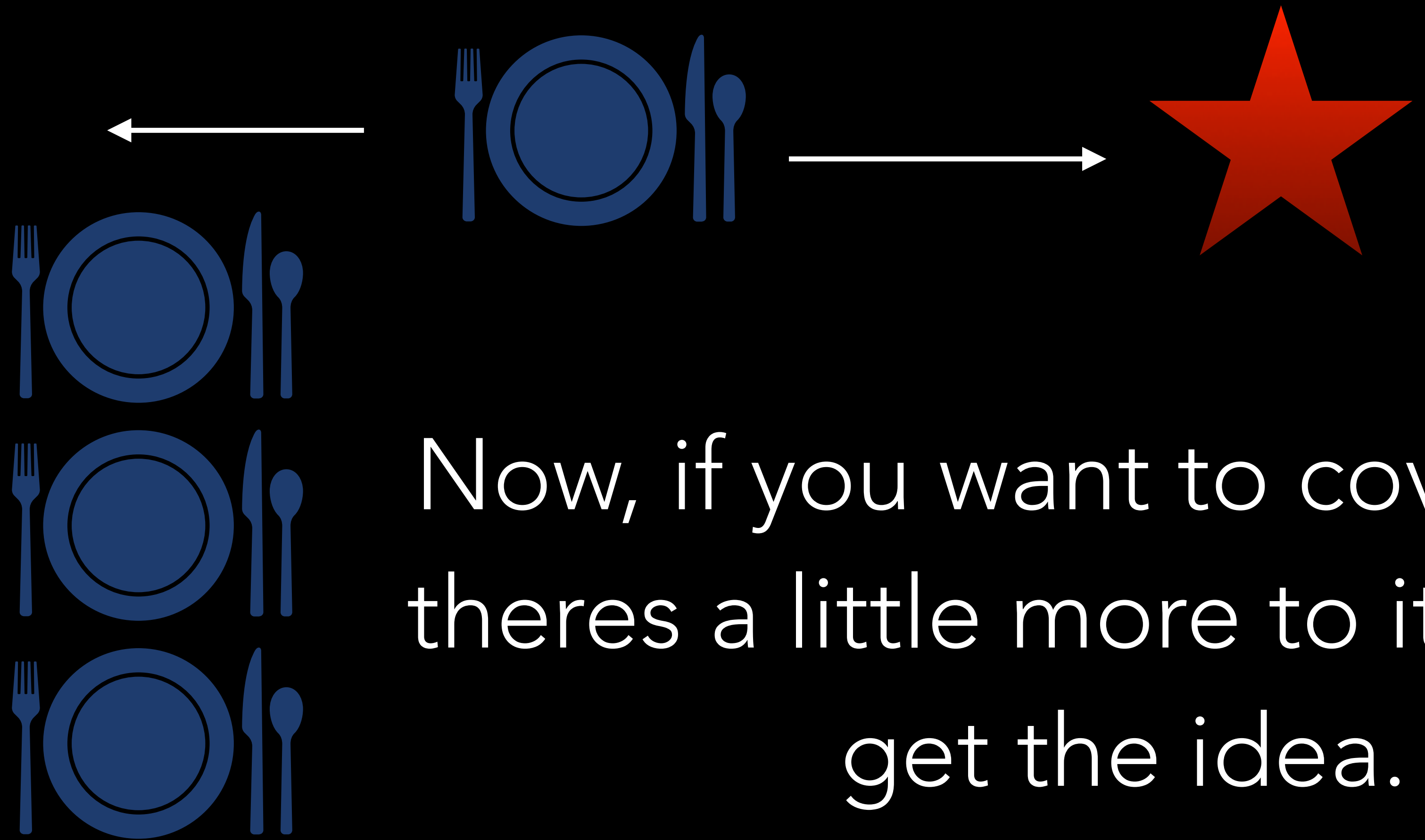
WHEN THEY WANT TO UNDO

```
std::function haha = v.pop_back()
```

GET THE MOST RECENT REVERSE ACTION OFF THE VECTOR

```
haha();
```

THEN RUN IT



Now, if you want to cover Redo, theres a little more to it, but you get the idea.

SMART FUNCTIONS

SO SOME ALGORITHMS CAN TAKE A FUNCTION AS AN ARGUMENT

Either with a function pointer, std:function or lambda function.

FOR EXAMPLE, THE SORT FUNCTION

```
double fData[] = {19.2, 87.4, 33.6, 55.0, 11.5, 42.2}
int main() {
    sort(fData, fData+6, greater<double>());
    for (int i = 0; i < 6; i++{
        cout << fdata[i] << ' ';
    }
    return 0;
}
```

THE THIRD ARGUMENT IS A FUNCTION.

THE POWER OF THIS ISN'T IMMEDIATELY APPARENT.

- Now you can write functions to help you sort objects that the computer can't easily compare 1 to 1.
- Theres other tools than sort; you can filter, transform, find, and many others; targeting exactly what you're interested in with functions you pass as arguments.

THANK YOU FOR YOUR PATIENCE.

PLEASE TAKE A 15 MINUTE BREAK.
SUPER COOL TOPIC WHEN WE GET BACK.

FUNCTORS

FUNCTORS

- You can actually overload the () operator
- So your classes can be treated like functions in the right circumstances.
- I want to cover operator overloading in a separate lecture but essentially…

```
class Matcher
{
   int target;
   public:
     Matcher(int m) : target(m) {}
     bool operator()(int x) { return x == target;}
}

Matcher Is5(5);

if (Is5(n))    // same as if (n == 5)
{
}
```

Ok now for the really cool stuff.

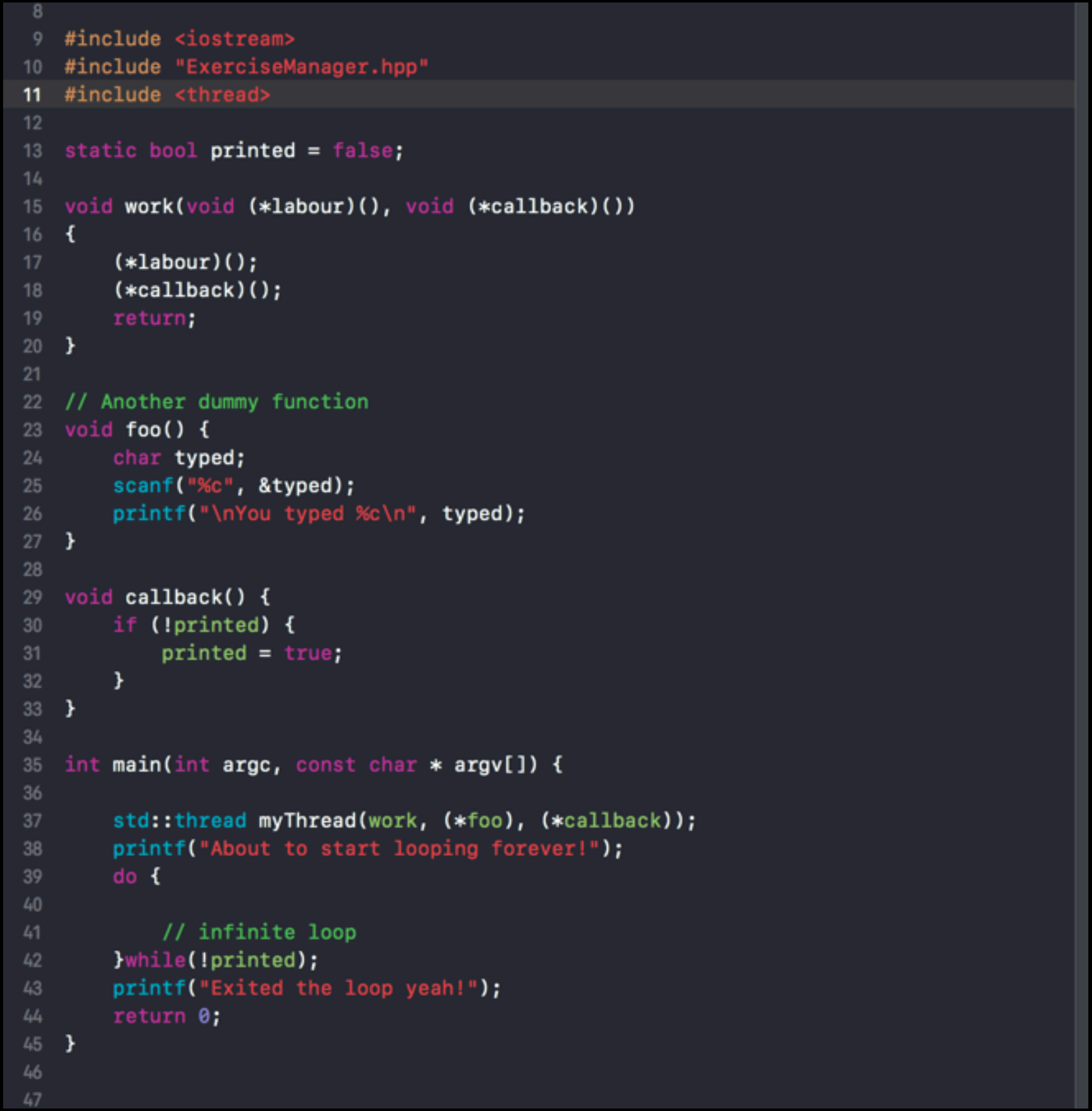
MULTITHREADING

FIRST THING YOU NEED TO KNOW…

IT KINDA SUCKS IN C++.

STD::THREAD

- super simple
- super dangerous



```
#include <iostream>
#include "ExerciseManager.hpp"
#include <thread>

static bool printed = false;

void work(void (*labour)(), void (*callback)())
{
    (*labour)();
    (*callback)();
    return;
}

// Another dummy function
void foo() {
    char typed;
    scanf("%c", &typed);
    printf("\nYou typed %c\n", typed);
}

void callback() {
    if (!printed) {
        printed = true;
    }
}

int main(int argc, const char * argv[]) {

    std::thread myThread(work, (*foo), (*callback));
    printf("About to start looping forever!");
    do {

        // infinite loop
    }while(!printed);
    printf("Exited the loop yeah!");
    return 0;
}


```

THREADS CAN BEHAVE TWO WAYS.

- BLOCKING - main thread waits for your thread to finish before continuing.
- NON-BLOCKING - main thread keeps going while your thread keeps going in essence.
- To enable blocking on a thread call its .join(); method
- Theres no guarantee how long your thread will have process or when process will switch to another thread.
- However you can set priorities on threads.

- There are some key concepts we should talk about.
- Mutex - like a librarian that monitors access to resources like standard output.
- Semaphore - like a lock or variable that acts like a gate keeper.
- In the example I wrote on the previous slide, theres big issues with it if the program was more complex it could of been printing to standard output when my other thread was as well.
- ANWhDen you COUuseLD threSEadsE you nWEIReedD to stOUTart loPUckingT down all your services that can be interacted with simultaneously.
- Anything that uses state, if accessed while running could have unintended results.

In my example printed was acting like a what?

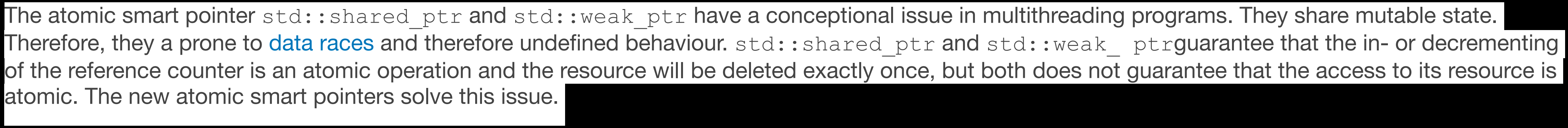
A Semaphore, sorta it wasn’t completely guarding output but it was continuing a loop of nothing until it was changed by the thread.

- You can also run into an issue when you have too many high priority threads, that cannibalize resources a lower priority thread might need.
- Even if the lower priority thread gets processed the resource they share might still be tied up waiting for the higher priority thread to releases it.

- So multithreading programming in c++ is still undergoing a lot of catching up to other programming languages.

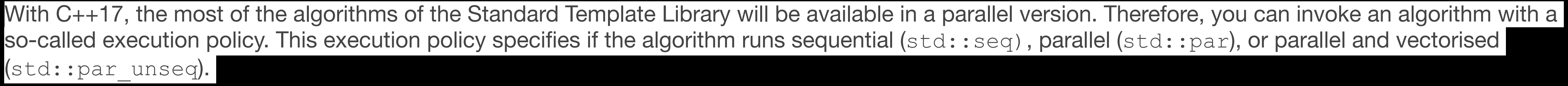
- in c++ 20 you'll have access to an Atomic Smart Pointer, fixing threadsafety when using a shared pointer.

Atomic smart pointer



The atomic smart pointer `std::shared_ptr` and `std::weak_ptr` have a conceptional issue in multithreading programs. They share mutable state. Therefore, they a prone to data races and therefore undefined behaviour. `std::shared_ptr` and `std::weak_ ptr`guarantee that the in- or decrementing of the reference counter is an atomic operation and the resource will be deleted exactly once, but both does not guarantee that the access to its resource is atomic. The new atomic smart pointers solve this issue.

- Most recently in c++ 17 they added parallel algorithms.



With C++17, the most of the algorithms of the Standard Template Library will be available in a parallel version. Therefore, you can invoke an algorithm with a so-called execution policy. This execution policy specifies if the algorithm runs sequential (`std::seq`) , parallel (`std::par`), or parallel and vectorised (`std::par_unseq`).