Rust Mastery Saga
Unlocking Backend Power with Rust

Dancing with My Code

Introduction to Rust

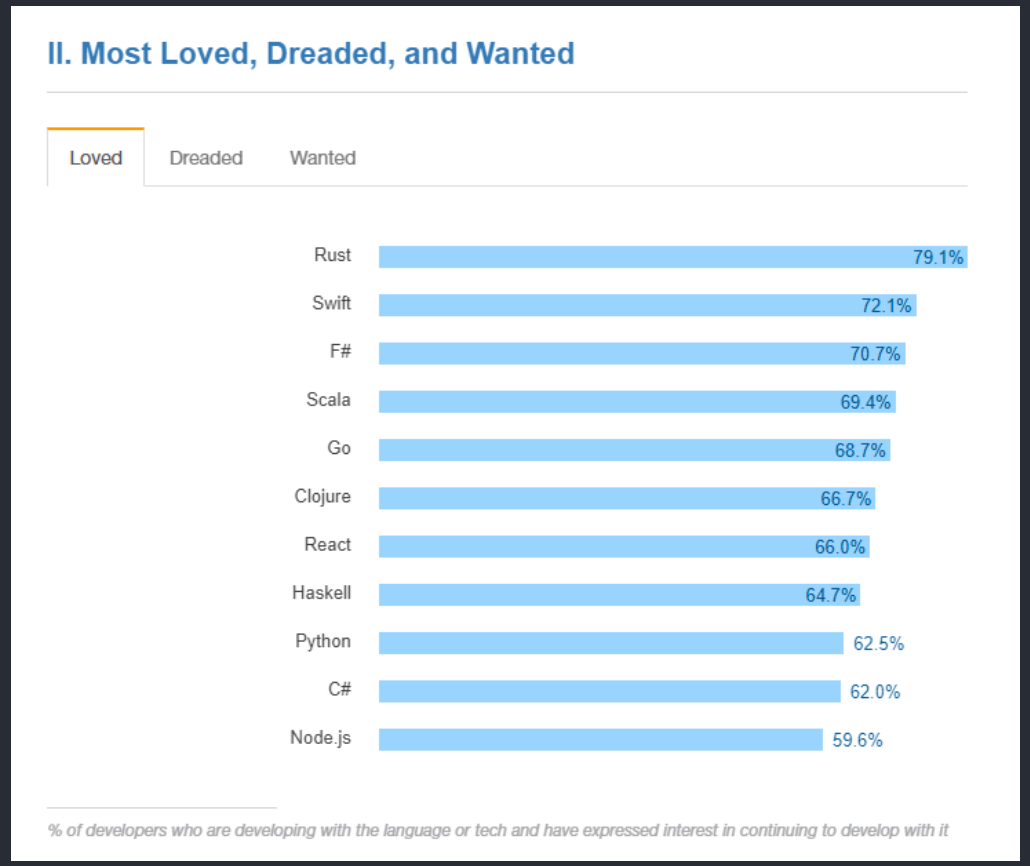
Why Rust ???

From a side project

to the world's most-loved programming language

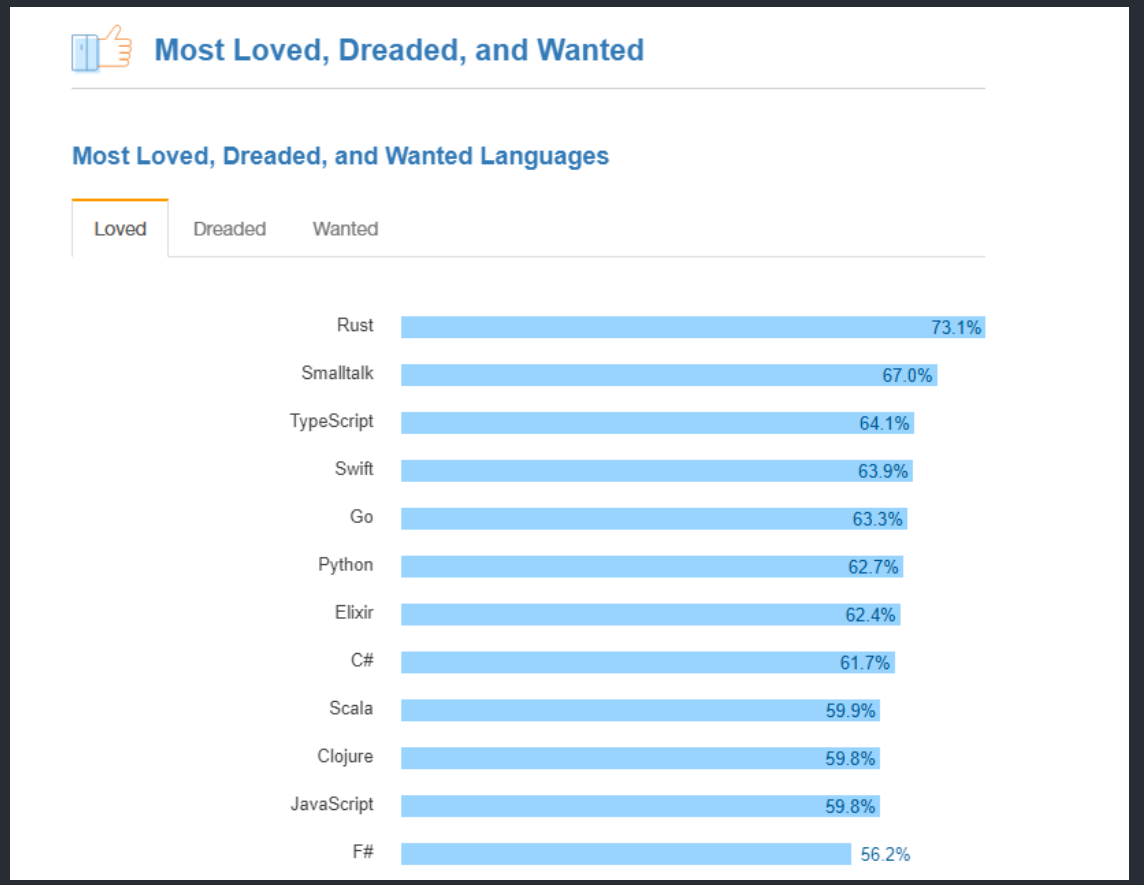
Let's review the Stackoverflow surveys.

2016



https://survey.stackoverflow.co/2016#technology-most-loved-dreaded-and-wanted

2017



https://survey.stackoverflow.co/2017#most-loved-dreaded-and-wanted

2018

Most Loved, Dreaded, and Wanted

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted

Rust	78.9%
Kotlin	75.1%
Python	68.0%
TypeScript	67.0%
Go	65.6%
Swift	65.1%
JavaScript	61.9%
C#	60.4%
F#	59.6%
Clojure	59.6%
Bash/Shell	59.1%
Scala	58.5%

https://survey.stackoverflow.co/2018#most-loved-dreaded-and-wanted

2019

Most Loved, Dreaded, and Wanted

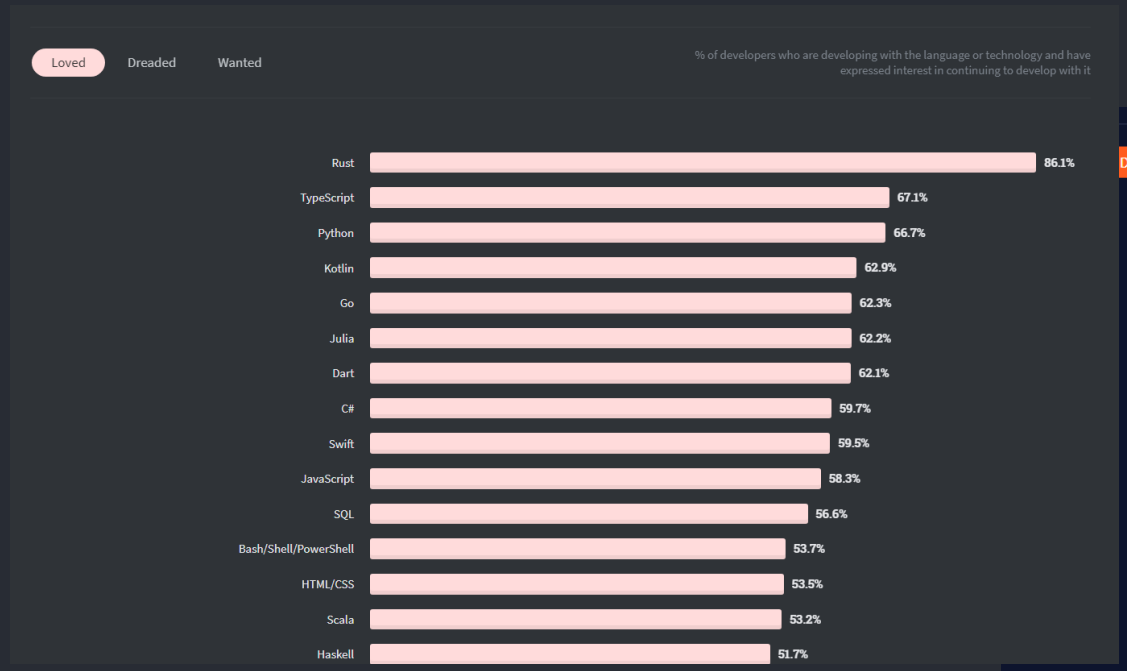
Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted

Rust	83.5%
Python	73.1%
TypeScript	73.1%
Kotlin	72.6%
WebAssembly	69.5%
Swift	69.2%
Clojure	68.3%
Elixir	68.2%
Go	67.9%
C#	67.0%
JavaScript	66.8%
Dart	66.3%
SQL	64.1%

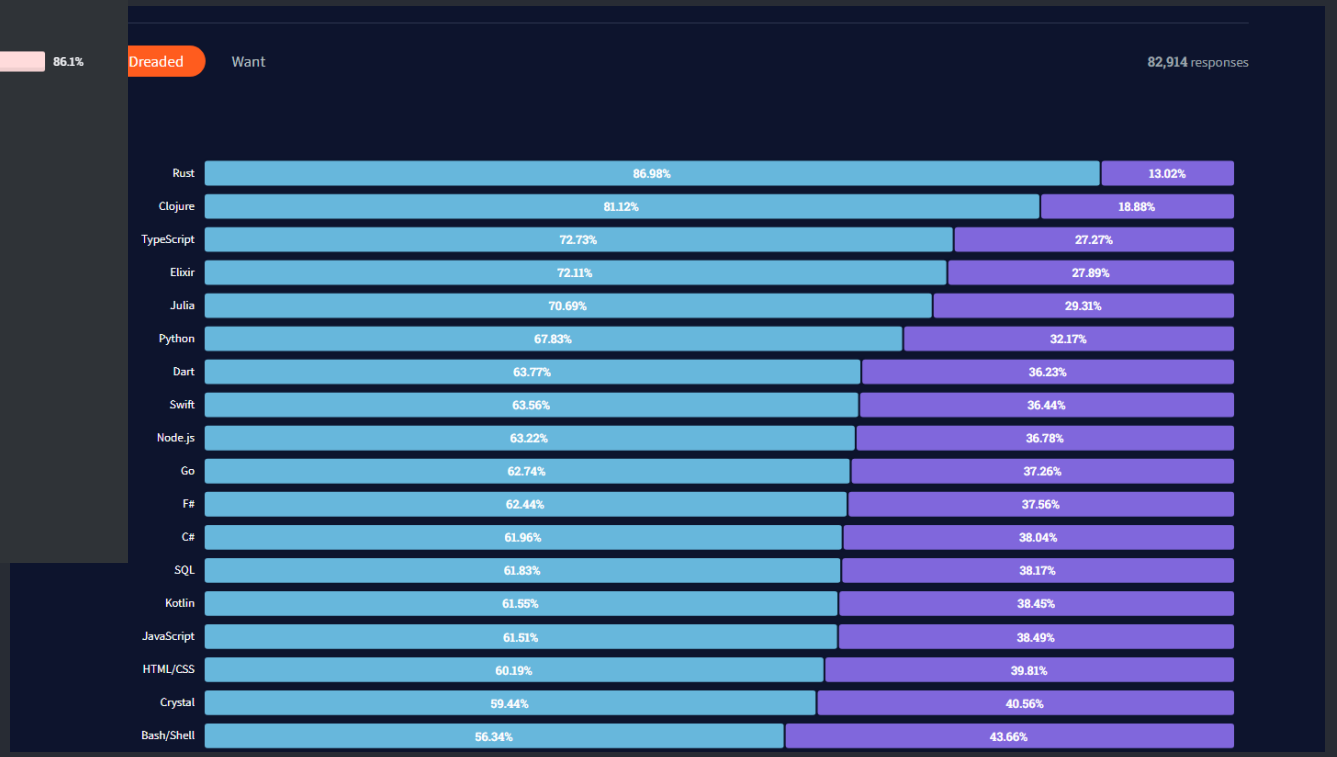
https://survey.stackoverflow.co/2019#most-loved-dreaded-and-wanted

2020



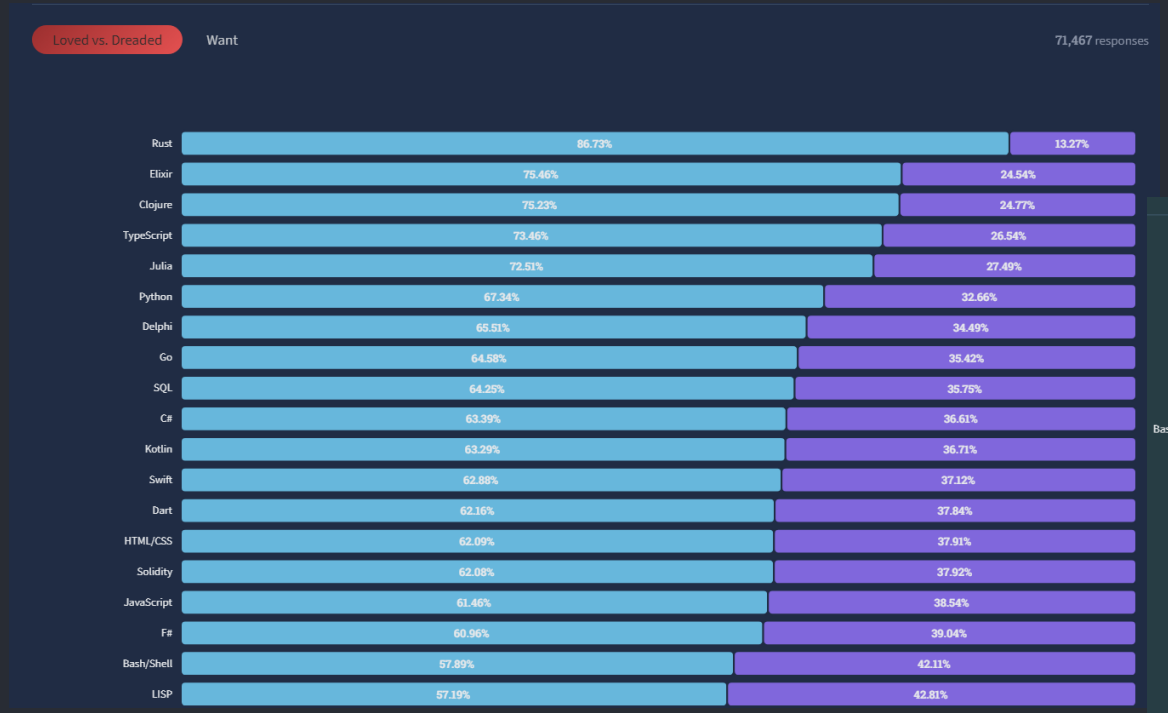
https://survey.stackoverflow.co/2020#technology-most-loved-dreaded-and-wanted-languages-loved

2021



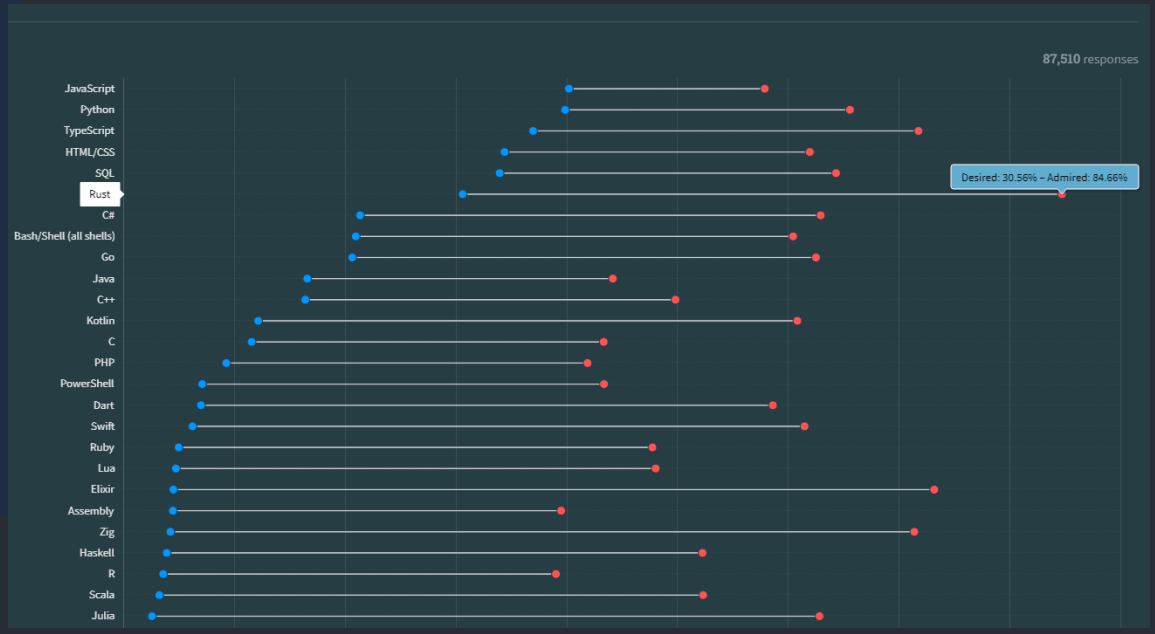
https://survey.stackoverflow.co/2021#technology-most-loved-dreaded-and-wanted

2022



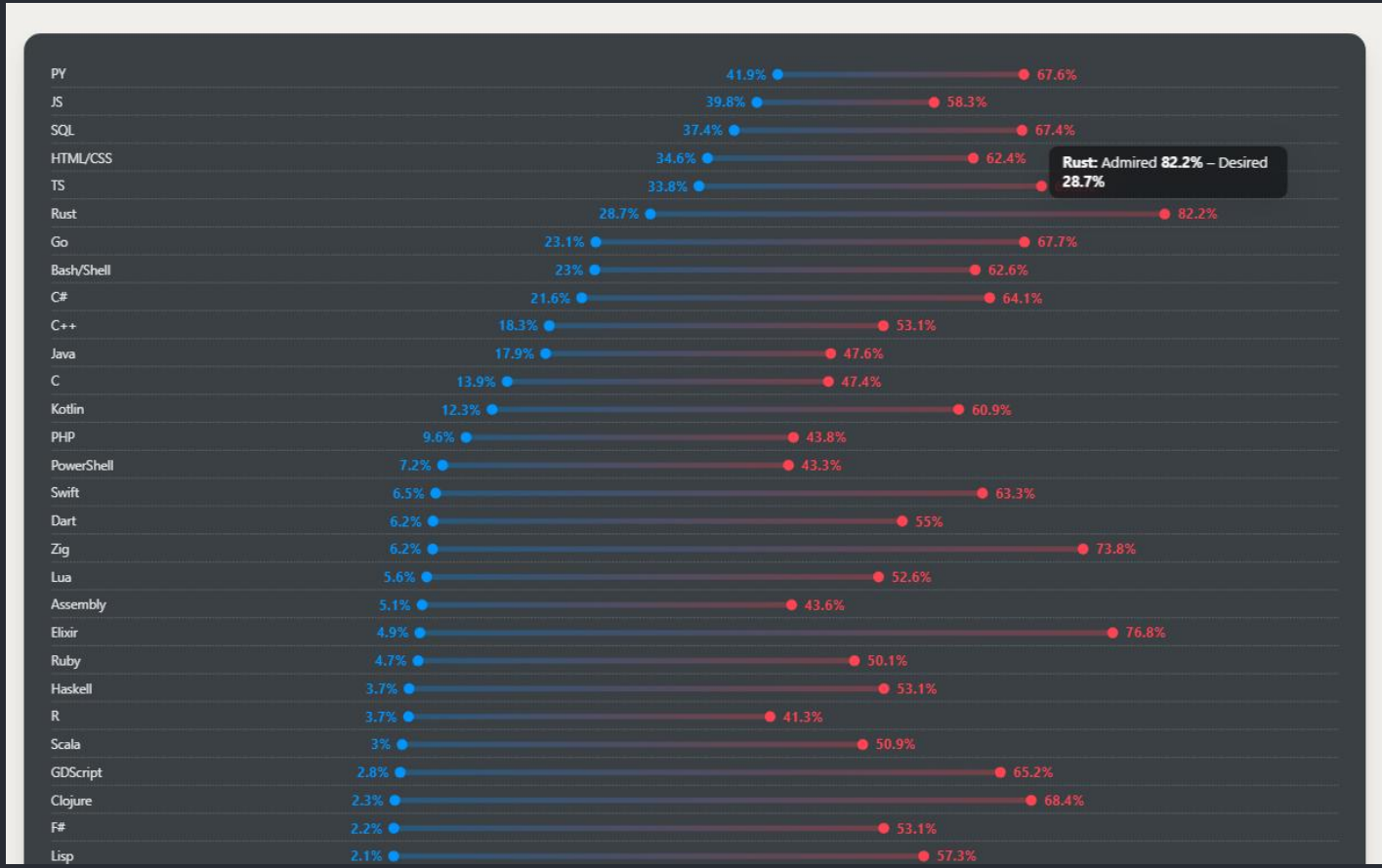
https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted

2023



https://survey.stackoverflow.co/2023/#technology-admired-and-desired

2024



https://survey.stackoverflow.co/2024/technology#admired-and-desired

9-Years Winning Streak

Let's take a deep dive into Rust's history.

2006

Graydon Hoare begins developing Rust as a personal project while working at Mozilla, because he found that the elevator was out of order.

He named it **Rust**, after a group of remarkably hardy fungi that are, he says,

“over-engineered for survival.”

2009

Mozilla decided to officially sponsor Rust. The language would be open source, and accountable only to the people making it

2010

Mozilla engineers and Rust volunteers worldwide gradually honed Rust’s core to manage memory. They created an “**ownership**” system so that a piece of data can be referred to by only one variable

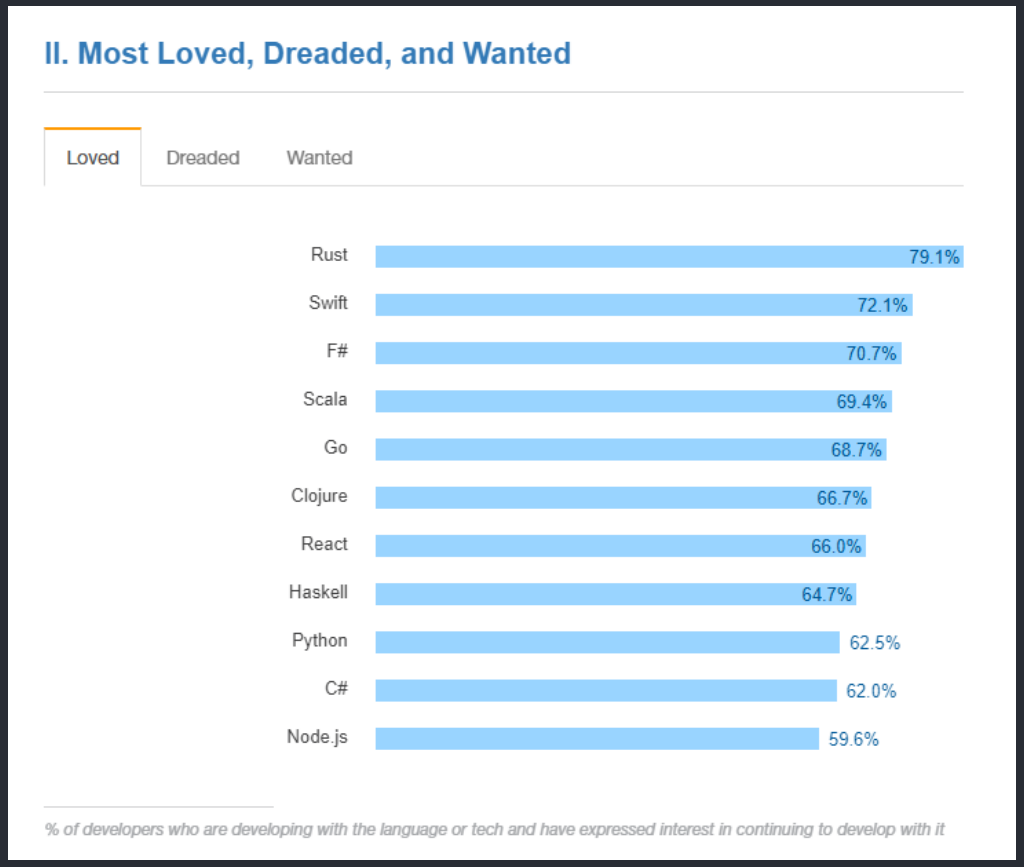
2013

As the team improved the memory-management system, **Rust** had increasingly little need for its own garbage collector—and **by 2013, the team had removed it.**

2015

The team was obsessed with finally **releasing a “stable” version of Rust**, one reliable enough for companies to use to make software for real customers.

2016



https://survey.stackoverflow.co/2016#technology-most-loved-dreaded-and-wanted

Full history of Rust:

https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/

Hello, crustaceans.

Meet **Ferris**, the unofficial mascot for Rust!

https://rustacean.net/

So, which industries use Rust?

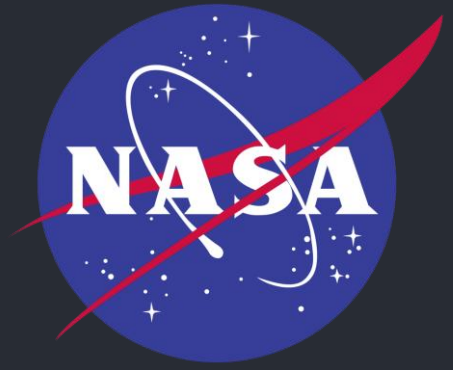
- Technology 👩‍💻
- Cloud Computing & Infrastructure ☁️
- Fintech 💵
- Cryptocurrencies 🪙
- Aerospace ✈️
- AI 🤖
- Embedded systems 🤖
- Game Development 🎮
- …



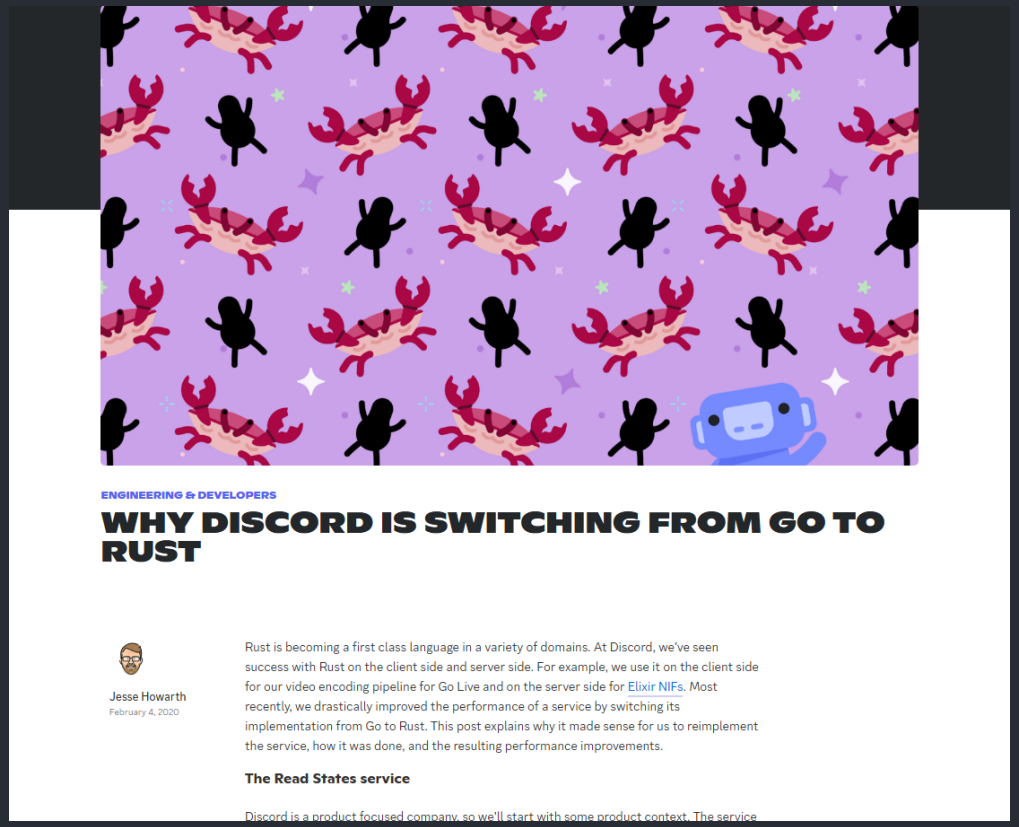



Who uses Rust ?




Example use cases of Rust.

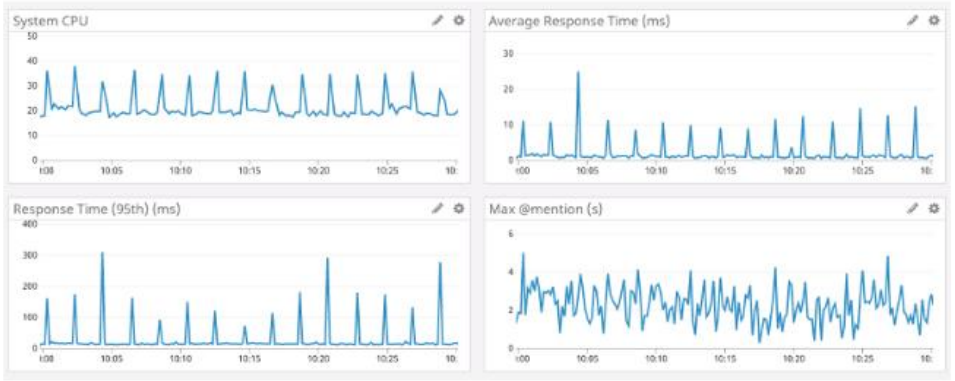


https://discord.com/blog/why-discord-is-switching-from-go-to-rust

Discord

In early 2020, Discord transitioned its "Read States" service from Go to Rust to enhance performance and reduce latency spikes.

In the picture below, you can see the response time and system cpu for a peak sample time frame for the Go service.[1] As you might notice, there are latency and CPU spikes roughly every 2 minutes.

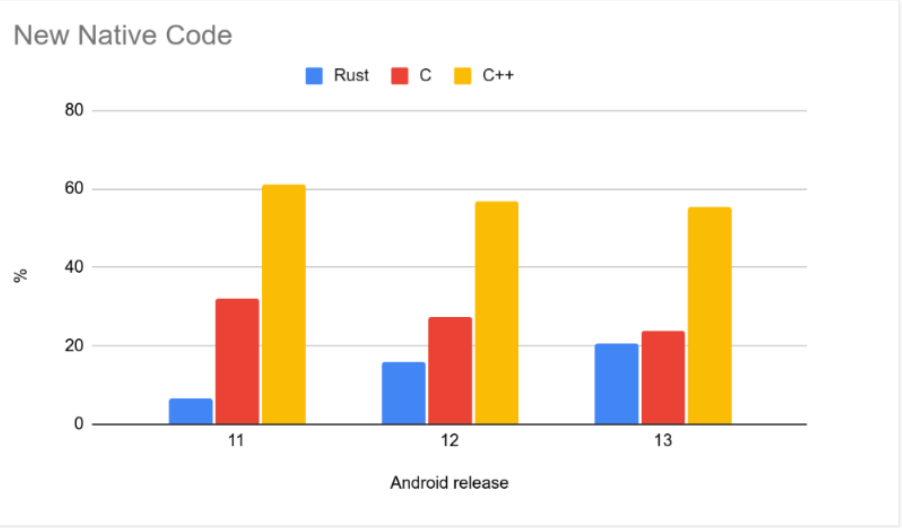


After digging through the Go source code, we learned that **Go will force a garbage collection run every 2 minutes at minimum**.

Rust for Native Code

In Android 12 we announced support for the Rust programming language in the Android platform as a memory-safe alternative to C/C++. Since then we've been scaling up our Rust experience and usage within the Android Open Source Project (AOSP).

As we noted in the original announcement, our goal is not to convert existing C/C++ to Rust, but rather to shift development of new code to memory safe languages over time.



In Android 13, about 21% of all new native code (C/C++/Rust) is in Rust. There are approximately 1.5 million total lines of Rust code in AOSP across new functionality and components such as Keystore2, the new Ultra-wideband (UWB) stack, DNS-over-HTTP3, Android's Virtualization framework (AVF), and various other components and their open source dependencies. These are low-level components that require a systems language which otherwise would have been implemented in C++.

Android 13

In Android 13, about 21% of all new native code (C/C++/Rust) is in Rust for the safety of memory purpose.

https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html?m=1

Introducing Firecracker

Today I would like to tell you about Firecracker, a new virtualization technology that makes use of KVM. You can launch lightweight micro-virtual machines (microVMs) in non-virtualized environments in a fraction of a second, taking advantage of the security and workload isolation provided by traditional VMs and the resource efficiency that comes along with containers.

Here's what you need to know about Firecracker:

Secure – This is always our top priority! Firecracker uses multiple levels of isolation and protection, and exposes a minimal attack surface.

High Performance – You can launch a microVM in as little as 125 ms today (and even faster in 2019), making it ideal for many types of workloads, including those that are transient or short-lived.

Battle-Tested – Firecracker has been battled-tested and is already powering multiple high-volume AWS services including AWS Lambda and AWS Fargate.

Low Overhead – Firecracker consumes about 5 MiB of memory per microVM. You can run thousands of secure VMs with widely varying vCPU and memory configurations on the same instance.

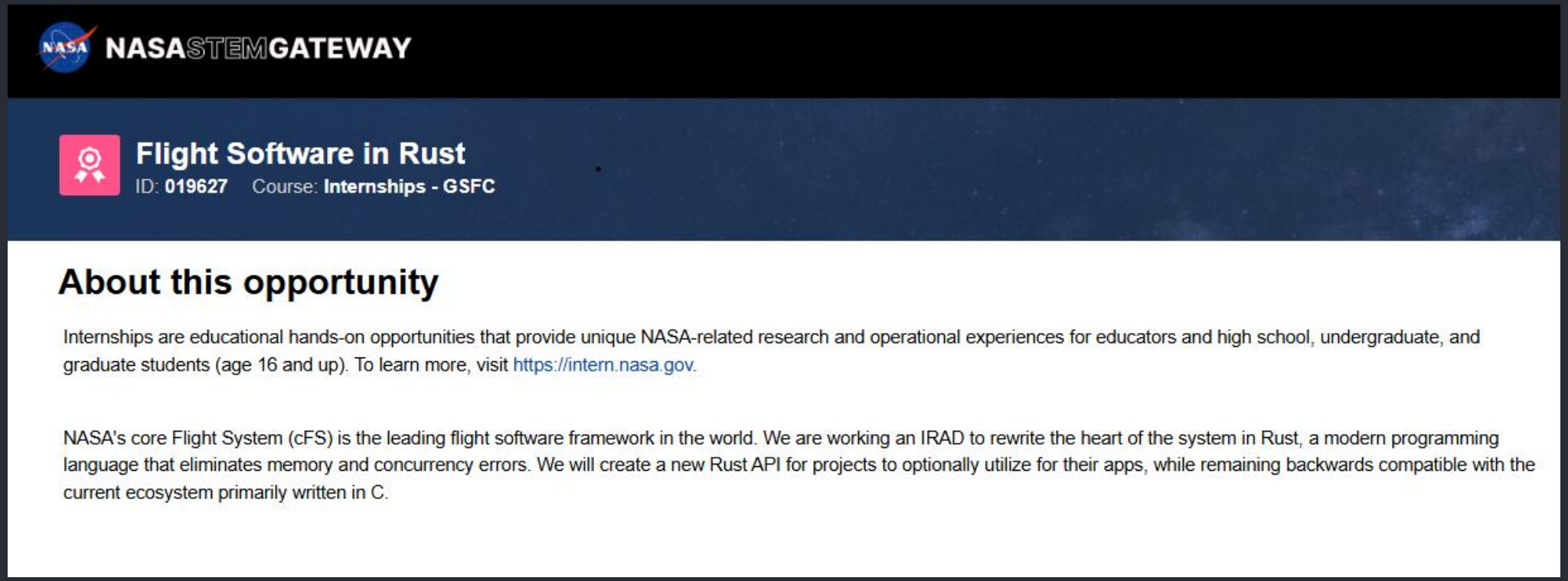
Open Source – Firecracker is an active open source project. We are already ready to review and accept pull requests, and look forward to collaborating with contributors from all over the world.

Firecracker was built in a minimalist fashion. We started with crosvm and set up a minimal device model in order to reduce overhead and to enable secure multi-tenancy. Firecracker is written in Rust, a modern programming language that guarantees thread safety and prevents many types of buffer overrun errors that can lead to security vulnerabilities.

AWS Firecracker

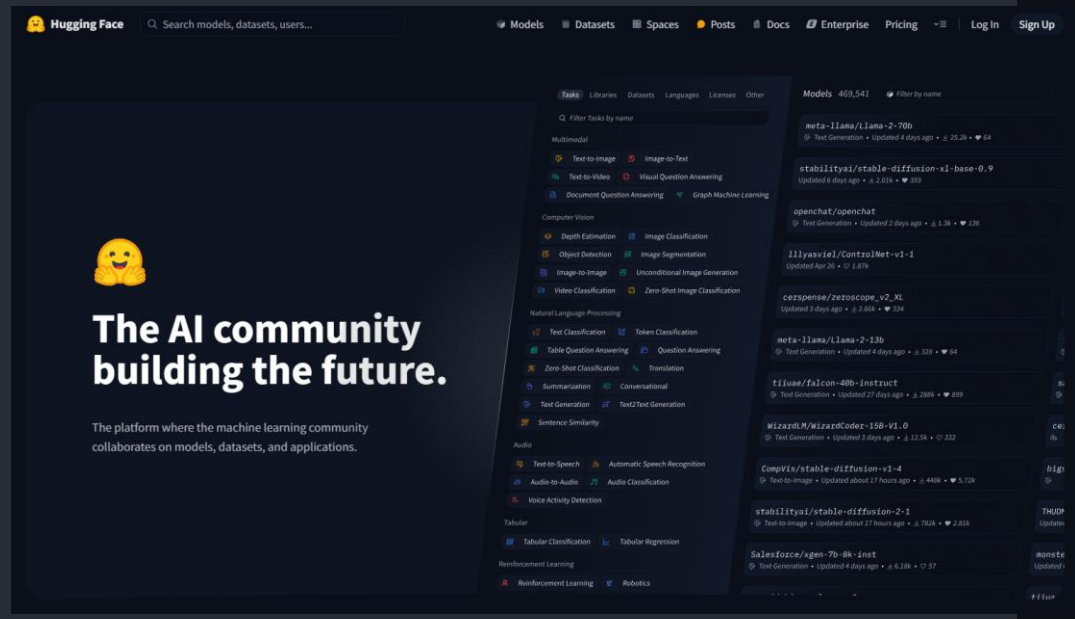
https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/

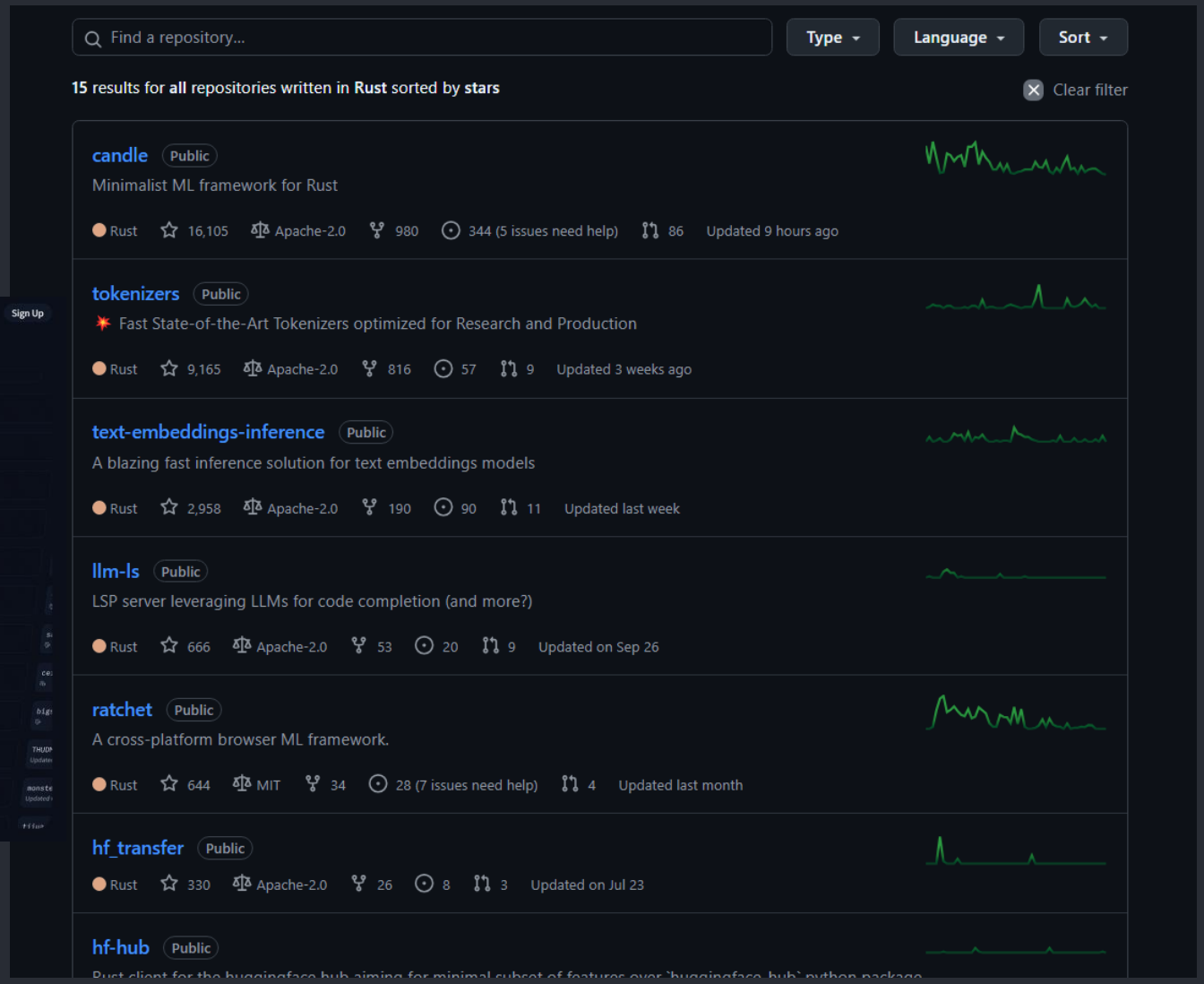
NASA



https://stemgateway.nasa.gov/public/s/course-offering/a0BSJ000000KS9p2AG/flight-software-in-rust

Hugging Face

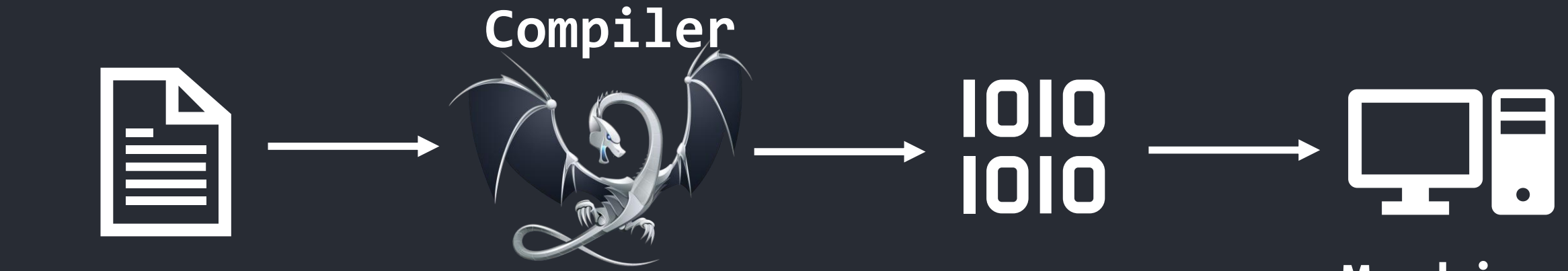




https://github.com/huggingface?q=&type=all&language=rust&sort=stargazers

It’s time to go deeper in Rust.

Low-Level (Rust)

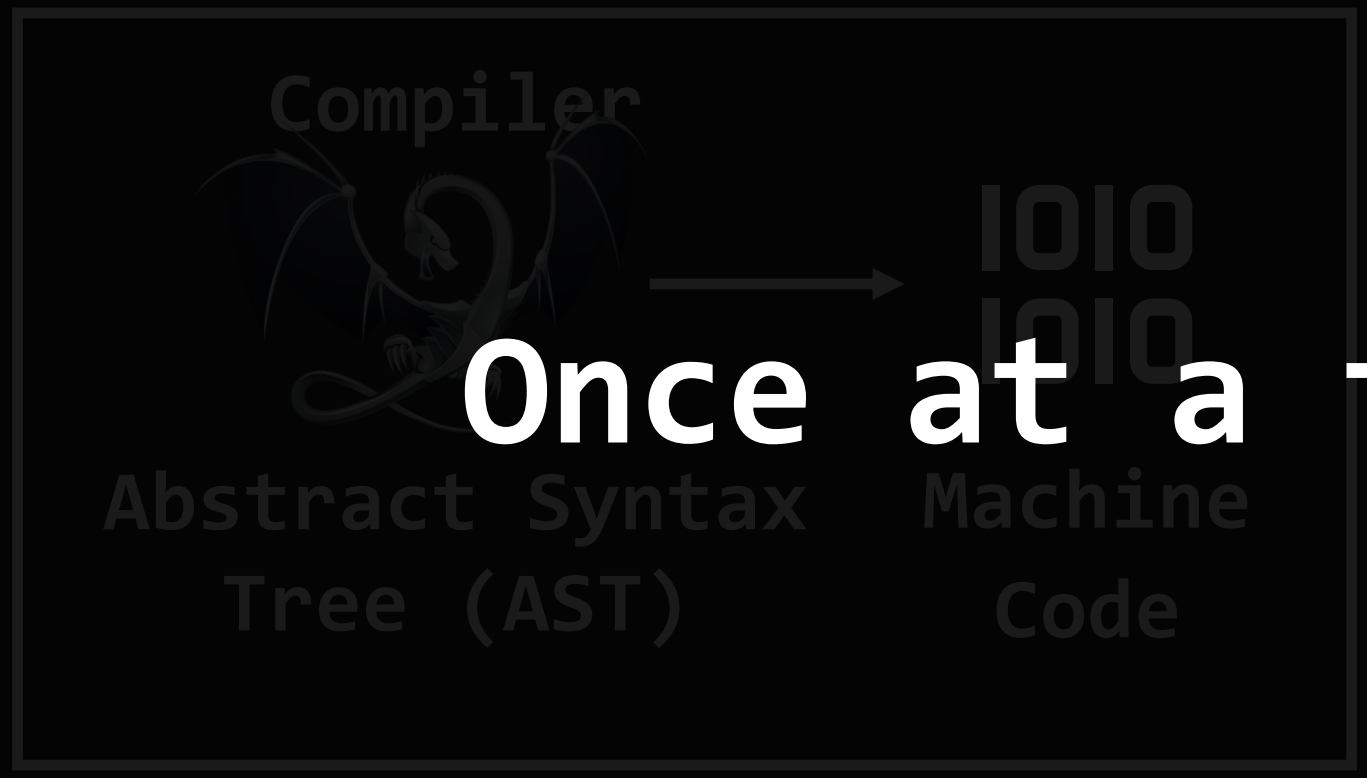


Compiler

Abstract Syntax Tree (AST)

Machine Code

Executable File

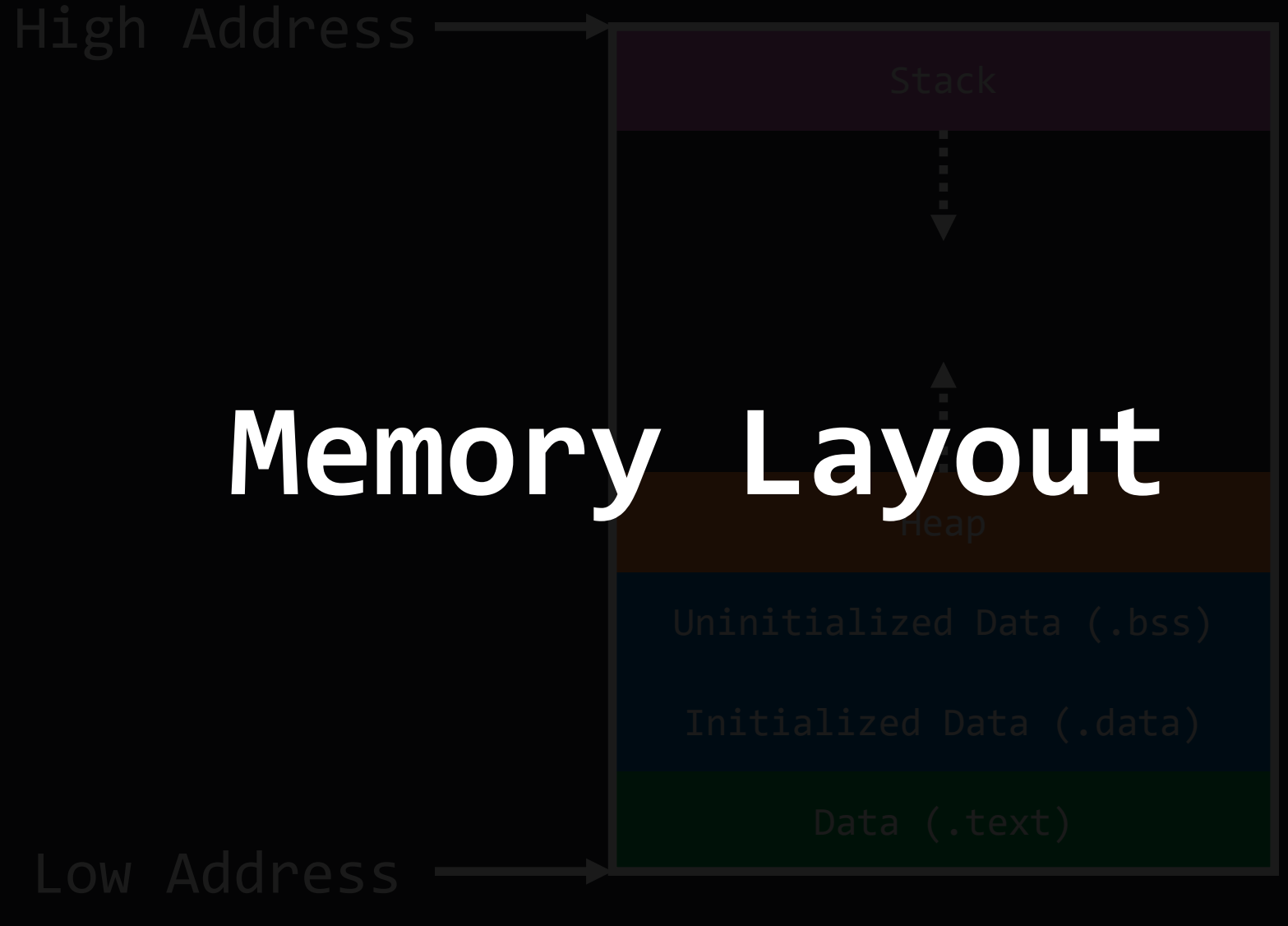


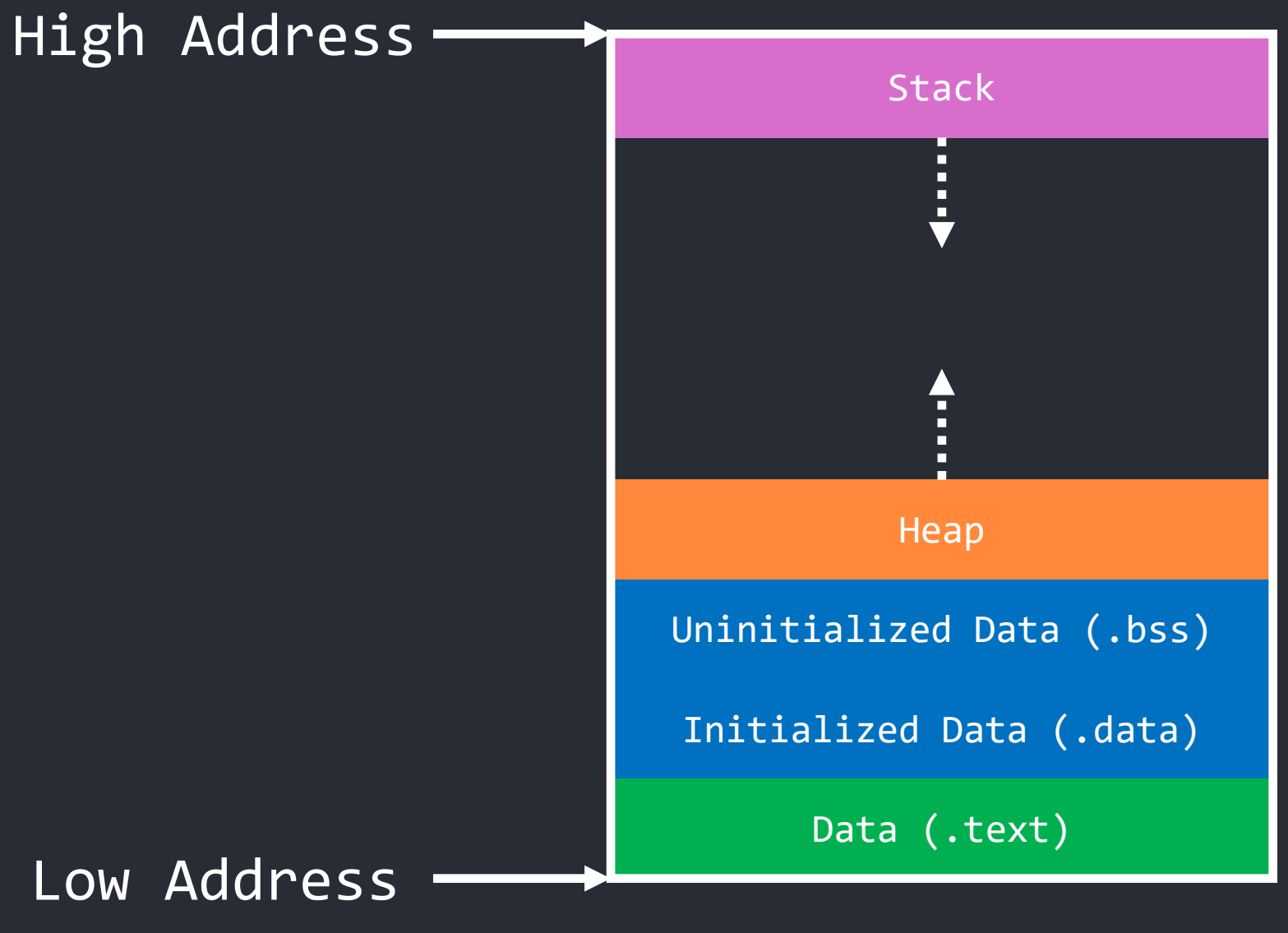
Once at a time


Why Rust is
memory safe ?

Let’s see, how **ownership** works.

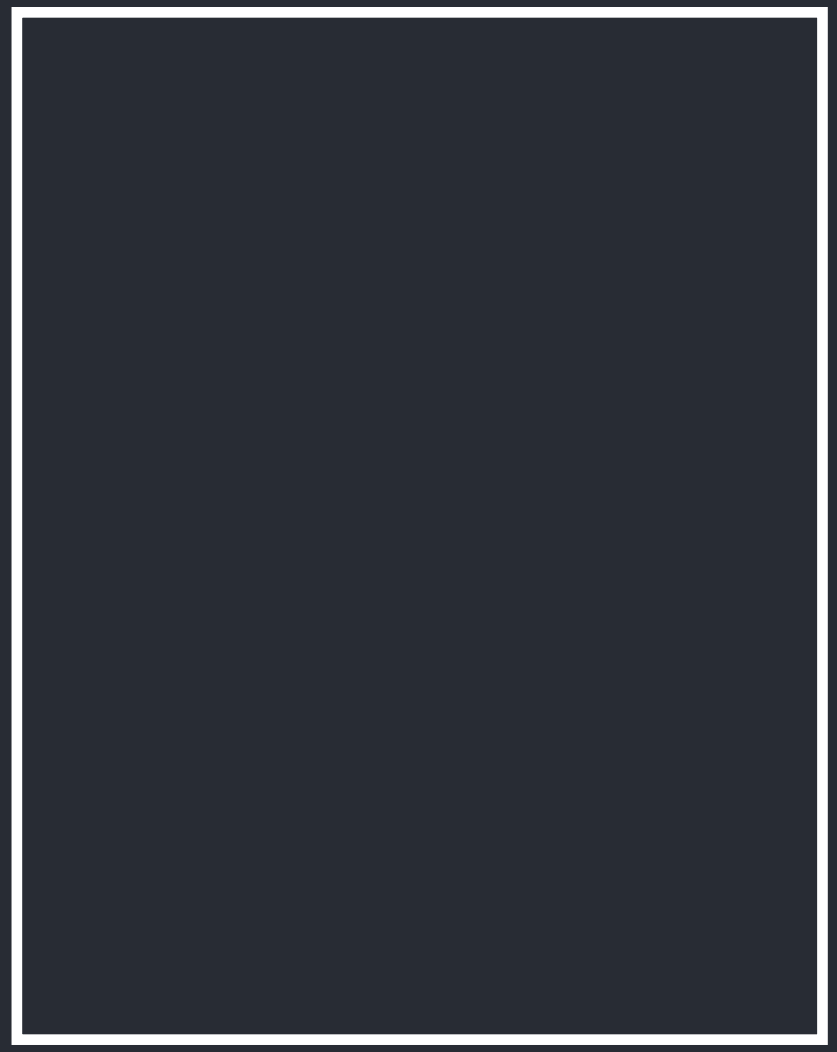
Memory Layout



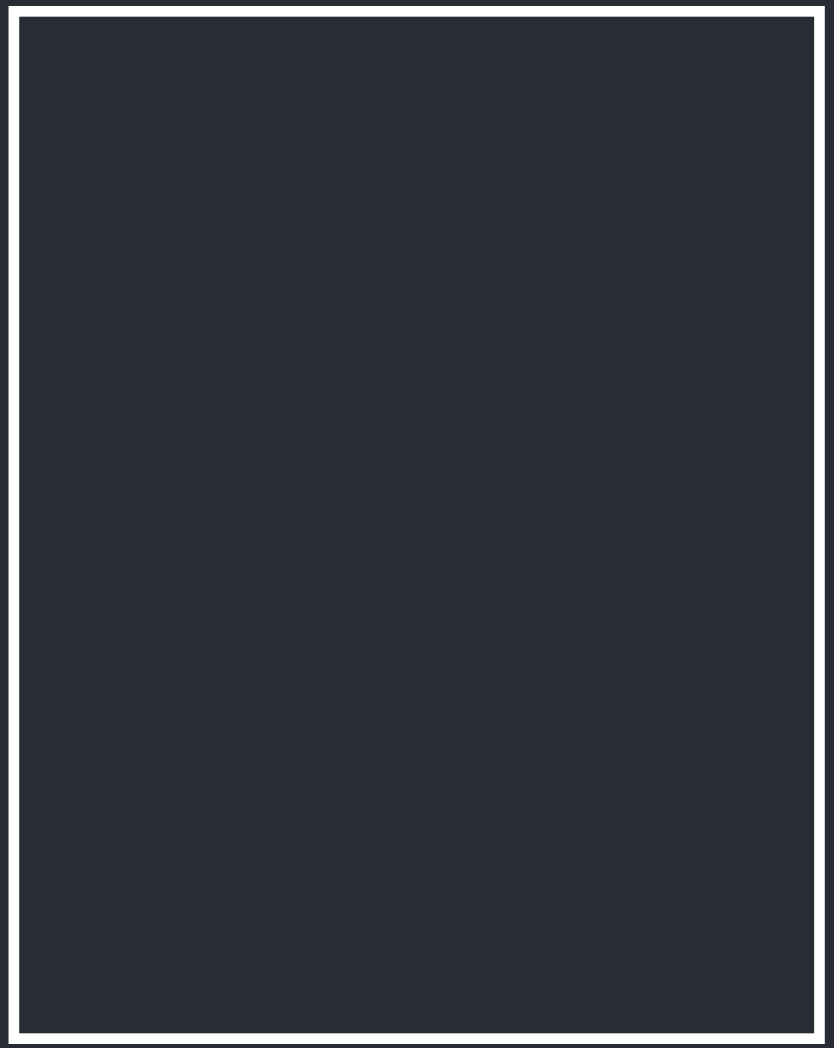

High Address
Stack
Heap
Uninitialized Data (.bss)
Initialized Data (.data)
Data (.text)
Low Address

Let’s see, how **stack** works in **C**.

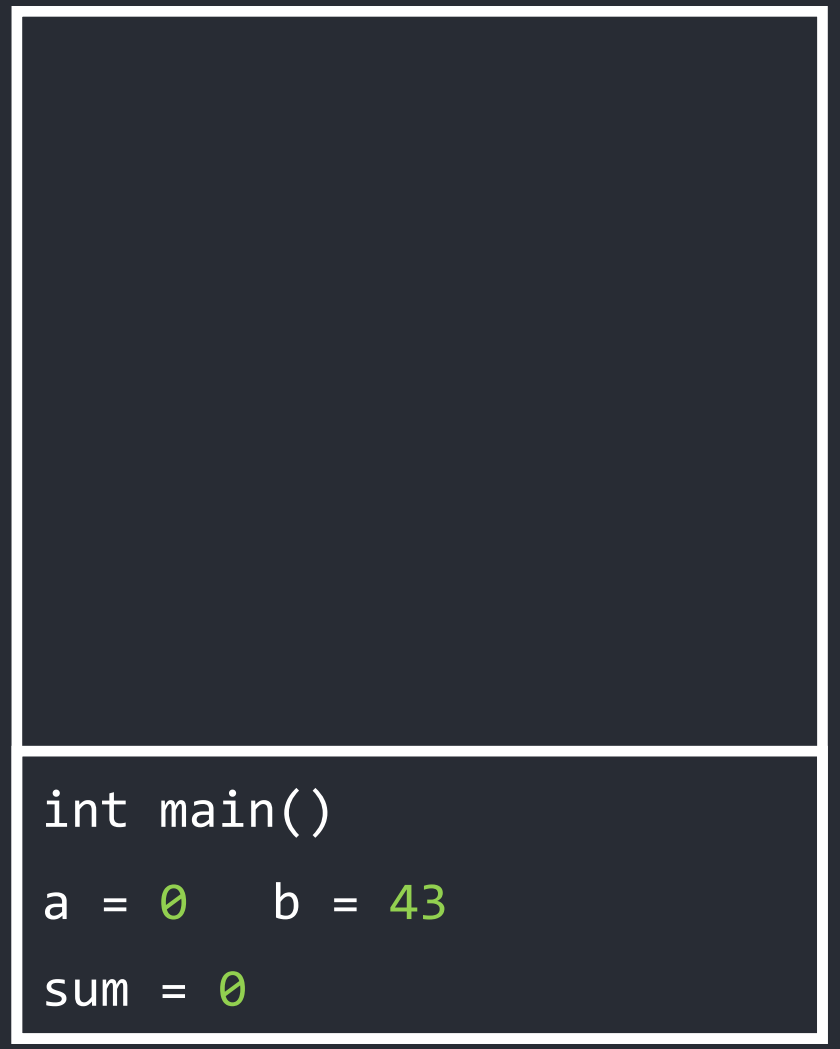
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(g(a), g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



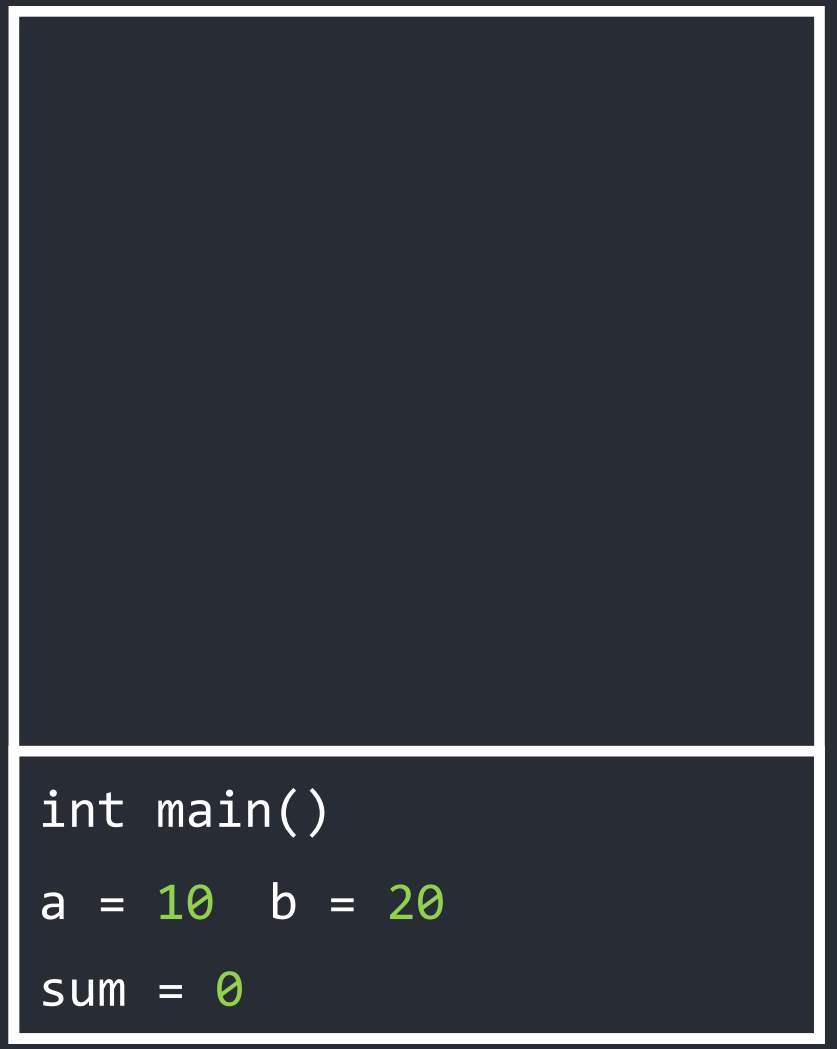
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(g(a), g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



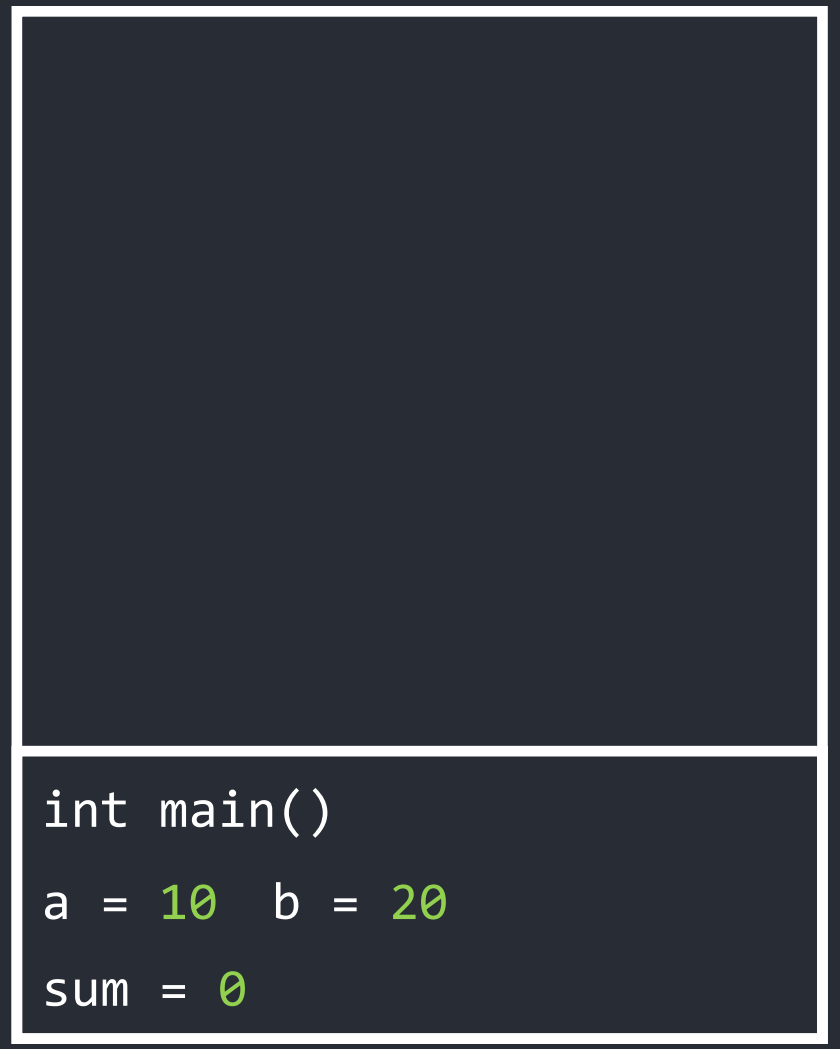
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(g(a), g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



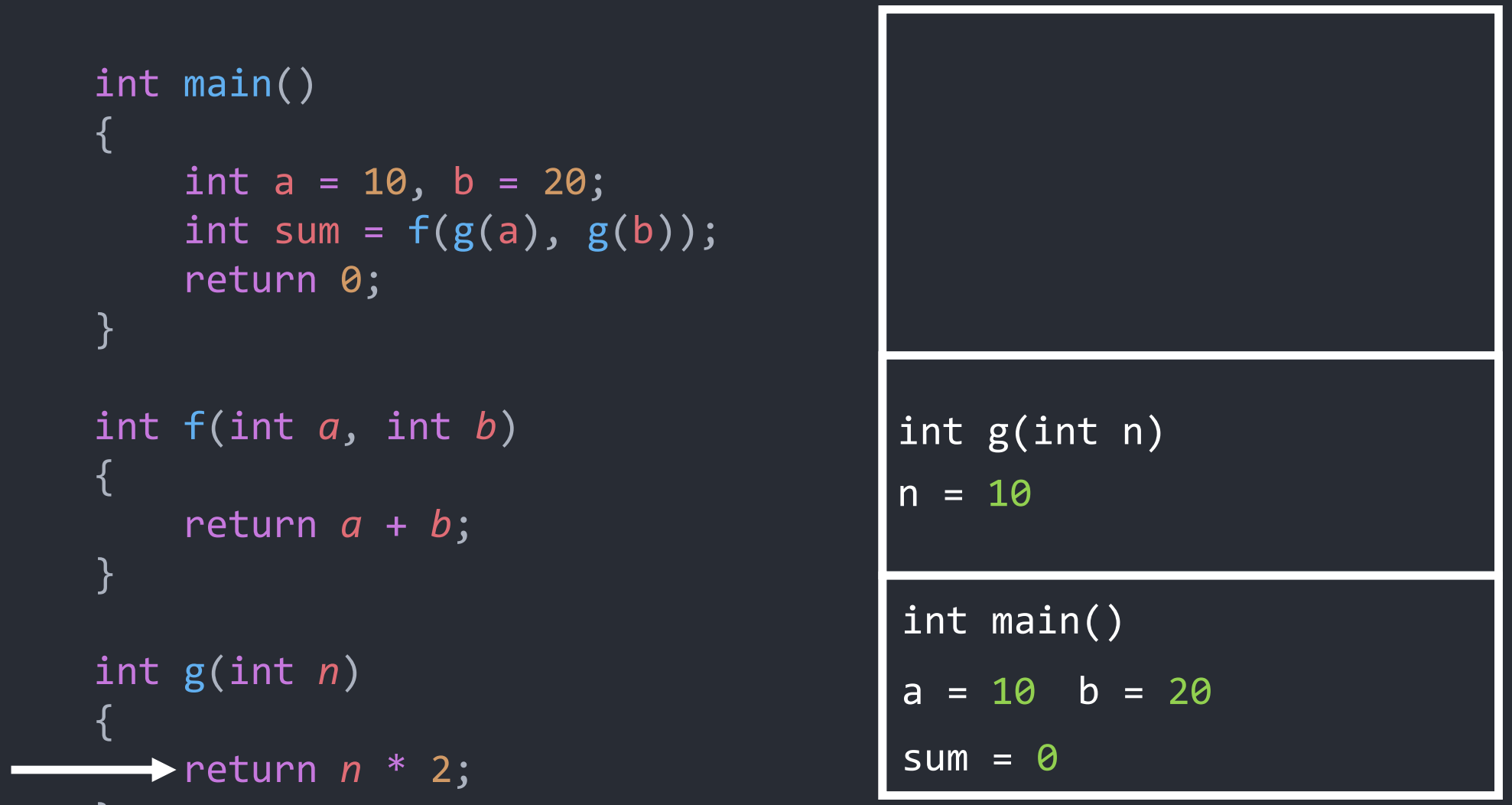
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(g(a), g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



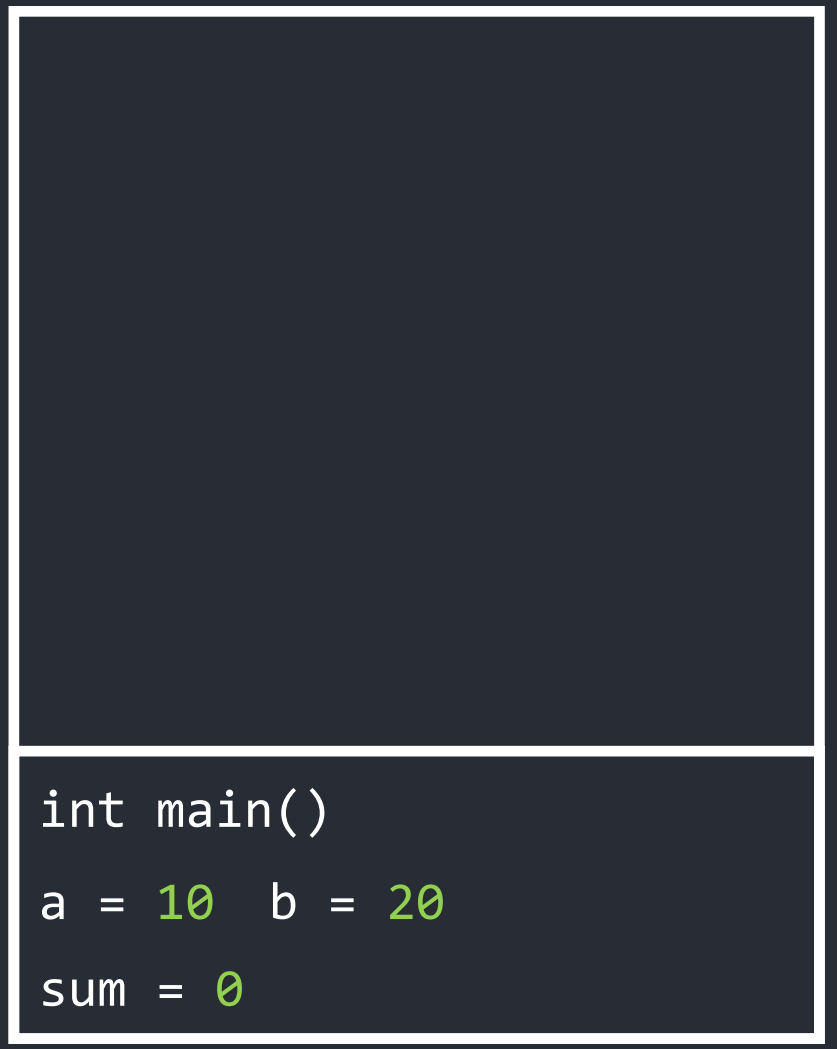
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(g(a), g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



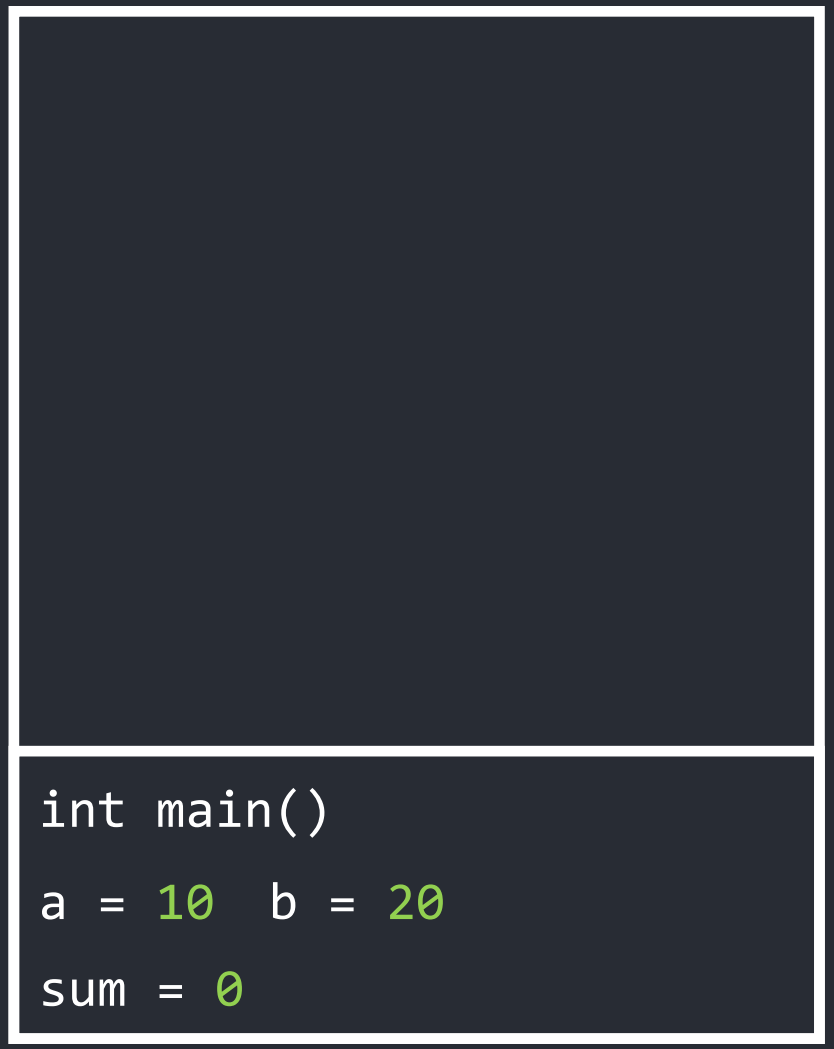
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(g(a), g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



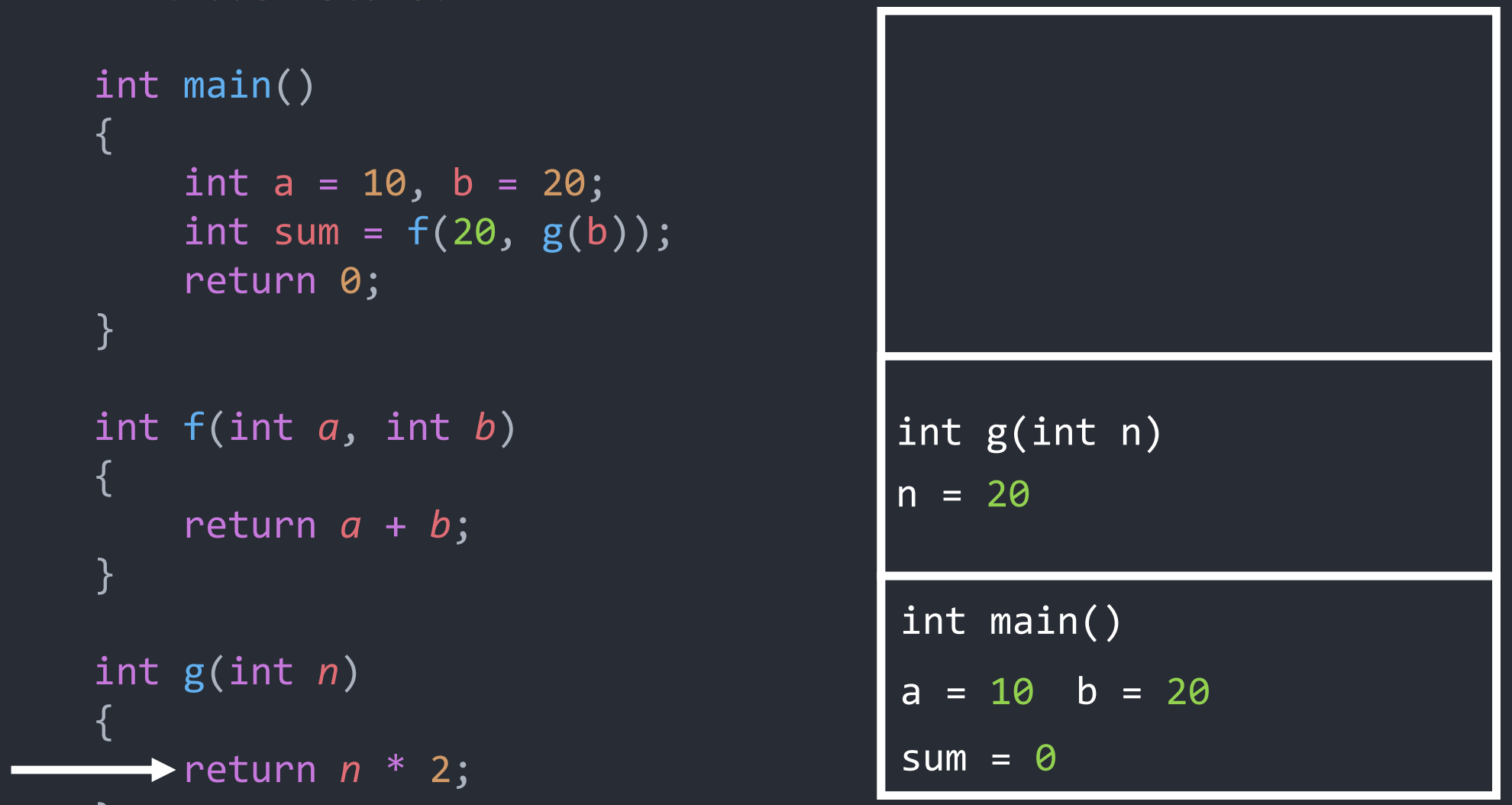
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



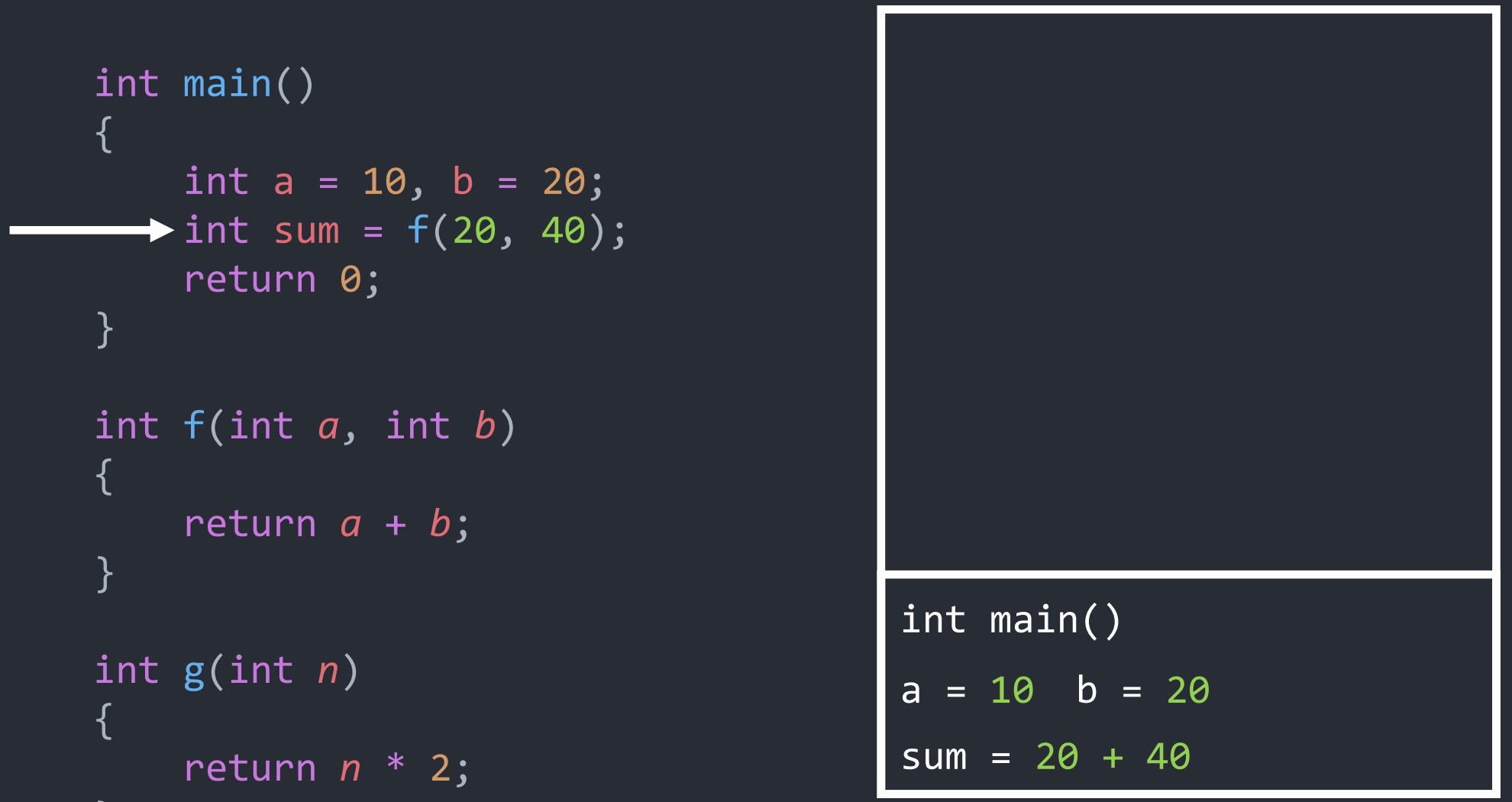
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



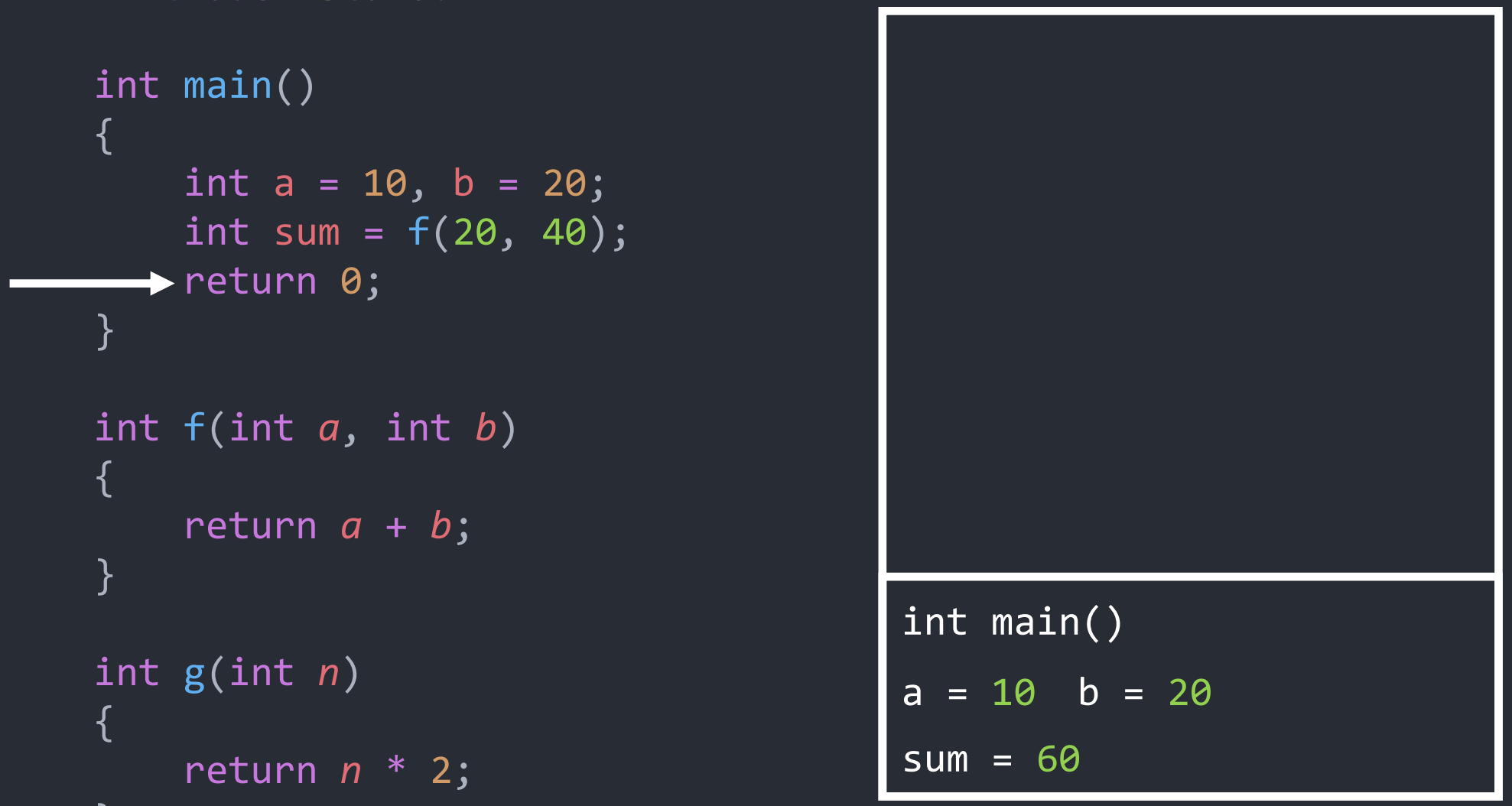
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, g(b));
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



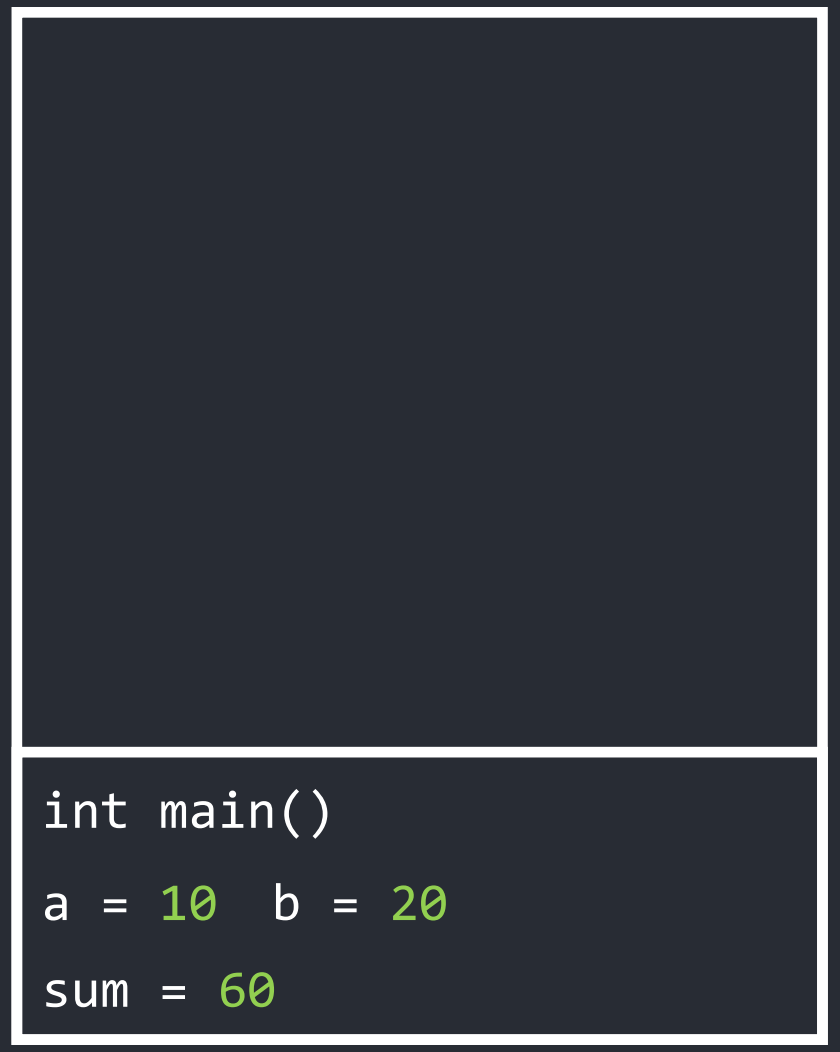
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, 40);
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



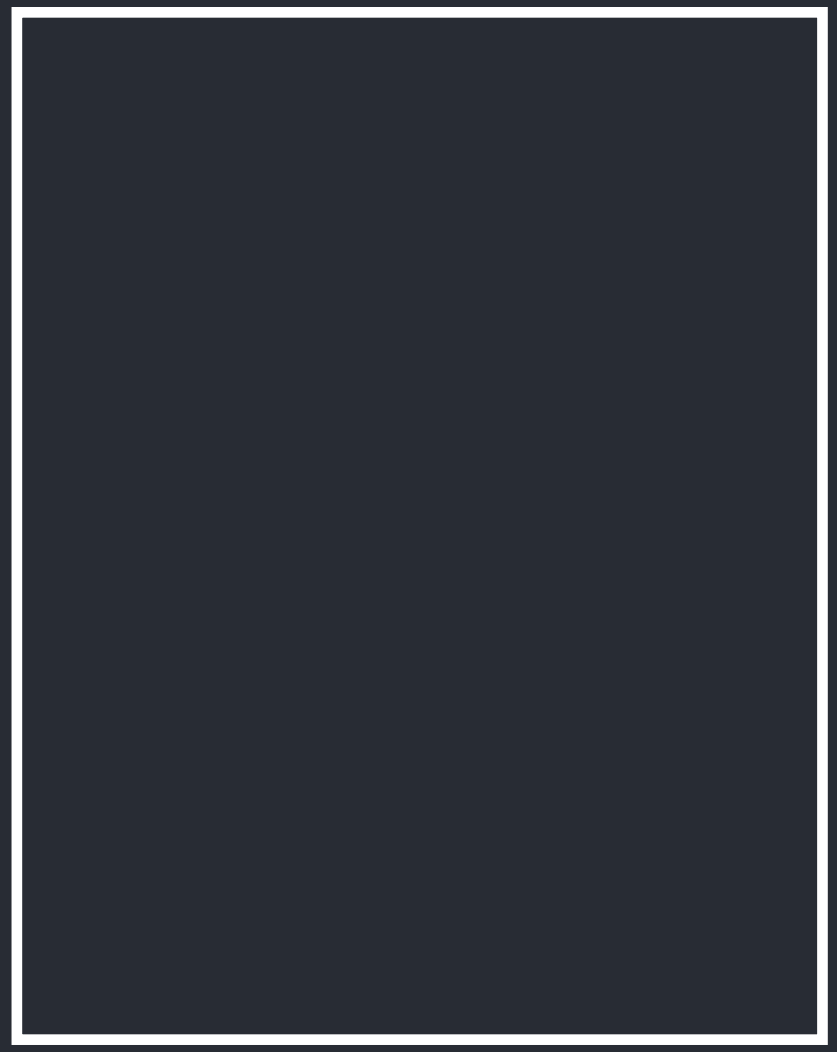
```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, 40);
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, 40);
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```

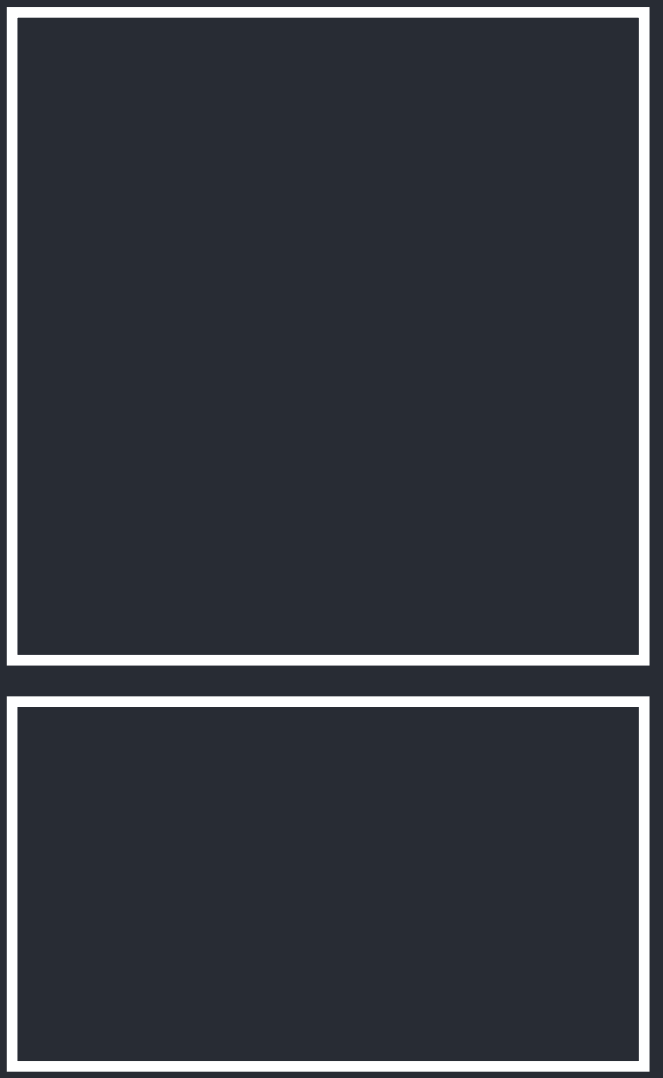


```c
#include <stdio.h>

int main()
{
    int a = 10, b = 20;
    int sum = f(20, 40);
    return 0;
}

int f(int a, int b)
{
    return a + b;
}

int g(int n)
{
    return n * 2;
}
```



Let’s see, how **heap** works in **C**.

```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```

Stack

int main()

p = (int *) 0x8

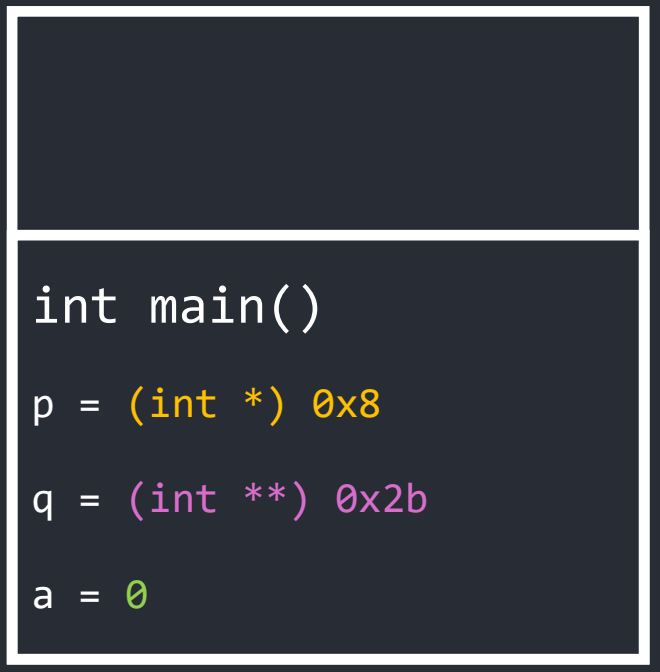
q = (int **) 0x2b

a = 0

Heap

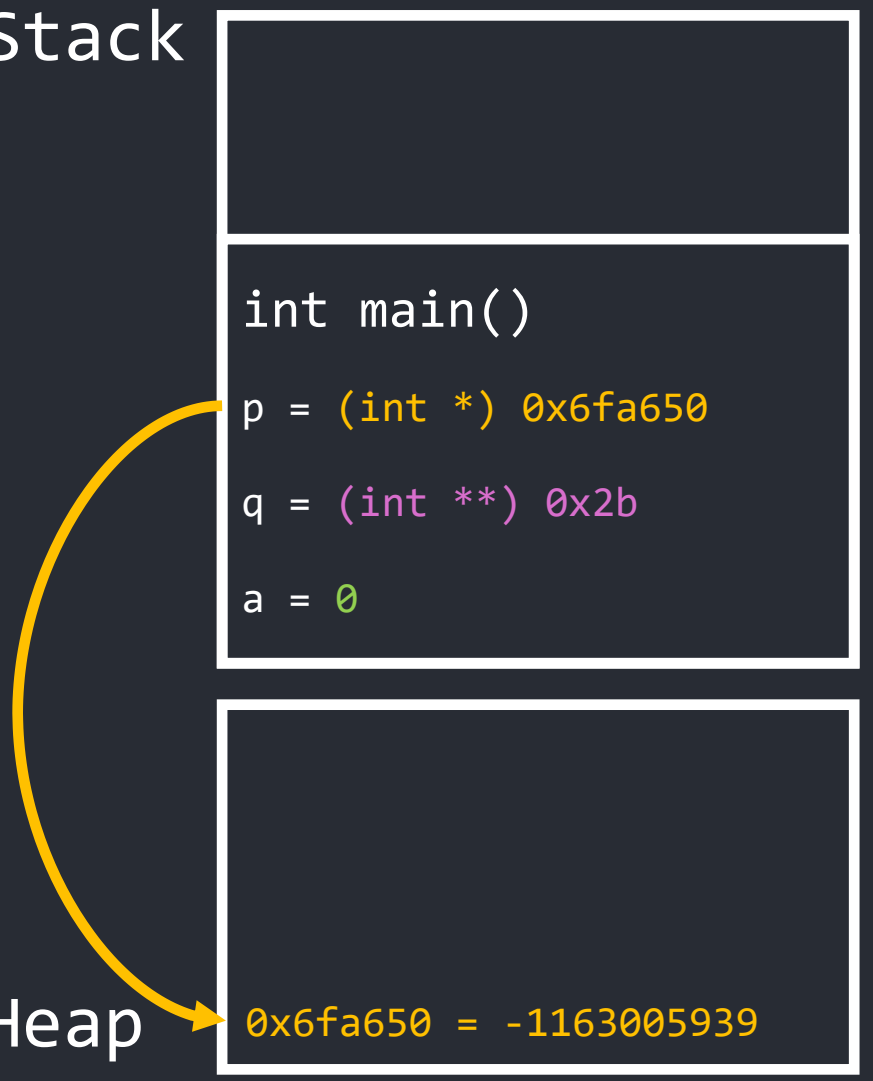
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```


Stack

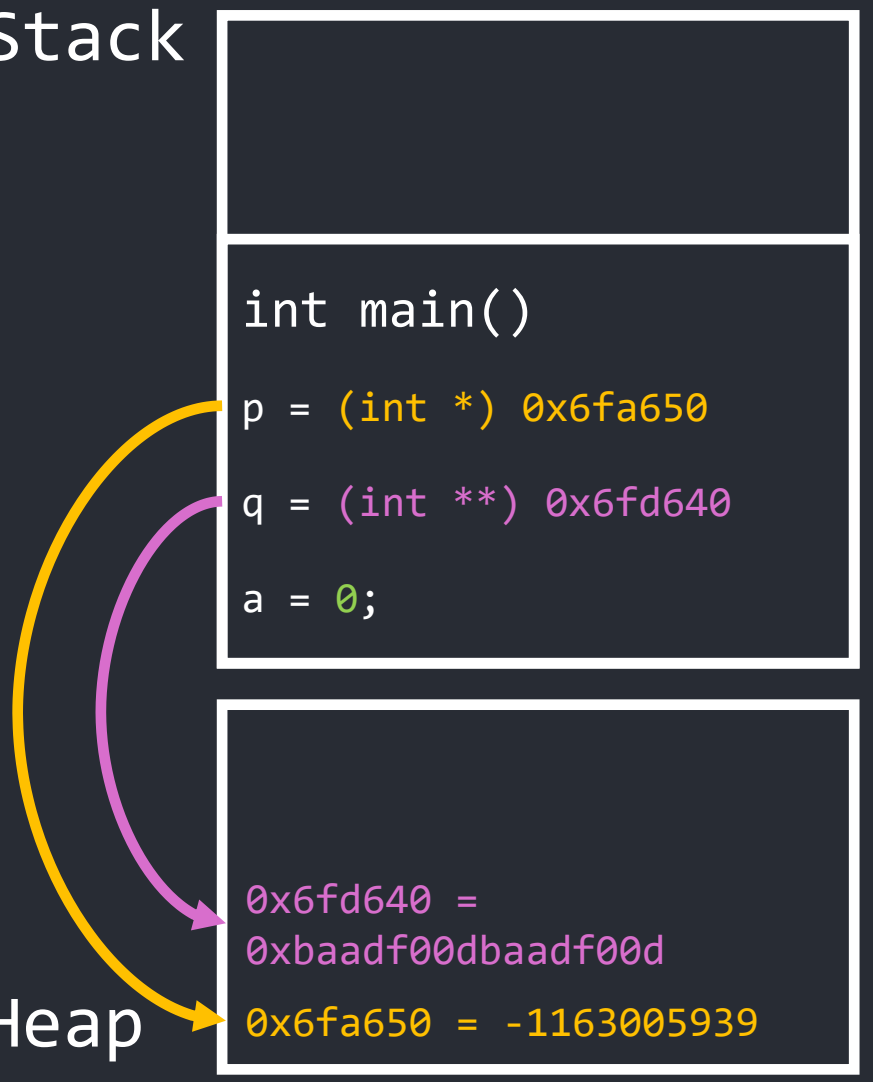


Heap

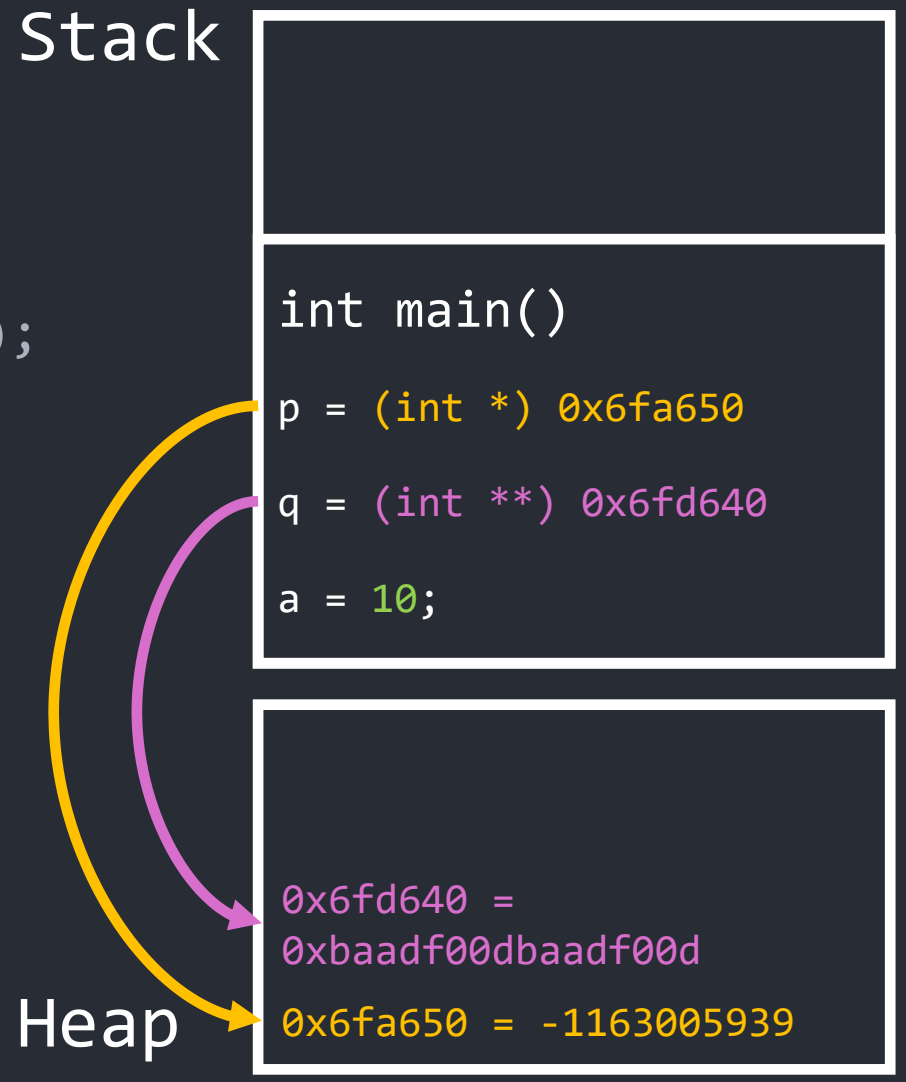
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



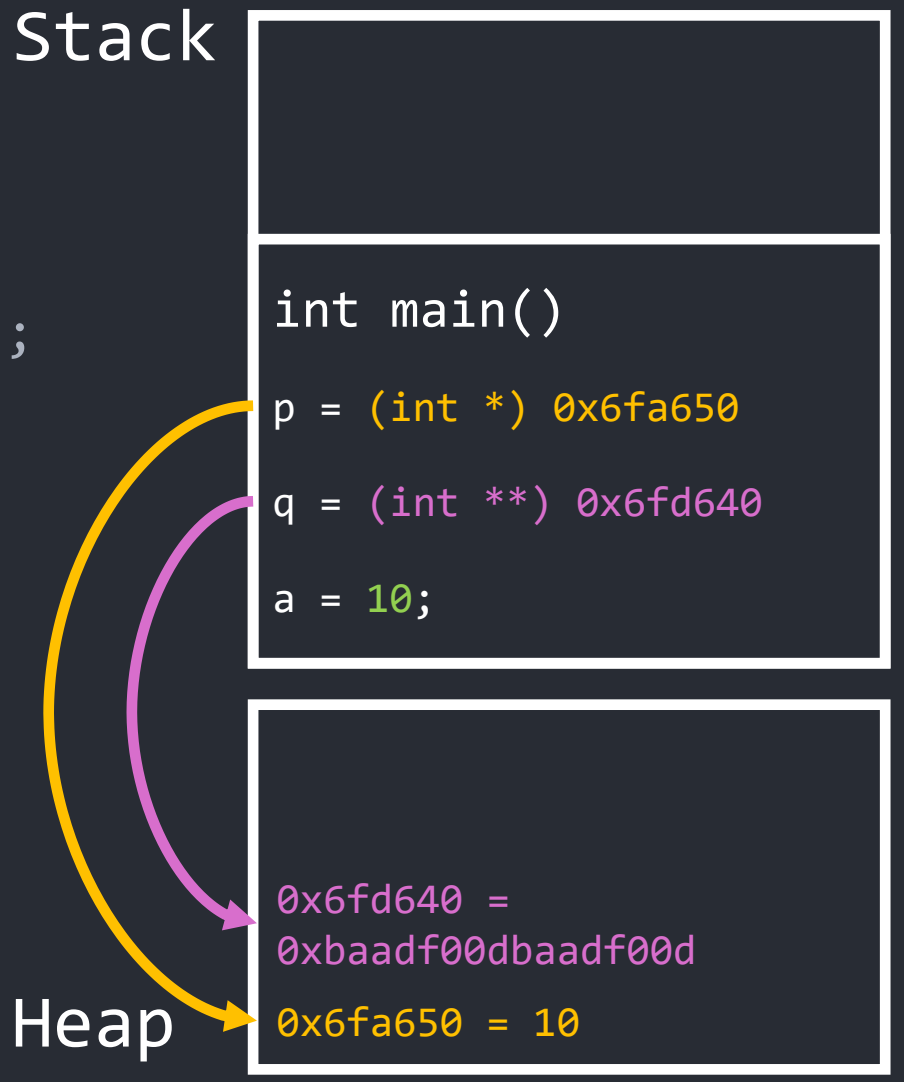
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



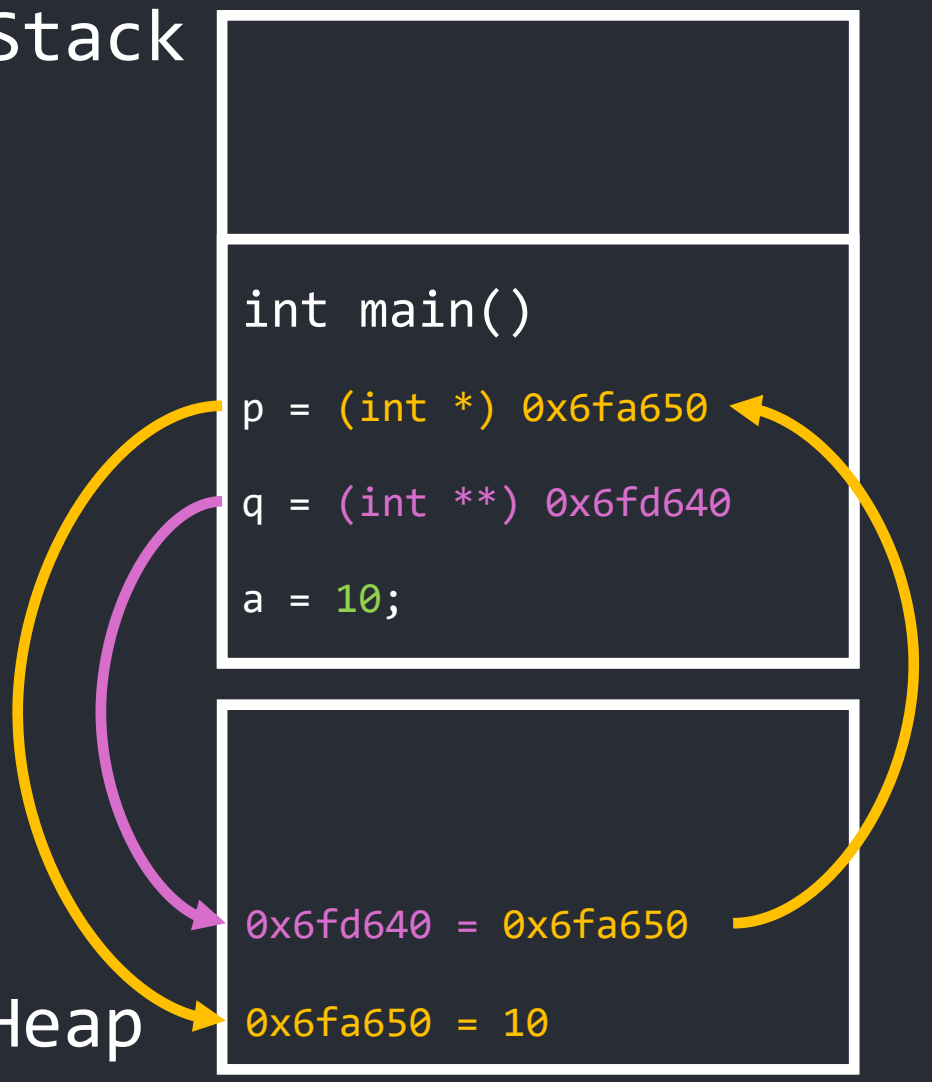
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



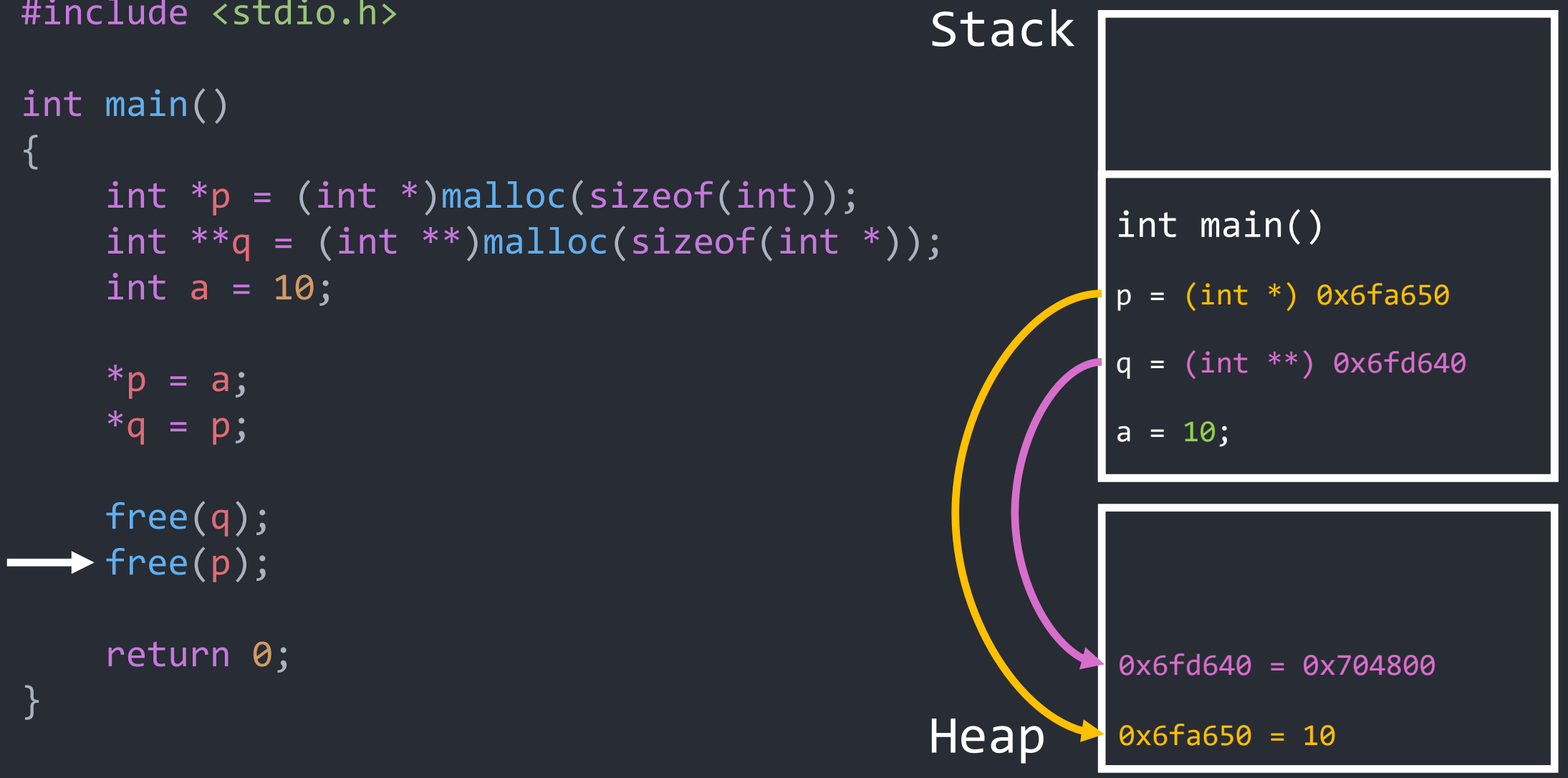
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



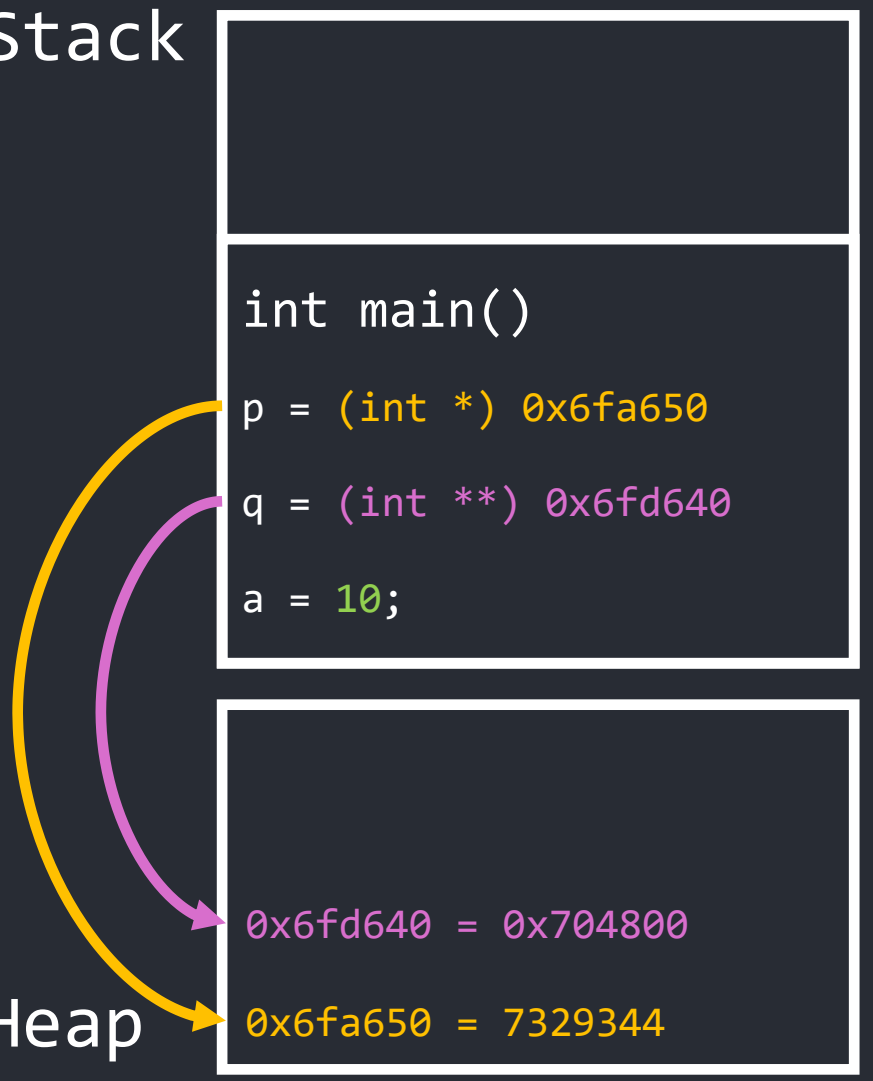
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



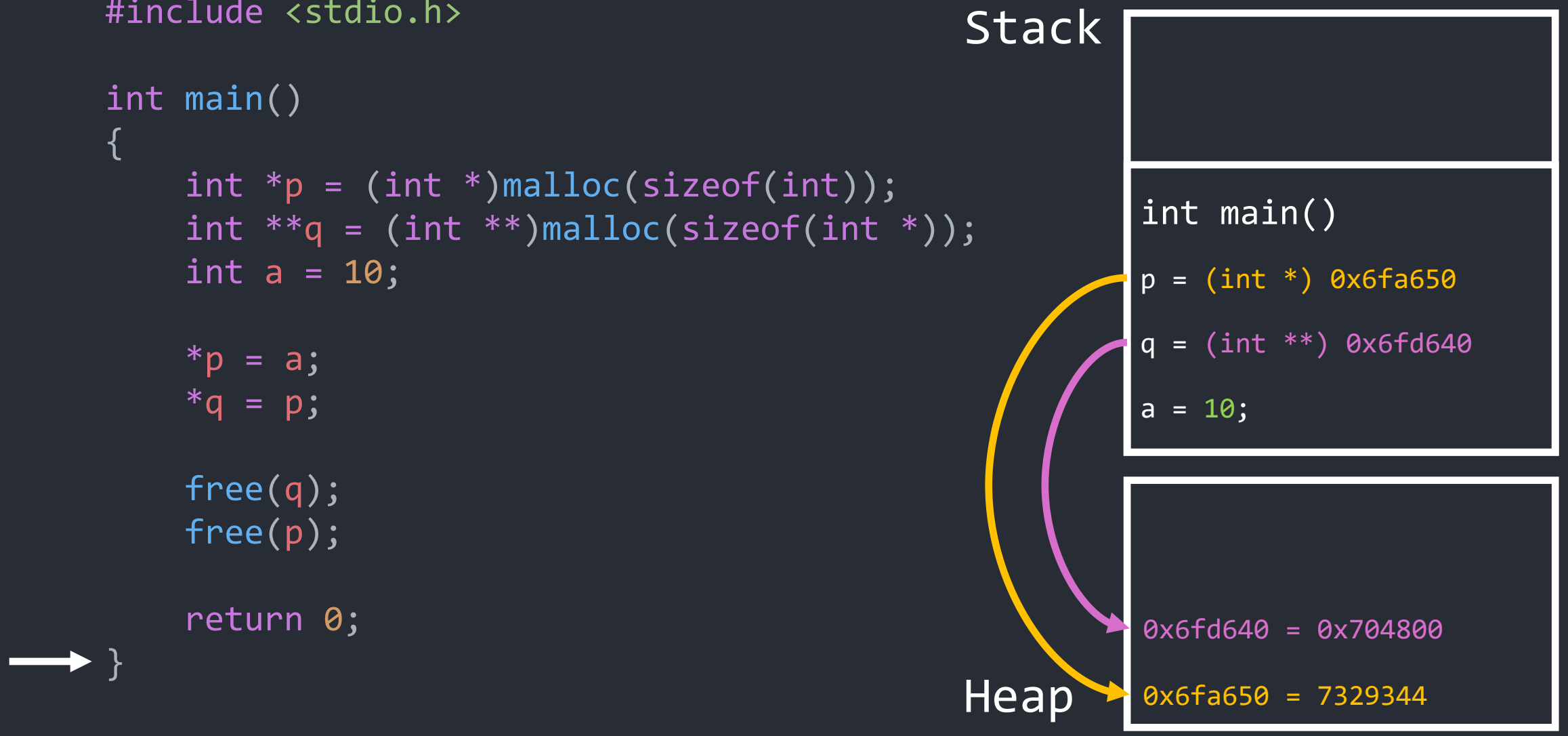
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



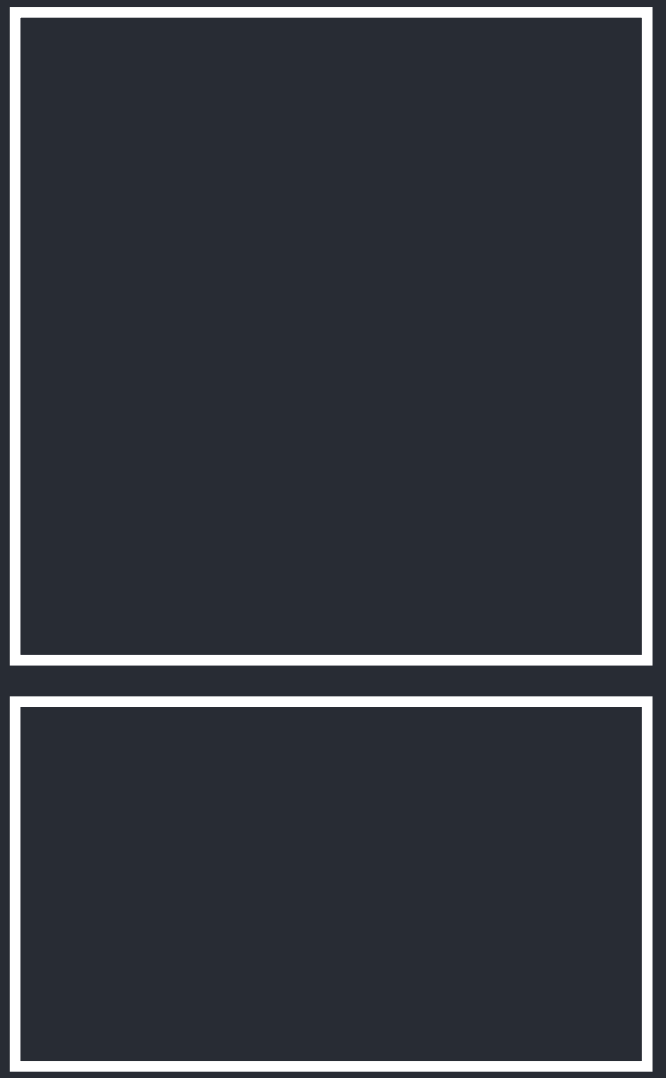
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```



```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```

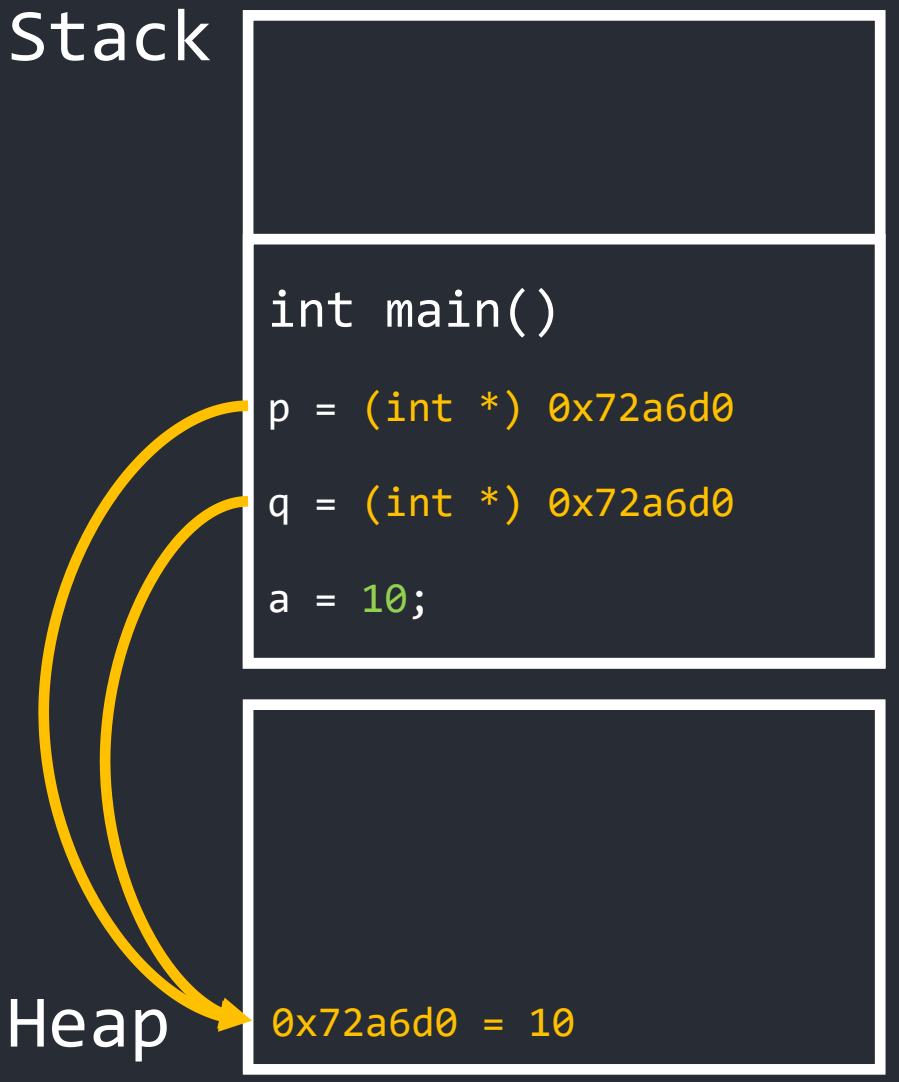


```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int **q = (int **)malloc(sizeof(int *));
    int a = 10;

    *p = a;
    *q = p;

    free(q);
    free(p);

    return 0;
}
```

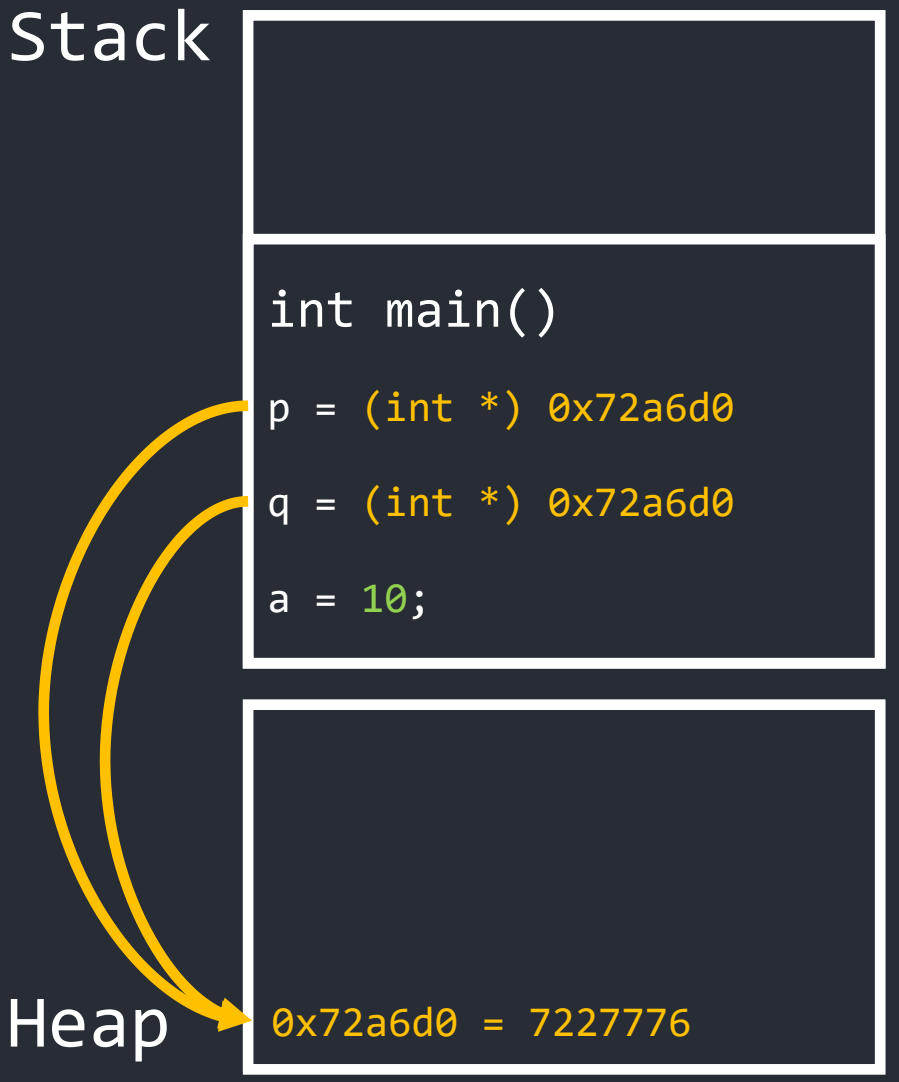


Sometimes, we can point many pointers to **the same variable.**

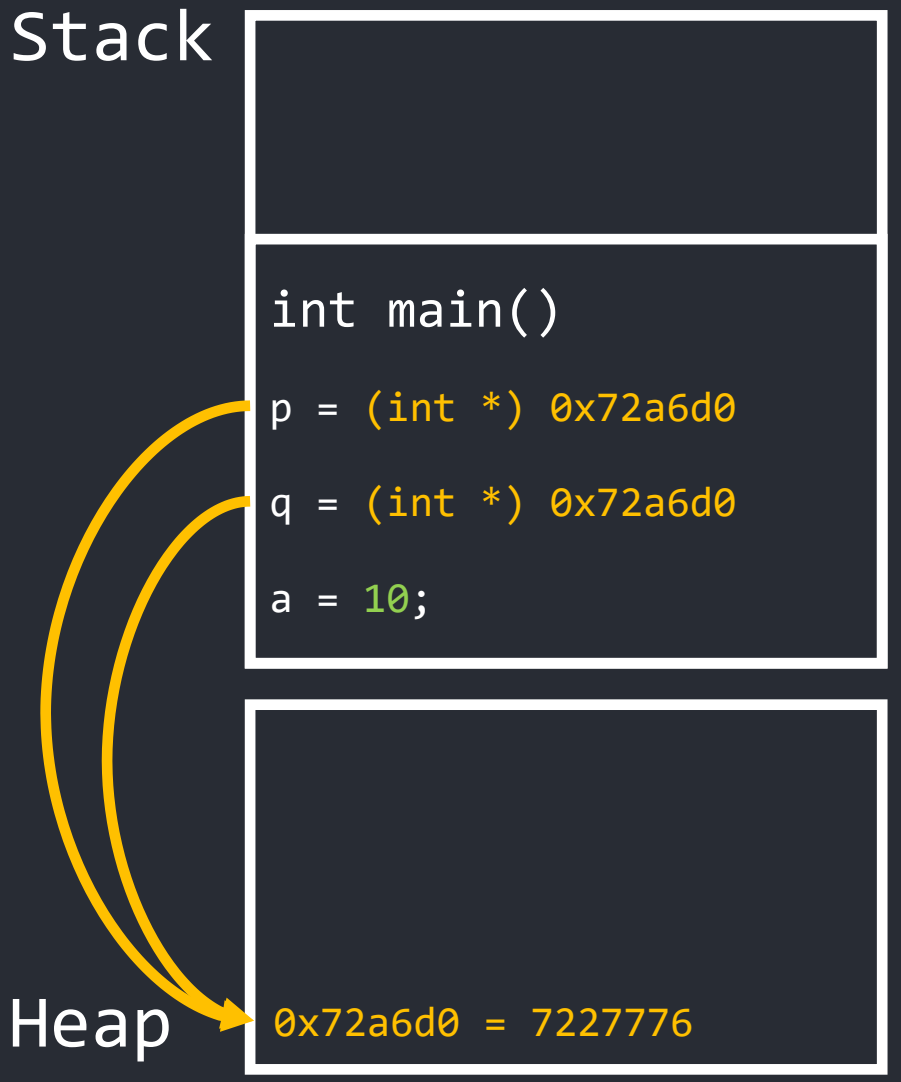
```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int *q = (int *)malloc(sizeof(int));
    int a = 10;

    *p = a;
    q = p;

    free(p);
    printf("%d\n", *q);

    return 0;
}
```



```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int *q = (int *)malloc(sizeof(int));
    int a = 10;

    *p = a;
    q = p;

    free(p);
    printf("%d\n", *q);

    return 0;
}
```



```c
#include <stdio.h>

int main()
{
    int *p = (int *)malloc(sizeof(int));
    int *q = (int *)malloc(sizeof(int));
    int a = 10;

    *p = a;
    q = p;

    free(p);
    7227776

    return 0;
}
```



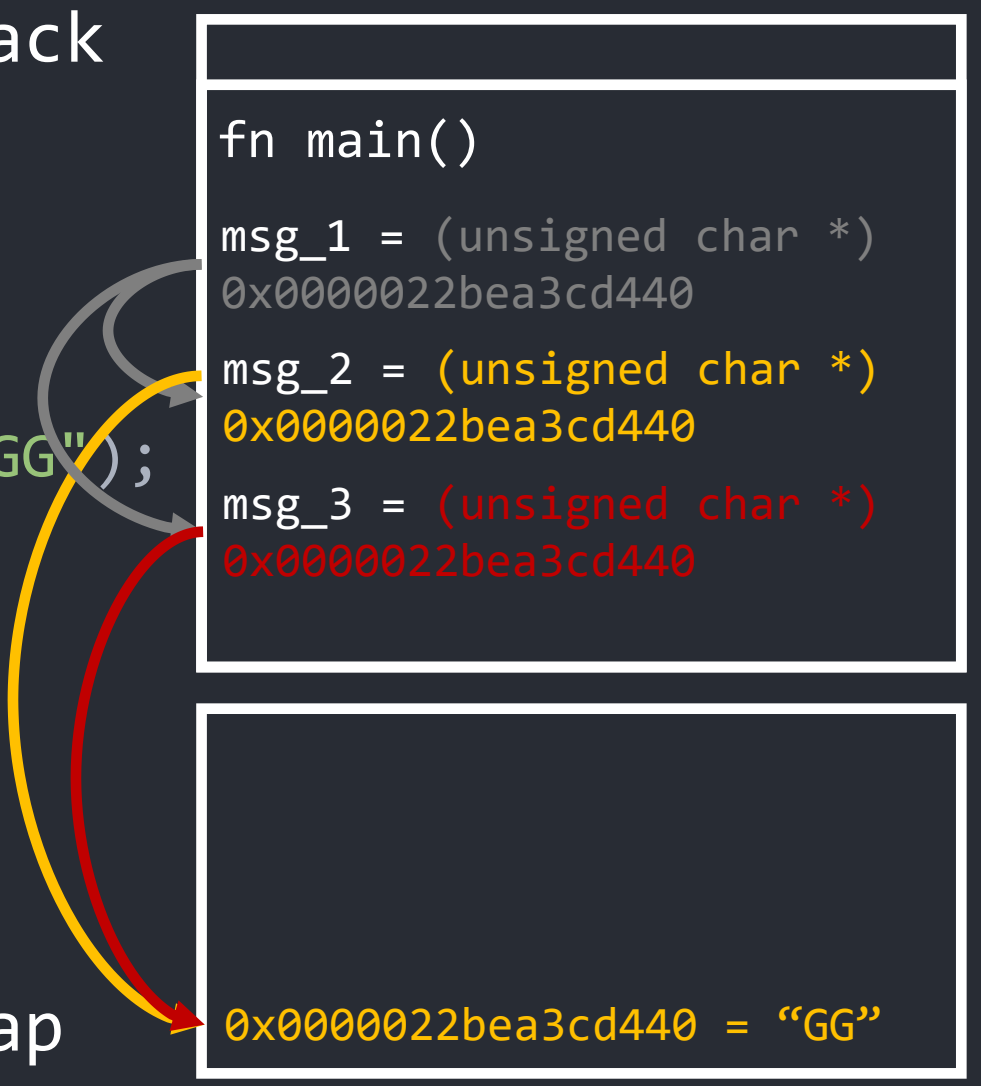
Heap is always
managed by **human.**

But, In Rust
We have “**Ownership**”.

This’s going to manage all memories at compile time.

```
fn main() {
    let msg_1 = String::from("GG");
    let msg_2 = msg_1;
    let msg_3 = msg_1;
}
```

Error: use of moved value: `msg_1`



Let’s talk about **WASM**

Rust

Install Learn Playground Tools Governance Community Blog English (en-US)

WebAssembly

Why Rust?

Predictable performance

No unpredictable garbage collection pauses. No JIT compiler performance cliffs. Just low-level control coupled with high-

Small code size

Small code size means faster page loads. Rust-generated `.wasm` doesn't include extra bloat, like a garbage collector. Advanced optimizations and tree shaking

Modern amenities

A lively ecosystem of libraries to help you hit the ground running. Expressive, zero-cost abstractions. And a welcoming community to help you learn

Get started!

WEBASSEMBLY

Learn more about the fast, safe, and open virtual machine called WebAssembly, and read its standard.

LEARN MORE

Learn how to build, debug, profile, and deploy WebAssembly applications using Rust!

READ THE BOOK

MDN web docs moz://a

Learn more about WebAssembly on the Mozilla Developer Network.

CHECK IT OUT

Production use

CLOUDFLARE

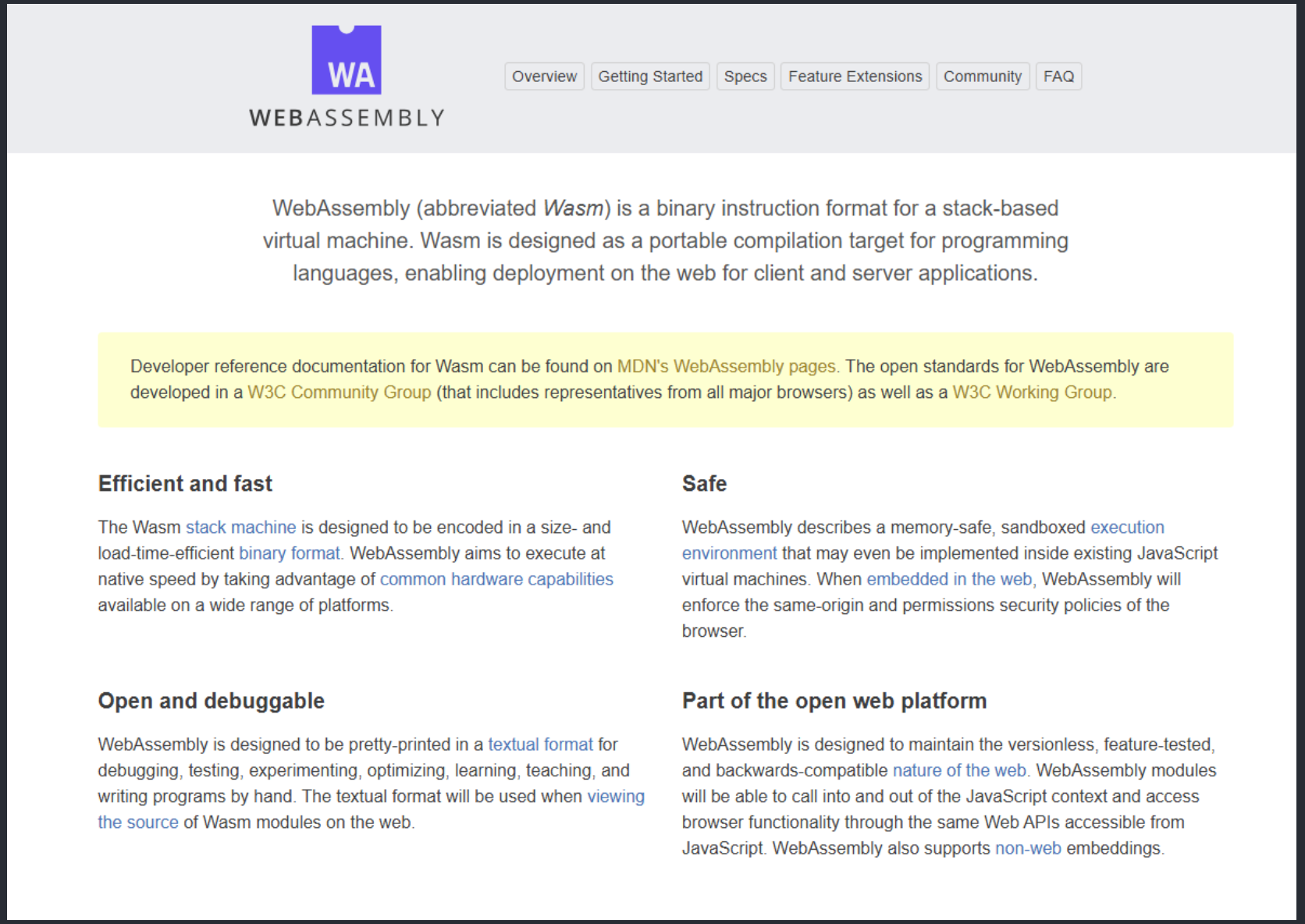
> We can compile Rust to WASM, and call it from Serverless functions woven into the very fabric of the Internet. That's huge and I can't wait to do more of it.
>
> – Steven Pack, Serverless Rust with Cloudflare Workers

> The JavaScript implementation [of the `source-map` library] has accumulated convoluted code in the name of performance, and we replaced it with idiomatic Rust. Rust does not force us to choose between clearly expressing intent and runtime performance.
>
> – Nick Fitzgerald, Oxidizing Source Maps with Rust and WebAssembly

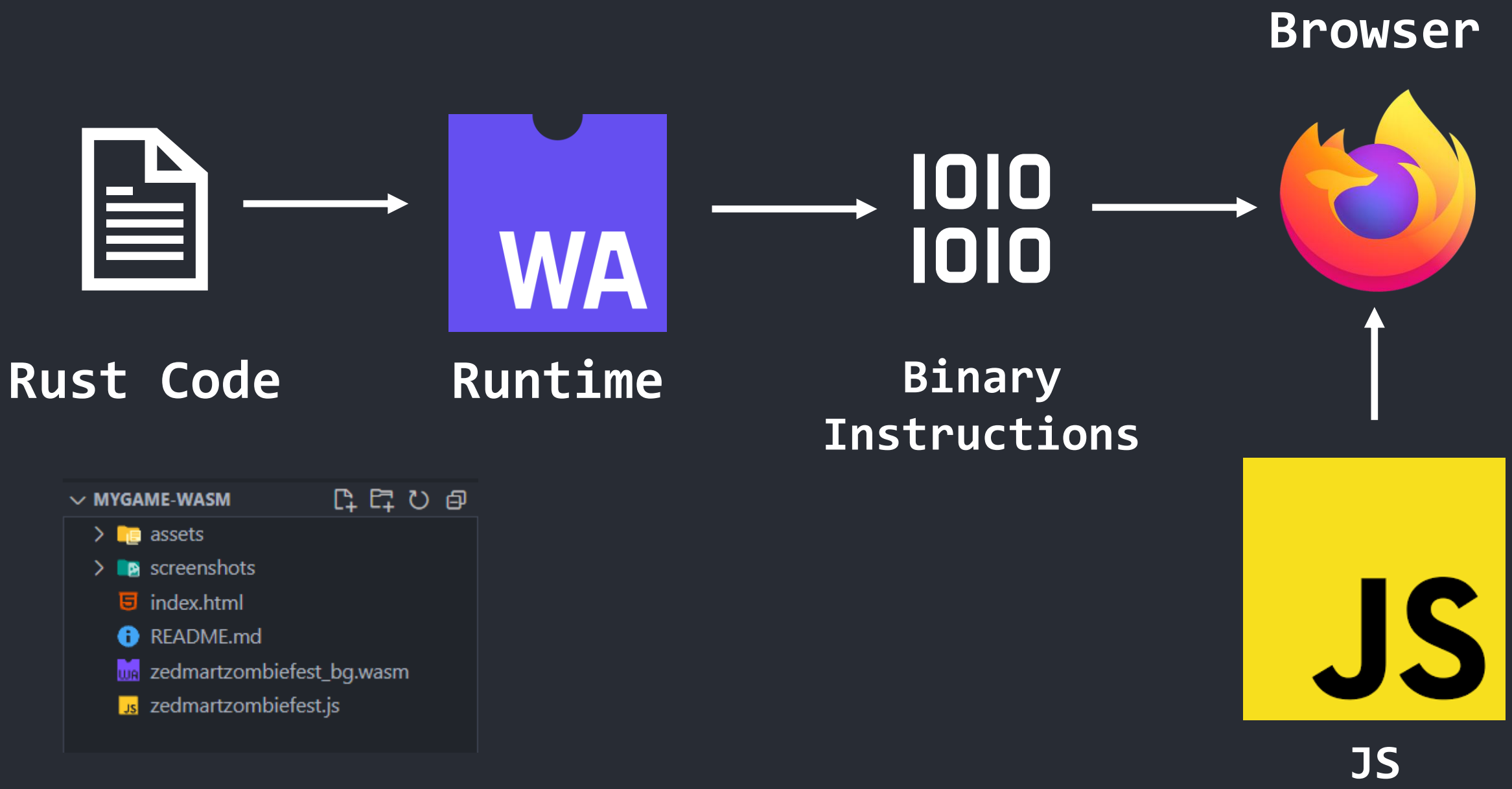
Dropbox

> [Rust's] properties make it easy to embed the DivANS codec in a webpage with WASM, as shown above.
>
> – Daniel Reiter Horn and Jongmin Baek, Building Better Compression Together with DivANS

https://www.rust-lang.org/what/wasm



https://webassembly.org/


Browser
WA
1010
1010
Rust Code
Runtime
Binary
Instructions
MYGAME-WASM
assets
screenshots
index.html
README.md
zedmartzombiefest_bg.wasm
zedmartzombiefest.js
JS
JS

WASM in Rust no garbage collector required