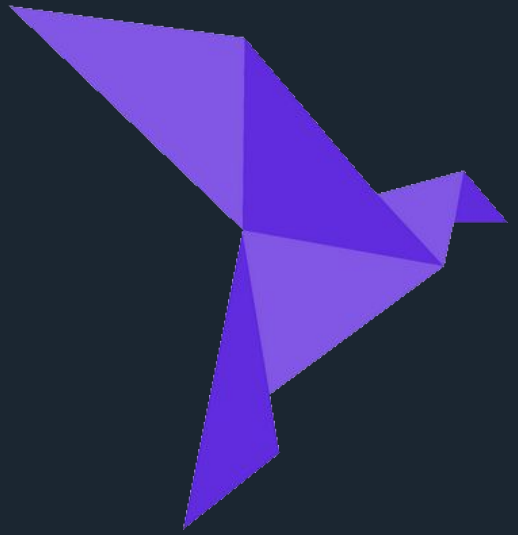
Fundamentals

Please follow these instructions to get set up:

`github.com/rtfeldman/c-workshop-v1`



roc-lang.org

zed.dev

Introduction

What is C?

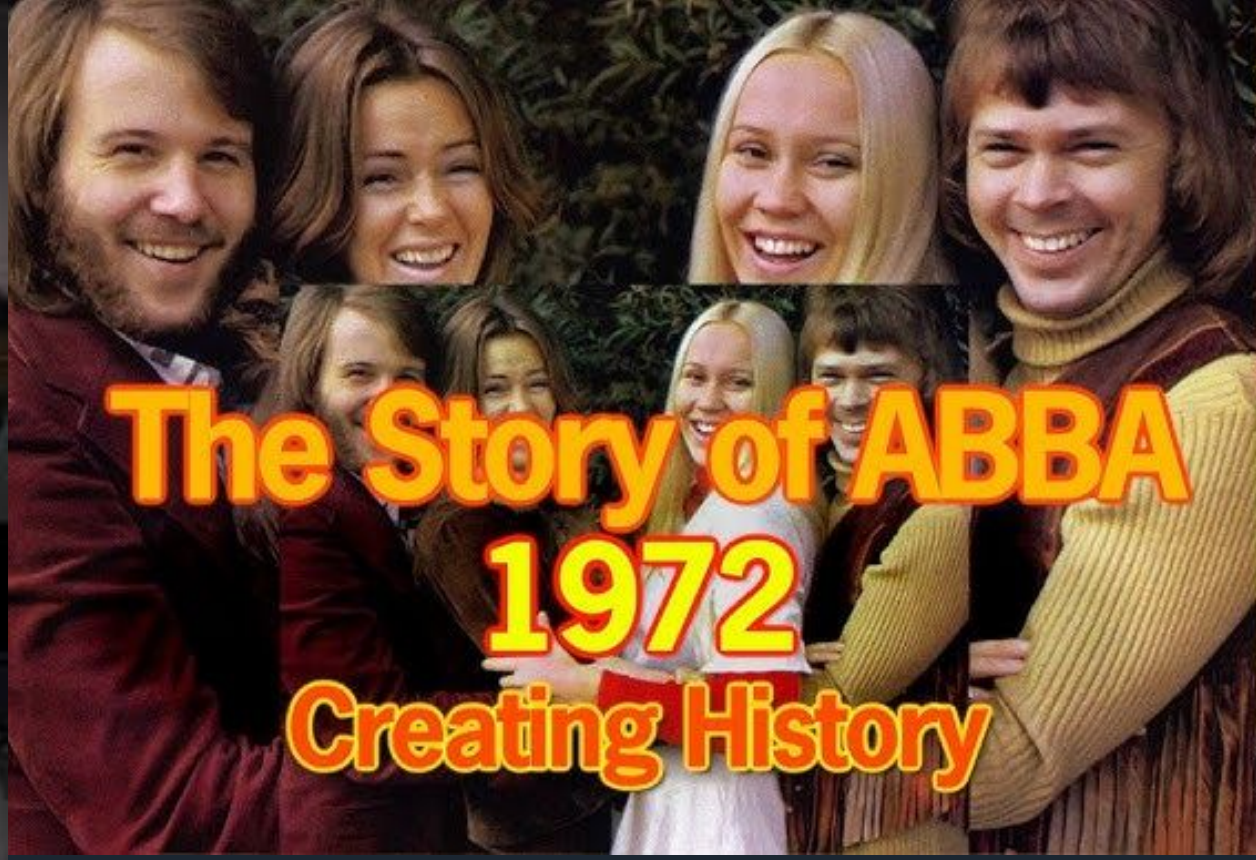
Why is C still popular?

Why no Windows for this course?

Course overview

1. What is C?

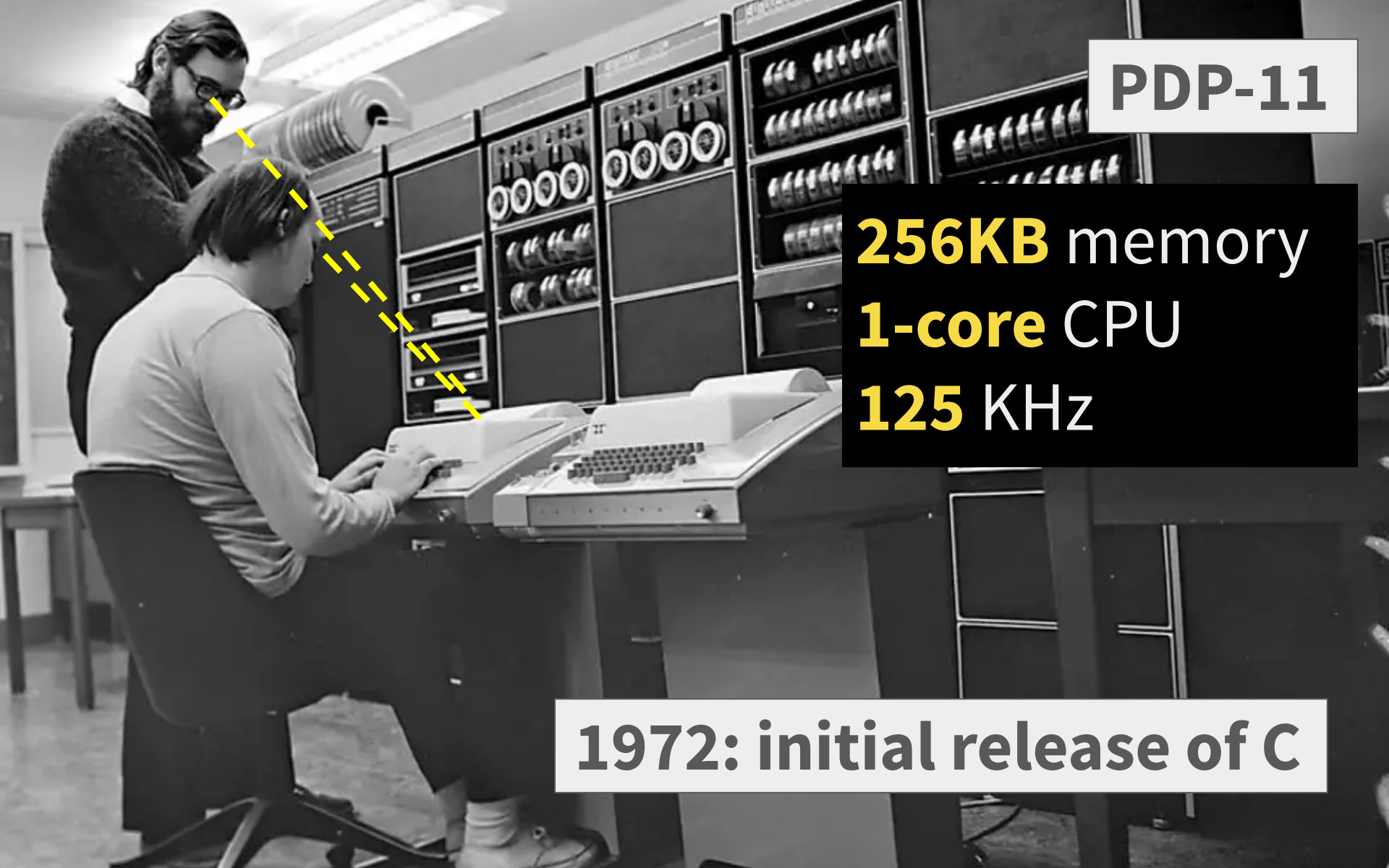
1972: initial release of C



1972: initial release of C


Dennis Ritchie, creator of C
Ken Thompson, UNIX designer
1972: initial release of C


PDP-11
1972: initial release of C


PDP-11
256KB memory
1-core CPU
125 KHz
1972: initial release of C


redis
PostgreSQL
MySQL
TOP 10
PDP-11
256KB memory
1-core CPU
0.001 GHz
UNIX®
MICROSOFT WINDOWS
mac OS
iOS
Linux
android
php
Ruby
python
1972: initial release of C

What is C?

- “Portable assembly”
- Zero-overhead programming
- Almost zero-safety programming
- The language other languages use to talk to each other

Why is C still popular?

Maximum possible performance

thanks to Zero-overhead programming

Much more ergonomic than Assembly

Much simpler than C++

Why is C still popular?

Local Static HTTP Server Perf Olympics

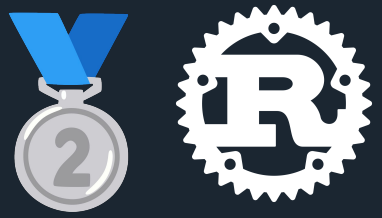
Time
9 ms
5 ms
3 ms
4 ms
5 ms
3 ms
5 ms
2 ms

http-server

Weekly Downloads

2,360,975

Time
6 ms
1 ms
4 ms
3 ms
2 ms
2 ms
2 ms
2 ms



simple-http-server

52,089

Downloads all time

Time
2 ms
3 ms
2 ms
2 ms
1 ms
3 ms
2 ms
1 ms

the ~300 lines of C code we write in this workshop using only the stdlib

Low-overhead C alternatives

C++ (very complex, huge in game dev)

Rust (Zed, Biome, Ripgrep, …) - my course!

Zig (Ghostty, Bun, TigerBeetle, Roc, …)

many others (D, Odin, JAI, Carbon, …)

Low-overhead C alternatives

all support C interop

because C is **the** language that

other languages use to talk to each other

C is the most **universal** language!

Why no Windows for this course?

We're using OS-specific APIs

macOS, Linux, and BSD all use POSIX APIs

WSL also supports POSIX APIs

Windows-specific APIs are often similar

Course Overview

Strings & Numbers

File I/O

Static Webserver

Node.js C Addon

Part 1: Hello, Metal!

"Hello, World!" in C

What the hardware sees

Performance implications

Improving ergonomics

HELLO WORLD

```
#include <unistd.h>  imports write()
main() is a function that returns nothing ("void")
void main() {
    write(1, "Hello, World!", 13);
}           1 means write to stdout
```

HELLO WORLD

```
#include <unistd.h>

void main() {
    write(1, "Hello, World!", 13);
}
```

13 is the length of the string

WHAT THE HARDWARE SEES

```
.LC0:
  .string "Hello, World!"
```

"LC0" is "local constant number zero"

```
main:
  mov edx, 13
  mov esi, OFFSET FLAT:.LC0
  mov edi, 1
  jmp write
```


.LC0:
.string "Hello, World!"
main:
mov edx, 13
mov esi, OFFSET FLAT:.LC0
mov edi, 1
jmp write
write(1, "Hello, World!", 13);

```
.LC0:
   .string "Hello, World!"

main: "set these CPU memory slots"
   mov edx, 13
   mov esi, OFFSET FLAT:.LC0
   mov edi, 1
   jmp write
```

```
write(1, "Hello, World!", 13)
```

```
.LC0:
   .string "Hello, World!"

main:
   mov edx, 13
   mov esi, OFFSET FLAT:.LC0
   mov edi, 1
   jmp write
```

"jump to the OS-provided `write` function"

(the OS will read from those 3 CPU memory slots)

Q: How fast does this program run?

A: As fast as the hardware can possibly run it!

```
.LC0:
   .string "Hello, World!"

main:
   mov edx, 13
   mov esi, OFFSET FLAT:.LC0
   mov edi, 1
   jmp write
```

only the OS can make `write` faster

This is the minimal set of instructions. ***Unbeatably* fast**!

```
.LC0:
  .string "Hello, World!"
main:
  mov edx, 13
  mov esi, OFFSET FLAT:.LC0
  mov edi, 1
  jmp write
```

```
.LC0:
  .string "Hello World!"
main:
  push rbp
  mov rbp, rsp
  mov esi, OFFSET FLAT:.LC0
  mov edi, OFFSET FLAT:std::cout
  call std::basic_ostream<char, std::char_traits<char>>
  mov eax, 0
  pop rbp
  ret
```

```
main_main_pc0:
  TEXT main.main(SB), ABIInternal, $64-0
  CMPQ SP, 16(R14)
  PCDATA $0, $-2
  JLS main_main_pc76
  PCDATA $0, $-1
  PUSHQ BP
  MOVQ SP, BP
  SUBQ $56, SP
  FUNCDATA $0, gclocals·g2BeySu+wFnoycgXfElmcg==(SB)
  FUNCDATA $1, gclocals·EaPwxsZ75yY1hHMVZLmk6g==(SB)
  FUNCDATA $2, main.main.stkobj(SB)
  LEAQ type:string(SB), DX
  MOVQ DX, main..autotmp_8+40(SP)
  LEAQ main..stmp_0(SB), DX
  MOVQ DX, main..autotmp_8+48(SP)
  MOVQ os.Stdout(SB), BX
  NOP
  LEAQ go:itab.*os.File,io.Writer(SB), AX
  LEAQ main..autotmp_8+40(SP), CX
  MOVL $1, DI
  MOVQ DI, SI
  PCDATA $1, $0
  CALL fmt.Fprintln(SB)
  ADDQ $56, SP
  POPQ BP
  RET
main_main_pc76:
  NOP
  PCDATA $1, $-1
  PCDATA $0, $-2
  CALL runtime.morestack_noctxt(SB)
  PCDATA $0, $-1
  JMP main_main_pc0
  TEXT type:.eq.sync/atomic.Pointer[os.dirInfo](SB), DUPOK|NOSPLIT|NOFRAME|ABIInternal, $0-16
  FUNCDATA $0, gclocals·TjPuuCwdlCpTaRQGRKTrYw==(SB)
  FUNCDATA $1, gclocals·J5F+7Qw7O7ve2QcWC7DpeQ==(SB)
  FUNCDATA $5, type:.eq.sync/atomic.Pointer[os.dirInfo].arginfo1(SB)
  FUNCDATA $6, type:.eq.sync/atomic.Pointer[os.dirInfo].argliveinfo(SB)
  PCDATA $3, $1
  MOVQ (AX), CX
  CMPQ (BX), CX
  SETEQ AL
  RET
```

```
.LC0:
  .string "Hello, World!"
main:
  mov edx, 13
  mov esi, OFFSET FLAT:.LC0
  mov edi, 1
  jmp write
```

```
.LC0:
  .string "Hello World!"
main:
  push rbp
  mov rbp, rsp
  mov esi, OFFSET FLAT:.LC0
  mov edi, OFFSET FLAT:std::cout
  call std::basic_ostream<char, std::char_traits<char>>
  mov eax, 0
  pop rbp
  ret
```

Rust, **Zig**, **JAI**, **Odin**, **Carbon**…

compared to C:

- more ergonomic features
- *minimal* extra overhead

HELLO WORLD

```
#include <unistd.h>

void main() {
    write(1, "Hello, World!", 13);
}
```

If anything benchmarks faster than this…

…then the benchmark must be wrong!

HELLO WORLD

```c
#include <unistd.h>

int main() {
    write(1, "Hello, World!", 13);

    return 0;
}
```

HELLO WORLD

```c
#include <stdio.h>
```
(write is in unistd.h, printf is in stdio.h)
```c
int main() {
    printf("Hello, World!");
        wrapper around write()
    return 0;
}
```

HELLO WORLD

```c
#include <stdio.h>

int main() {
    printf("Hello, World!");
           "formatted"
    return 0;
}
```

HELLO WORLD

```c
#include <stdio.h>

int main() {
    int num = 42;
    printf("The number is: %d", num);

    return 0;
}
```

HELLO WORLD

```c
#include <stdio.h>

int main() {
    int num = 42;
    printf("The number is: %d", num);
          "formatted"
    return 0;
}
```

HELLO WORLD

```c
#include <stdio.h>

int main() {
    int num = 42;
    printf("The number is: %d", num);
    return 0;
}
```

Why "**print**"? And why no underscore before **f**?


PDP-11
1972: initial release of C

Summary of Part 1

"Hello, World!" in C

What the hardware sees

Performance implications

Improving ergonomics

Part 1 Exercise

Open `exercises/1.c`

See comments with 👉 in them

Try doing what those comments say!

Part 2: Building HTTP Responses

Numbers in memory

Byte arrays

Null-terminated strings

Getting a string's length

0 $0+2^0$ $0+2^1$ $0+2^1+2^0$

0 1 2 3

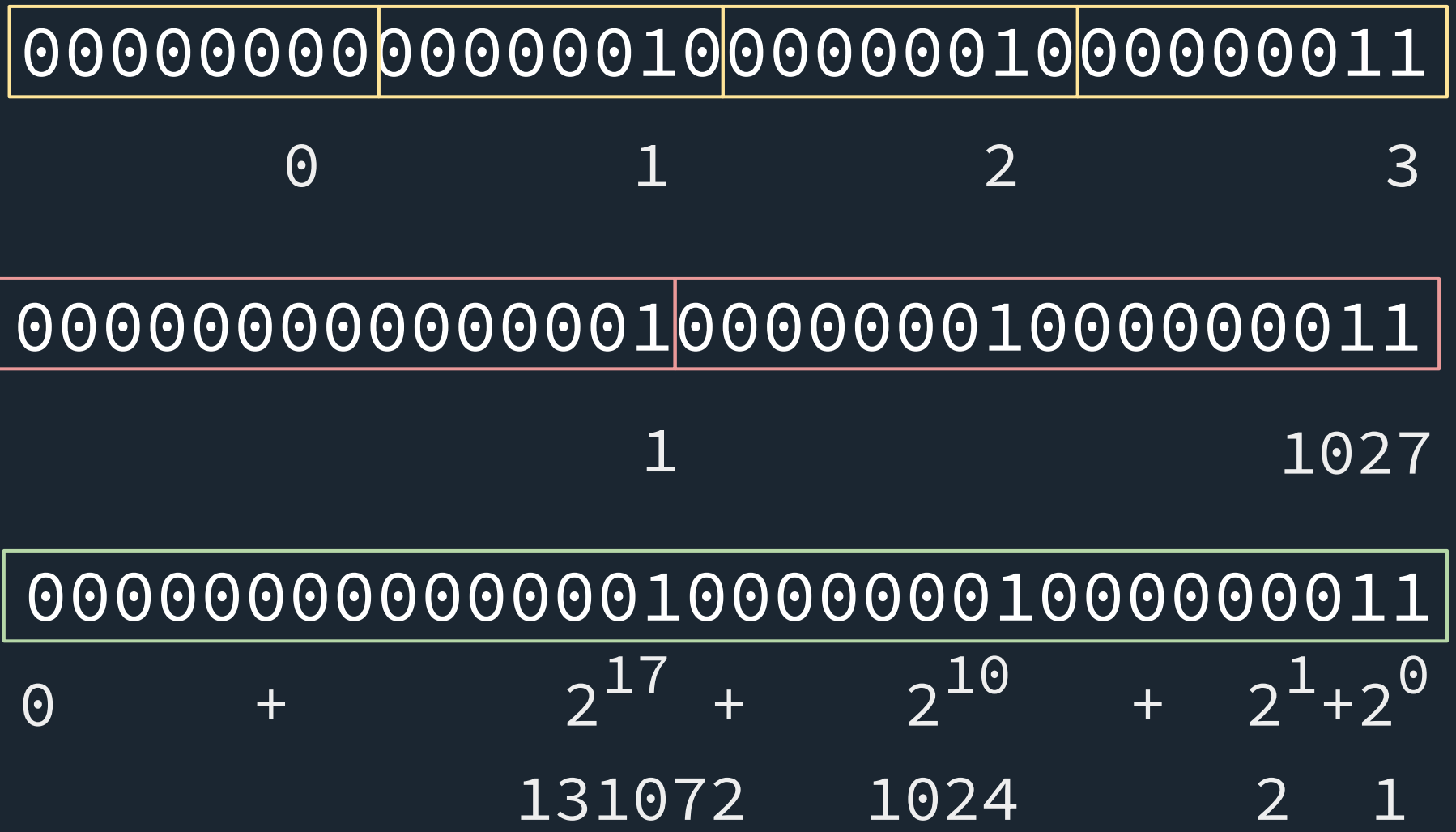
4 00000100

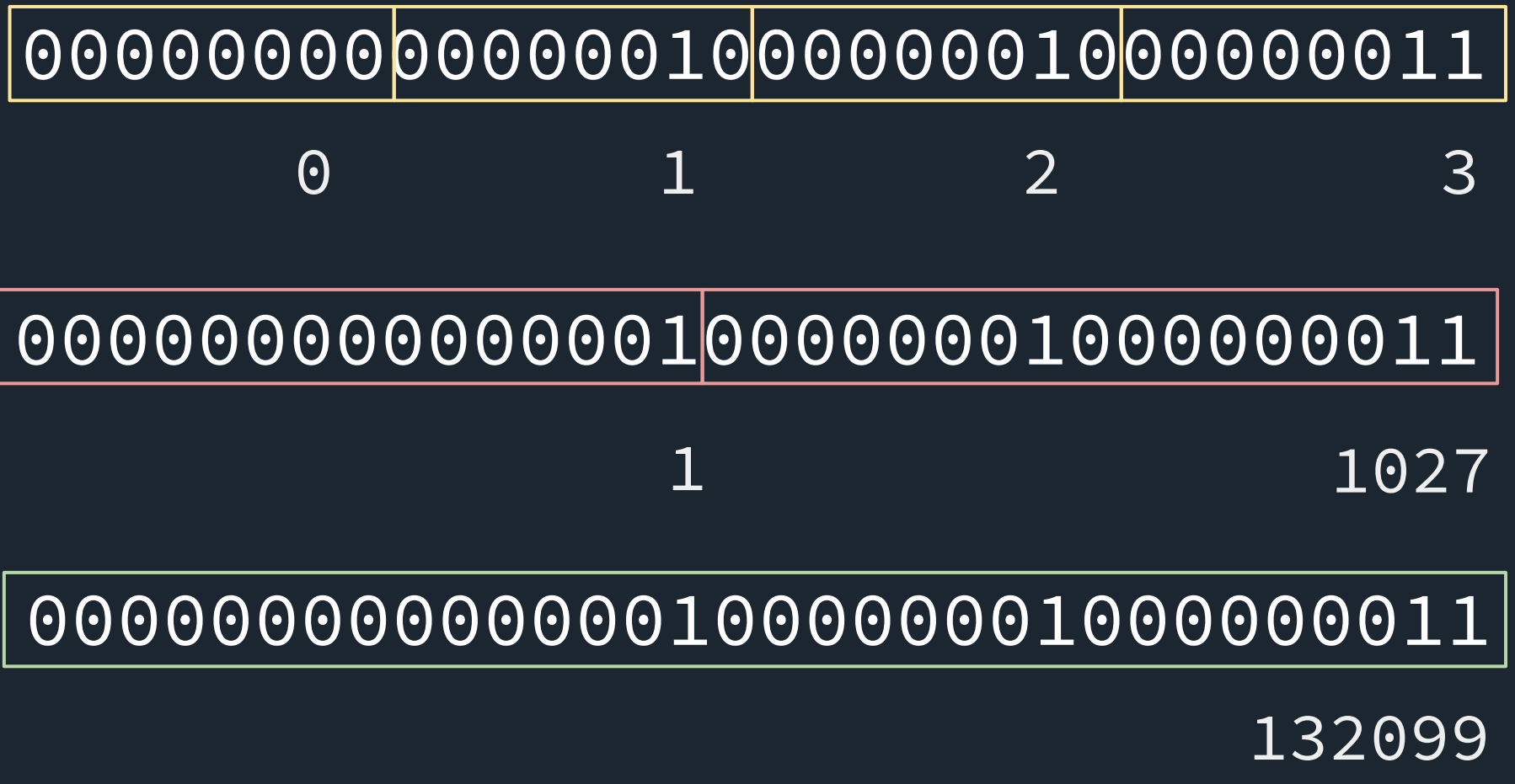
5 00000101

6 00000110

7 00000111

8 00001000


00000000 00000010 00000010 00000011
0 1 2 3
0000000000000001 0000000100000011
1 1027
0000000000000010000000100000011
0 + 2^17 + 2^10 + 2^1+2^0
131072 1024 2 1



we decide how to interpret memory

HELLO WORLD

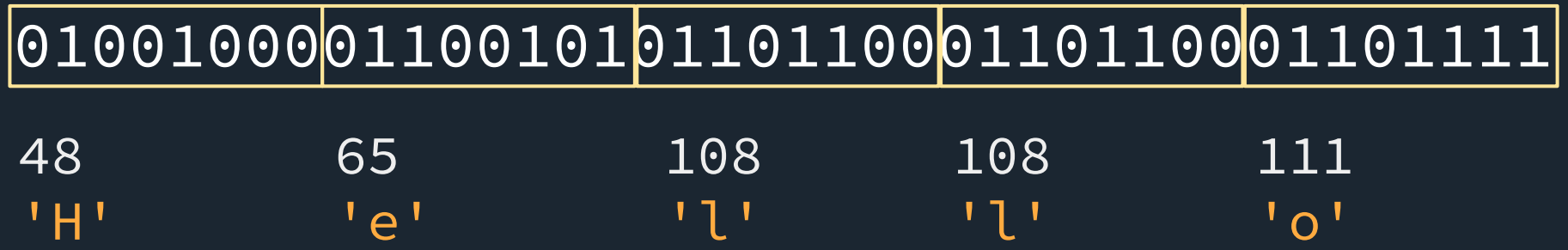
```c
#include <unistd.h>

int main() {
    write(1, "Hello, World!", 13);

    return 0;
}
```

```
write(1, "Hello, World!", 13);
```

first argument is a *32-bit integer*

(meaning 32 ones and zeros in memory)

```
00000000000000000000000000000001
```



ASCII → UTF-8

```
write(1, "Hello, World!", 13);
```

we decide how to interpret memory

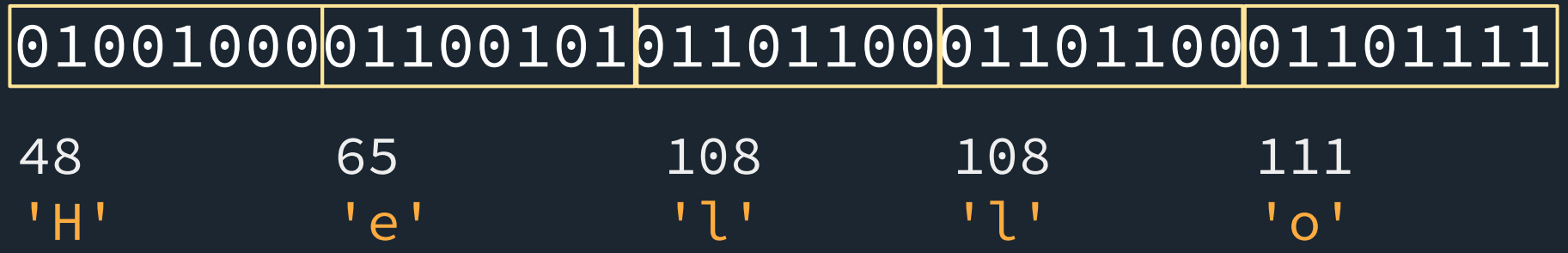
last argument is a *64-bit integer*
(meaning 64 ones and zeros in memory)

```
write(1, "Hello, World!", 13);
```

```
0000000000000000000000000000000000000000000000000000000000001101
```

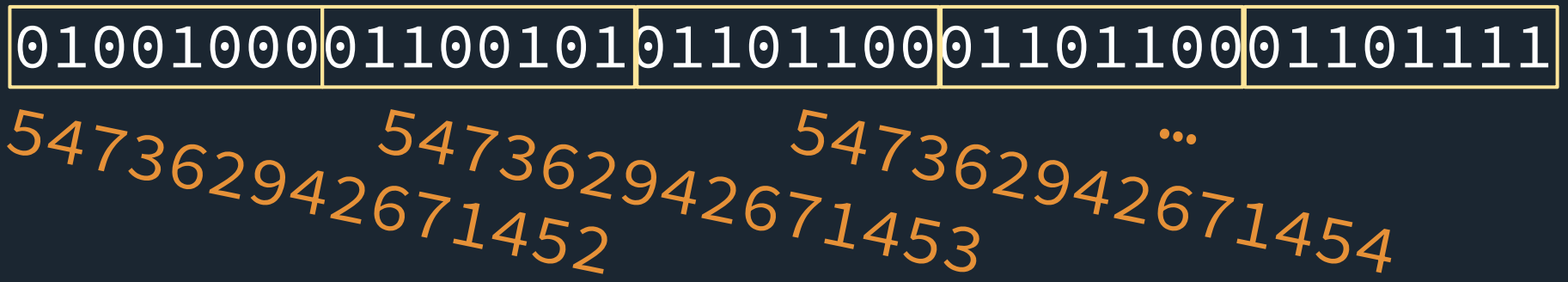
`write` actually takes three integers!

```
write(1, "Hello, World!", 13);
```




```
write(1, 547362942671452, 13);
```

the **memory address** of the first byte in the string



memory is basically a **gigantic array of bytes**

```
                start index                length
write(1, 547362942671452, 13);
```

the **memory address** of the first byte in the string

memory is basically a **gigantic array of bytes**

start index **length**

```
write(1, "Hello, World!", 13);
         547362942671452
```

“Pointer” means “Memory Address”

(“Monads are just monoids in the category of endofunctors, what's the problem?”)

HTTP Responses

https://frontendmasters.com

```
HTTP/1.1 200 OK

<!doctype html><html
```

```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75
H  T  T  P  /  1  .  1     2  0  0     O  K
```

"HTTP/1.1 200 OK"

```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```

"HTTP/1.1 200 OK"

```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```

```
char *header = "HTTP/1.1 200 OK";
```

"address of a byte" (**char** means "byte")

aka "pointer to a byte" ("pointer" means "mem address")

in memory, **header** is an integer

(namely, the memory address of the first byte in the array)

```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```

```
char *header = "HTTP/1.1 200 OK";

write(1, header, 15);
```

```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```

```
char *header = "HTTP/1.1 200 OK";

write(1, header, strlen(header));
```

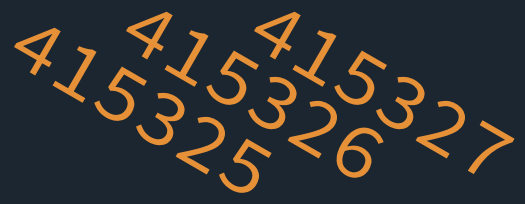
```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```

```c
char *header = 415325;

write(1, header, strlen(header));
```

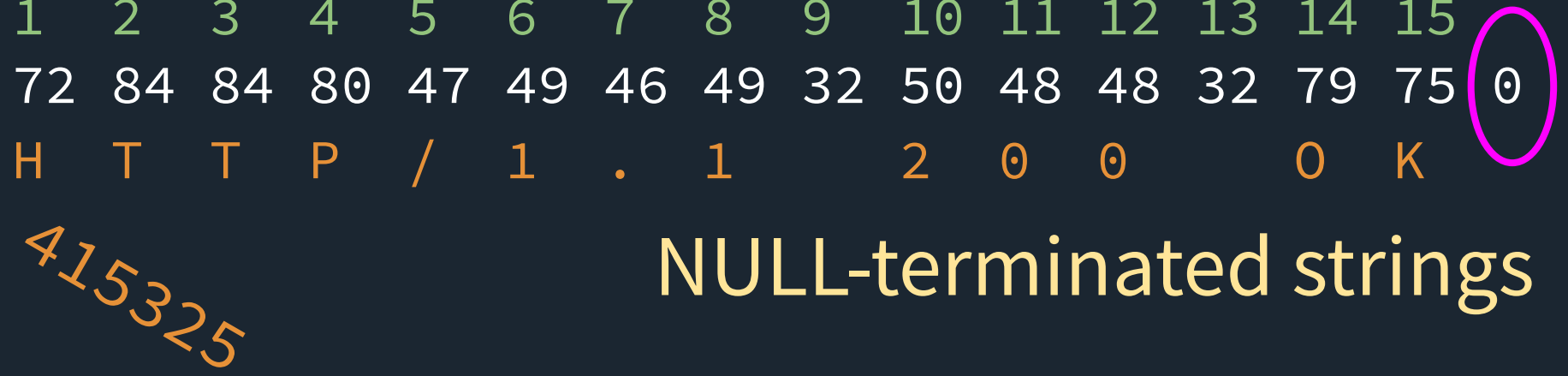
```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```

```
char header[] = 415325;

write(1, 415325, strlen(415325));
```

```
72 84 84 80 47 49 46 49 32 50 48 48 32 79 75 0
H  T  T  P  /  1  .  1     2  0  0     O  K
```



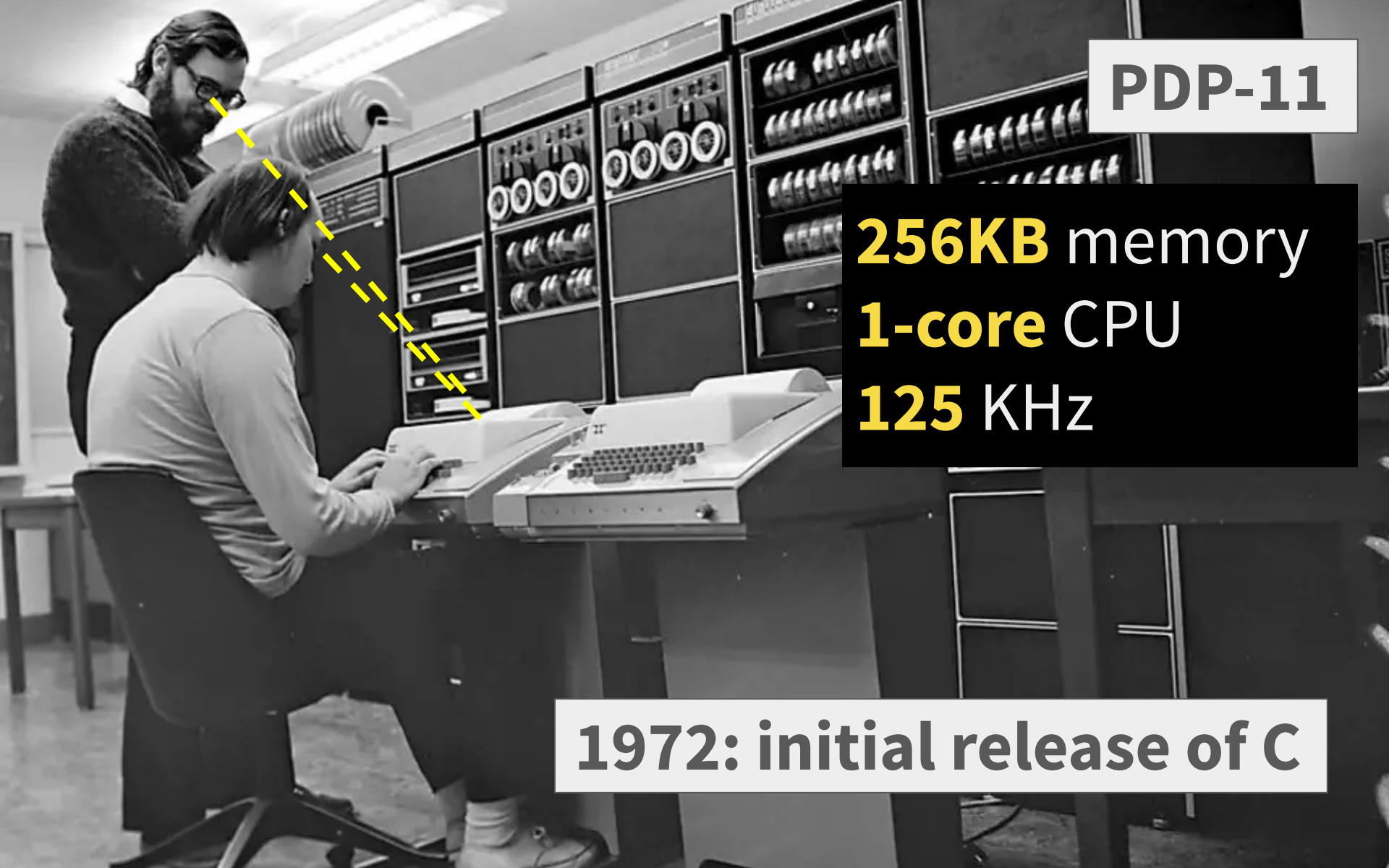

```
char *header = 415325;

write(1, 415325, strlen(415325));
```




```
char *header = 415325;

write(1, 415325, strlen(415325));
```

Isn't `strlen` incredibly slow?!

Why not store the length up front?


PDP-11
256KB memory
1-core CPU
125 KHz
1972: initial release of C

```
char *header = "HTTP/1.1 200 OK";
write(1, header, strlen(header));
```

15

Isn't `strlen` incredibly slow?

Yes, but this gets optimized away

```c
#include <unistd.h>


    char *header = "HTTP/1.1 200 OK";
    write(1, header, strlen(header));
```

```c
#include <unistd.h>
#include <string.h>

    char *header = "HTTP/1.1 200 OK";
    write(1, header, strlen(header));
```

```c
#include <unistd.h>
#include <string.h>

int main() {
    char header[] = "HTTP/1.1 200 OK";
    write(1, header, strlen(header));
    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *header = "HTTP/1.1 200 OK";
    printf("Header: %s", header);
    return 0;
}
```

prints it as a string
(assumes null-terminated!)

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *header = "HTTP/1.1 200 OK";
    printf("Header: %d", header);
    return 0;
}
```

prints it as a 32-bit integer

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *header = "HTTP/1.1 200 OK";
    printf("Header: %zud", header);
    return 0;
}
```

prints it as a 64-bit integer

memory is basically a **gigantic array of bytes**

Summary of Part 2

Numbers in memory

Byte arrays

Null-terminated strings

Getting a string's length

Part 2 Exercise

Open `exercises/2.c`

See comments with 👉 in them

Try doing what those comments say!

Part 3: Parsing HTTP Requests

Defining new functions

Iteration

Copying memory

Readonly memory

Incoming request:

```
GET /blog HTTP/1.1
Host: www.example.com
User-Agent: …
```

File to open:

```
blog/index.html
```

Incoming request:	File to open:
`GET /blog`	`blog/index.html`
`GET /blog/`	`blog/index.html`

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *req = "GET /blog HTTP/1.1…";




    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *req = "GET /blog HTTP/1.1…";
    char *path = to_path(req);



    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *req = "GET /blog HTTP/1.1…";
    char *path = to_path(req);

    printf("Path: %s", path);

    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

int main() {
    char *req = "GET /blog HTTP/1.1…";
    char *path = to_path(req);
    // …
}

char *to_path(char *req) {
    // …
```

ERROR: called before defined!

```c
char *to_path(char *req);
// Forward-declare to_path

int main() {
    char *req = "GET /blog HTTP/1.1…";
    char *path = to_path(req);
    // …
}

char *to_path(char *req) {
    // …
```

ERROR: called before defined!

```
char *to_path(char *req) {




}

int main() {
```

```
 integer                integer
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:          "blog/index.html"









}

int main() {
```

```c
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;




}

int main() {
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {




    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {



        start = start + 1;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {



        start += 1;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:      "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
```

```
        start++;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
        if (start[0] == '\0')


        start++;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
        if (start[0] == 0)


        start++;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:      "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
        if (!start[0])


        start++;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
        if (!start[0])
            return NULL;

        start++;
    }
}
```

```c
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
        if (!start[0])
            return NULL;

        start++;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start = req;
    while (start[0] != ' ') {
        if (!start[0]) {
            return NULL;
        }
        start++;
    }
}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start;
    for (
        start = req;
        start[0] != ' ';
        start++
    ) {
        if (!start[0]) { return NULL; }
    }
```

```c
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"

    for (
        char *start = req;
        start[0] != ' ';
        start++
    ) {
        if (!start[0]) { return NULL; }
    }
```

start is only in scope inside this loop

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start;
    for (start = req; …) { … }




}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start;
    for (start = req; …) { … }
    start++; // skip over the space




}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start;
    for (start = req; …) { … }
    start++; // skip over the space

    printf("Start: %s", start);


}
```

```c
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start;
    for (start = req; …) { … }
    start++; // skip over the space

    printf("Start: %s", start);
    //      "Start: /blog HTTP/1.1…"

}
```

```c
char *to_path(char *req) {
    // Input: "GET "
    // Goal:  ""
    char *start;
    for (start = req; …) { … }
    start++; // skip over the space

    printf("Start: %s", start);


}
```

```c
char *to_path(char *req) {
    // Input: "GET "
    // Goal:  ""
    char *start;
    for (start = req; …) { … }
    start++; // skip over the space
    // start[0] == 0
    printf("Start: %s", start);


}
```

```c
char *to_path(char *req) {
    // Input: "GET "
    // Goal:  ""
    char *start;
    for (start = req; …) { … }
    start++; // skip over the space
    // start[0] == 0
    printf("Start: %s", start);
    //      "Start: "

}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end;
    for (start = req; …) { … }
    start++; // skip over the space




}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end;
    for (start = req; …) { … }
    start++; // skip over the space
    for (end = start; …) { … }



}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these




}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these

    // Ensure there's a "/" there




}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:          "blog/index.html"
    char *start, *end; … // set these

    if (end[-1] == '/') {
        // e.g. if it's "GET /blog/ …"
    }


}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these

    if (end[-1] == '/') {
        end--;
    }


}
```

```
char *to_path(char *req) {
    // Input: "GET /blog HTTP/1.1…"
    // Goal:          "blog/index.html"
    char *start, *end; … // set these

    if (end[-1] == '/') {
        end--;
    } else {
        end[0] = '/';
    }
}
```

mutates original request string!

```
char *to_path(char *req) {
    // Input: "GET /blog/ HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these


    // write "index.html\0" after '/'



}
```

```c
char *to_path(char *req) {
    // Input: "GET /blog/ HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these


    // write "index.html\0" after '/'
    memcpy(end + 1, "index.html", 11);


}
```

```
char *to_path(char *req) {
    // Input: "GET /blog/ HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these

    char *default_file = "index.html";
    memcpy(
        end + 1,
        default_file,
        strlen(default_file) + 1);
}
```

```
const char *DEFAULT_FILE = "index…";
char *to_path(char *req) {
    // Input: "GET /blog/ HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these


    memcpy(
        end + 1,
        DEFAULT_FILE,
        strlen(DEFAULT_FILE) + 1);
}
```

```
const char *DEFAULT_FILE = "index…";
char *to_path(char *req) {
    // Input: "GET /blog/ HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these


    memcpy(end + 1, DEFAULT_FILE, …);


    return start;
}
```

```
const char *DEFAULT_FILE = "index…";
char *to_path(char *req) {
    // Input: "GET /blog/ HTTP/1.1…"
    // Goal:       "blog/index.html"
    char *start, *end; … // set these


    memcpy(end + 1, DEFAULT_FILE, …);
    // ⚠️ What if req is too short?
    // (This comes up in the exercise)
    return start;
}
```

```
const char *DEFAULT_FILE = "index…";

char *to_path(char *req) {
    …
}

int main() {
    char *req = "GET /blog HTTP/1.1…";



}
```

```
const char *DEFAULT_FILE = "index…";

char *to_path(char *req) {
    … mutates original request string!
}

int main() {
    char *req = "GET /blog HTTP/1.1…";
    char *path = to_path(req);
                 bus error
    printf("Path: %s", path);
}
```

this string is a **readonly** constant!

```c
const char *DEFAULT_FILE = "index…";

char *to_path(char *req) {
    …
}

int main() {
    char req[] = "GET /blog HTTP/1.1…";
    char *path = to_path(req);

    printf("Path: %s", path);
}
```

Summary of Part 3

Defining new functions

Iteration

Copying memory

Readonly memory

Part 3 Exercise

Open `exercises/3.c`

See comments with 👉 in them

Try doing what those comments say!

Part 4: File I/O

Opening a file

Getting its size

Reading its contents

Stack vs. heap

Incoming request:	File to open:
`GET /blog`	`blog/index.html`
`GET /blog/`	`blog/index.html`

```
char *path = "example.txt";
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);

close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);

if (fd == -1) {
    // Handle errors
}




close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);

if (fd == -1) {
    // Handle errors
}

char buffer[100];
ssize_t length = read(fd, buffer, 100);



close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);

if (fd == -1) {
    // Handle errors
}

char buffer[100];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors


close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);

if (fd == -1) {
    // Handle errors
}
```

what if I want to read *all* the bytes?

```c
char buffer[100];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

// 1. Get the size of the file on disk




close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

// 1. Get the size of the file on disk
// 2. Make a buffer of that many bytes




close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

// 1. Get the size of the file on disk
// 2. Make a buffer of that many bytes
// 3. Read that many bytes into the buffer




close(fd);
```

```
struct stat {
    dev_t st_dev; // ID of device w/ file on it
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    link_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID if special file
    off_t    st_size; // total size, in bytes
    blksize_t st_blksize; // block size for I/O
    blkcnt_t st_blocks; // # of blocks allocated
    time_t st_atime; // last access time
    time_t st_mtime; // last modification time
    time_t st_ctime; // last status change time
}; // 144 bytes total
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;




close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata; // 144 bytes of memory




close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata; // 144 bytes of memory

if (fstat(fd, &metadata) == -1) {
    // Error
}

close(fd);
```

address of the first byte of those 144 bytes

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;

if (fstat(fd, &metadata) == -1) {
    // Error
}
// Now metadata has had its 144B populated


close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;
Why doesn't fstat just return the stat struct?
if (fstat(fd, &metadata) == -1) {
    // Error
}
// Now metadata has had its 144B populated
printf("%ld bytes\n", metadata.st_size);

close(fd);
```

Why doesn't `fstat` just `return` the `stat` struct?

Returning the `stat` struct means returning all 144B

If a caller wants to return it, they copy all 144B again

If another caller wants to return it…another 144B copy

This would get expensive!

Why doesn't `fstat` just `return` a `stat` address?

When functions return, memory for their variables gets freed. (Otherwise, calling them would leak memory!)

This means they can't safely return addresses to variables they defined locally.

```
char *to_path(char *req) { … }
```

address into an *argument's* memory, not a local variable's memory!

Why doesn't `fstat` just `return` a `stat` address?

Functions can reserve *long-lived* memory chunks using `malloc()`, and return the address of that memory.

However, that memory will leak unless some other function later calls `free()` on the address.

Calling `free()` at the wrong time can cause bugs!

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;

if (fstat(fd, &metadata) == -1) {
    // Error
}
// Now metadata has had its 144B populated
printf("%ld bytes\n", metadata.st_size);

close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);

if (fd == -1) {
    // Handle errors
}
```

what if I want to read *all* the bytes?

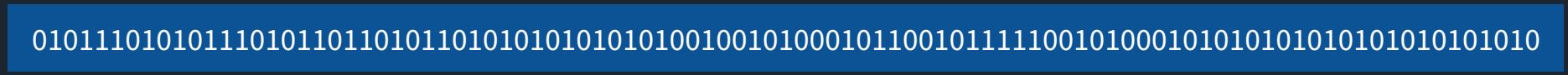
```c
char buffer[100];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;
if (fstat(fd, &metadata) == -1) { … }

char buffer[100];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;
if (fstat(fd, &metadata) == -1) { … }
char buffer[metadata.st_size];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```

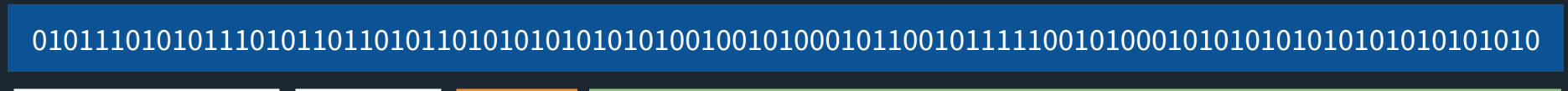
segmentation fault

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata; stack is fixed-size by default
if (fstat(fd, &metadata) == -1) { … }
// Reserve `st_size` bytes on the stack
char buffer[metadata.st_size];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```

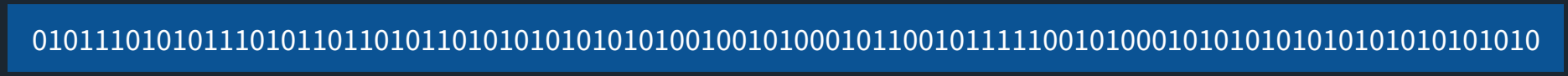


executable global stack heeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeap
+constants vars

```
struct stat metadata;
if (fstat(fd, &metadata) == -1) { … }
// Reserve `st_size` bytes on the stack
char buffer[metadata.st_size];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```




```
struct stat metadata;
if (fstat(fd, &metadata) == -1) { … }
// Reserve `st_size` bytes on the heap
char *buffer = malloc(metadata.st_size);
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);

close(fd);
```



executable +constants | global vars | stack | heeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeap

on macOS 15.3, reads over 2GB failed

fix: `read` multiple chunks

```
struct stat metadata;
if (fstat(fd, &metadata) == -1) { … }
char *buffer = malloc(metadata.st_size);
ssize_t length = fails if too many bytes read at once
      read(fd, buffer, metadata.st_size);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);
free(buffer);
close(fd);
```

```c
char *path = "example.txt";
int fd = open(path, O_RDONLY);
if (fd == -1) { … } // Handle errors

struct stat metadata;
if (fstat(fd, &metadata) == -1) { … }

char buffer[100];
ssize_t length = read(fd, buffer, 100);
if (length == -1) { … } // Handle errors
printf("Bytes read: %zd\n", length);
close(fd);
```

fix: read multiple chunks

stack is good for reading chunks!

no need to free(), and better perf than malloc()

Part 4 Summary

Opening a file

Getting its size

Reading its contents

Stack vs. heap

Part 5: Network I/O

Opening a socket

Listening for connections

Reading from a socket

Writing to a socket

LIBC ❤️ FILE DESCRIPTORS

```
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
```

IPv4 (use `AF_INET6` for IPv6)

TCP (use `SOCK_DGRAM` for UDP)

protocol number

```
int socket_fd = socket( … );
int opt = 1;
setsockopt(
    socket_fd,
    SOL_SOCKET,
    SO_REUSEADDR,
    &opt,
    sizeof(opt)
);
// Handle errors
```

Without this, sometimes when you're stopping and restarting the server a lot, you can get "address in use" even when the server isn't running.

```
int socket_fd = socket( … );
int opt = 1;
setsockopt(
    socket_fd,
    SOL_SOCKET,
    SO_REUSEADDR,
    &opt,
    sizeof(opt)
);
// Handle errors
```

Make a local variable

(a "boolean" in this case)

and pass in its address.

```
int socket_fd = socket( … );
int opt = 1;
setsockopt(
    socket_fd,
    SOL_SOCKET,
    SO_REUSEADDR,
    &opt,
    sizeof(opt)
);
// Handle errors
```

`sizeof` returns the number of bytes the argument takes up in memory (4 bytes for `int`)

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );

struct sockaddr_in address; // IPv4

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(8080);
```

localhost:8080

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );

struct sockaddr_in address;
bind(
    socket_fd,
    (struct sockaddr *)&address,
    sizeof(address)
);
// Handle errors
```

```
struct sockaddr_in {
    sa_family_t sin_family;
    uint16_t sin_port;
    …
};
struct sockaddr_in6 {
    sa_family_t sin6_family;
    uint16_t sin6_port;
    …
};
                        struct sockaddr {
                            sa_family_t sa_family;
                            char sa_data[];
                        };
```

```
struct sockaddr_in {
    sa_family_t sin_family;
    uint16_t sin_port;
    …
};
struct sockaddr_in6 {
    sa_family_t sin6_family;
    uint16_t sin6_port;
    …
};
```

2 bytes

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[];
};
```

covers bytes of multiple fields!

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );

struct sockaddr_in address;
bind(
    socket_fd,
    (struct sockaddr *)&address,
    sizeof(address)
);
```

confirming that this memory address can be treated as a `sockaddr` struct

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );

struct sockaddr_in address;
bind(
    socket_fd,
    (struct sockaddr *)&address,
    sizeof(address)
);
```

Tells `bind` how many **actual** bytes are in the second argument, since it may be more than a `sockaddr` struct

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );
struct sockaddr_in address; bind( … );

listen(socket_fd, 4);
```

number of pending connections that can stack up before we start refusing new ones

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );
struct sockaddr_in address; bind( … );
listen(socket_fd, 4);
printf("Listening on localhost:8080");
```

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );
struct sockaddr_in address; bind( … );
listen(socket_fd, 4);
printf("Listening on localhost:8080");
char req[MAX_REQUEST_BYTES + 1];
```

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );
struct sockaddr_in address; bind( … );
listen(socket_fd, 4);
printf("Listening on localhost:8080");
char req[MAX_REQUEST_BYTES + 1];
int addrlen = sizeof(address);
```

```
int socket_fd = socket( … );
int opt = 1; setsockopt( … );
struct sockaddr_in address; bind( … );
listen(socket_fd, 4);
printf("Listening on localhost:8080");
char req[MAX_REQUEST_BYTES + 1];
int addrlen = sizeof(address);
while (1) {
    // Wait for requests to come in
}
```

```
int addrlen = sizeof(address);
while (1) {
    int req_socket_fd = accept(
        socket_fd,
        (struct sockaddr *)&address,
        (socklen_t*)&addrlen
    );
    // Get data from req_socket_fd
    close(req_socket_fd);
}
```

wait for the queue to get a **connection**, then dequeue it as a **new socket**

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read(
        req_socket_fd,
        req,
        MAX_REQUEST_BYTES
    );
    close(req_socket_fd);
}
```

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read( … );
    char *path = to_path(req);



    close(req_socket_fd);
}
```

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read( … );
    char *path = to_path(req);
    int fd = open(path, O_RDONLY);

    close(fd);
    close(req_socket_fd);
}
```

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read( … );
    int fd = … // read file contents


    close(fd);
    close(req_socket_fd);
}
```

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read( … );
    int fd = … // read file contents

    write(1, contents, length);
    close(fd);
    close(req_socket_fd);
}
```

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read( … );
    int fd = … // read file contents

    write(req_socket_fd, contents, …);
    close(fd);
    close(req_socket_fd);
}
```

```
char req[MAX_REQUEST_BYTES + 1];
while (1) {
    int req_socket_fd = accept( … );
    ssize_t bytes_read = read( … );
    int fd = … // read file contents
    write(req_socket_fd, "200 OK\n\n"…
    write(req_socket_fd, contents, …);
    close(fd);
    close(req_socket_fd);
}
```

Recap of `while(1)` Loop

`accept()` waits for a new connection, then gives us a fd

`read()` gets bytes from that connection (from browser)

`to_path()` parses those req bytes into a filesystem path

`open()`, `fstat()`, `read()` to get the file's full contents

`write()` the full contents to the socket (after a header)

Part 5 Summary

Opening a socket

Listening for connections

Reading from a socket

Writing to a socket

Part 6: Preprocessor Macros

Bit Shifts

Switches

Preprocessor Macros

Platform-specific code

Incoming request:	File to open:
`GET /blog`	`blog/index.html`
`GET /blog/`	`blog/index.html`

Incoming request:	File to open:
`GET /blog`	`blog/index.html`
`GET /blog/`	`blog/index.html`
`GET /blog.png`	`blog.png`
`GET /blog/logo.png`	`blog/logo.png`

```
      number      number
if (extension == "png") {
    // When will this code run?
}
```

```
                          strcmp(extension, "png")
```

```
switch (extension) {
    case "png":
        content_type = "image/png";
        break;
}
```

```
if (!strcmp(extension, "png")) {
    content_type = "image/png";
}
```

```
if (!strcmp(extension, "png")) {
    content_type = "image/png";
} else if (!strcmp(extension, "css")) {
    content_type = "text/css";
}
```

```
if (!strcmp(extension, "png")) {
    content_type = "image/png";
} else if (!strcmp(extension, "css")) {
    content_type = "text/css";
} else if (!strcmp(extension, "js")) {
    content_type = "application/javascript";
}
```

```
if (!strcmp(extension, "png")) {
    content_type = "image/png";
} else if (!strcmp(extension, "css")) {
    content_type = "text/css";
} else if (!strcmp(extension, "js")) {
    content_type = "application/javascript";
} else if (!strcmp(extension, "html")) {
    content_type = "text/html";
}
```

01110000	01101110	01100111
'p' 112	'n' 110	'g' 103

```
"png"
"css"
"js"
"html"
```

1,752,460,652

01101000011101000110110101101100

01101000	01110100	01101101	01101100
'h' 104	't' 116	'm' 109	'l' 108

"png\0" 1,886,283,520

"css"

"js"

"html" 1,752,460,652


```
"png\0"                                  1,886,283,520
"css\0"
"js"
"html"                                   1,752,460,652
```


```
"png\0"                                  1,886,283,520
"css\0"
"js\0\0"
"html"                                   1,752,460,652
```

```
          number
switch (extension) {
    case "png":
        type = "image/png";
        break;
}
```

```
const int PNG = 1886283520;

          number
switch (extension) {
    case PNG:
        type = "image/png";
        break;
}
```

```
const int PNG = 1886283520;
```

“**cast**—that is, treat `extension` as an `int`'s address”

```
switch (*(int *)extension) {
    case PNG:
        type = "image/png";
        break;
}
```

```
const int PNG = 1886283520;
```

“get whatever 4-byte `int` is at that address”

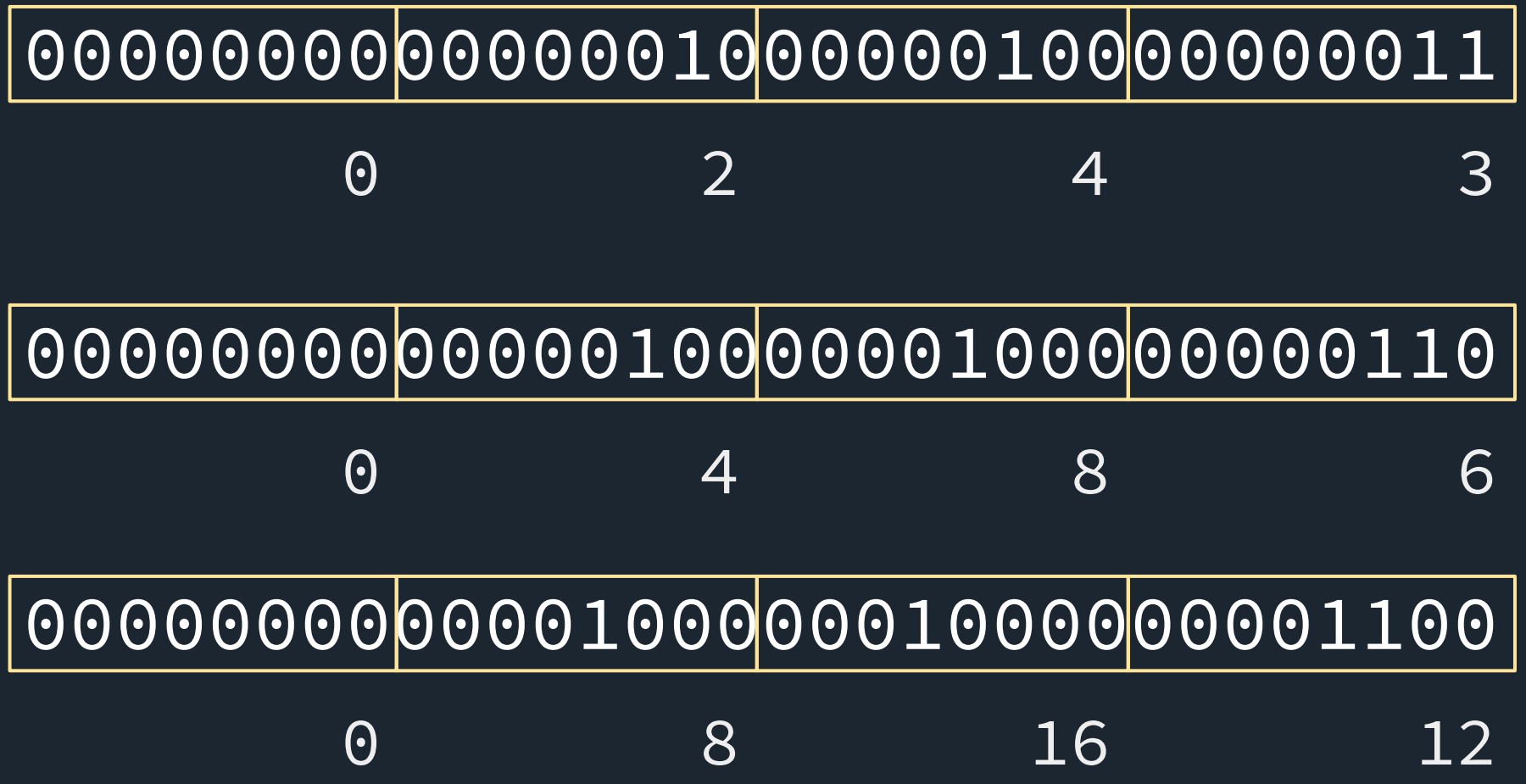
```
switch (*(int *)extension) {
    case PNG:
        type = "image/png";
        break;
}
```

```
const int PNG = 1886283520;
const int HTML = 1752460652;

switch (*(int *)extension) {
    case PNG:
        type = "image/png";
        break;
    case HTML:
        type = "text/html";
        break;
}
```

```
const int PNG = 1886283520;
const int HTML = 1752460652;

switch (*(int *)extension) {
    case PNG:
        type = "image/png";
        break;
    case HTML:
        type = "text/html";
        break;
    default:
        type = "application/octet-stream";
}
```

varies by **endianness**
(for this workshop,
don't worry about it)

Bit Shifts


00000000 00000010 00000100 00000011
0 2 4 3
00000000 00000100 00001000 00000110
0 4 8 6
00000000 00001000 00010000 00001100
0 8 16 12

Bit Shifts

```
'h' | ('t' << 8) | ('m' << 16) | ('l' << 24)
```

"bitwise OR"

Bit Shifts

```
'h' | ('t' << 8) | ('m' << 16) | ('l' << 24)
```

```
to_int('h', 't', 'm', 'l')
to_int('c', 's', 's', '\0')
to_int('j', 's', '\0', '\0')
```

```
const int PNG = 1886283520;
const int HTML = 1752460652;
```

Bit Shifts

```
'h' | ('t' << 8) | ('m' << 16) | ('l' << 24)
```

```
to_int('h', 't', 'm', 'l')
to_int('c', 's', 's', '\0')
to_int('j', 's', '\0', '\0')
```

```
const int PNG = to_int('p', 'n', 'g', '\0');
const int HTML = to_int('h', 't', 'm', 'l');
```

compile-time constant

```
const int PORT 8080;
printf("Listening on localhost:%d\n", PORT);
```

```
#define PORT 8080;
printf("Listening on localhost:%d\n", PORT);
```

```
#define PORT 8080;
printf("Listening on localhost:%d\n", PORT);

#define FOURCHAR(a, b, c, d) \
    a | (b << 8) | (c << 16) | (d << 24)
```

```c
#define PORT 8080;
printf("Listening on localhost:%d\n", PORT);

#define FOURCHAR(a, b, c, d) \
    a | (b << 8) | (c << 16) | (d << 24)

const int HTML = FOURCHAR('h', 't', 'm', 'l');
const int CSS = FOURCHAR('c', 's', 's', '\0');
const int JS = FOURCHAR('j', 's', '\0', '\0');
```

```
const int PNG = FOURCHAR('p', 'n', 'g', '\0');
const int HTML = FOURCHAR('h', 't', 'm', 'l');

switch (*(int *)extension) {
    case PNG:
        type = "image/png";
        break;
    case HTML:
        type = "text/html";
        break;
    default:
        type = "application/octet-stream";
}
```

this is what most languages compile to!

```
if (!strcmp(extension, "png")) {
    content_type = "image/png";
} else if (!strcmp(extension, "css")) {
    content_type = "text/css";
} else if (!strcmp(extension, "js")) {
    content_type = "application/javascript";
} else if (!strcmp(extension, "html")) {
    content_type = "text/html";
}
```

```
const int PNG = FOURCHAR('p', 'n', 'g', '\0');
const int HTML = FOURCHAR('h', 't', 'm', 'l');

switch (*(int *)extension) {
    case PNG:
        type = "image/png";
        break;
    case HTML:
        type = "text/html";
        break;
    default:
        type = "application/octet-stream";
}
```

This optimization relies on:

1. Knowing extensions fit in 4 bytes
2. Knowing memory is just 1s and 0s
3. Using a language that supports this

The `sendfile()` function

We **`read()`** from a file descriptor…

…then **`write()`** to another fd…

…can we avoid this handoff work?

`sendfile()` takes a src and dest fd

```
ssize_t sendfile(
    int out_fd,
    int in_fd,
    off_t *offset,
    size_t count
)
```

```
int sendfile(
    int fd,
    int s,
    off_t offset,
    off_t *len,
    sf_hdtr *hdtr,
    int flags
)
```

sendfile(2) — Linux manual page

sendfile(2) **System Calls Manual** **sendfile(2)**

NAME top

sendfile - transfer data between file descriptors

LIBRARY top

Standard C library (*libc*, *-lc*)

SYNOPSIS top

```
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *_Nullable offset,
                 size_t count);
```

SENDFILE(2) BSD System Calls Manual SENDFILE(2)

NAME

sendfile -- send a file to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>

int
sendfile(int fd, int s, off_t offset, off_t *len, struct sf_hdtr *hdtr,
    int flags);
```

DESCRIPTION

The **sendfile**() system call sends a regular file specified by descriptor *fd* out a stream socket specified by descriptor *s*.

```
ssize_t sendfile(
    int out_fd,
    int in_fd,
    off_t *offset,
    size_t count
)
```

```
int sendfile(
    int fd,
    int s,
    off_t offset,
    off_t *len,
    sf_hdtr *hdtr,
    int flags
)
```

What happened to “**portable assembly**?!”

Target-Specific Preprocessor Macros

```
#ifdef __linux__
#include <sys/sendfile.h>
#endif
```

Target-Specific Preprocessor Macros

```
#ifdef __linux__
    // Linux-specific code
#elif defined(__APPLE__)
    // macOS-specific code
#else
    #error "Unsupported OS"
#endif
```

Target-Specific Preprocessor Macros

```
#ifdef __linux__
    sendfile(/* args Linux expects */);
#elif defined(__APPLE__)
    sendfile(/* args macOS expects */);
#else
    #error "Unsupported OS"
#endif
```

Summary of Part 6

Bit Shifts

Switches

Preprocessor Macros

Platform-specific code

Wrap-Up

Course Summary

Further Resources

memory is basically a **gigantic array of bytes**

```
                start index                 length
write(1, "Hello, World!", 13);
         547362942671452
```

“Pointer” means “Memory Address”

(“Monads are just monoids in the category of endofunctors, what's the problem?”)

What is C?


TOP
10


Microsoft
Windows


PostgreSQL


MySQL
redis


UNIX


python


php

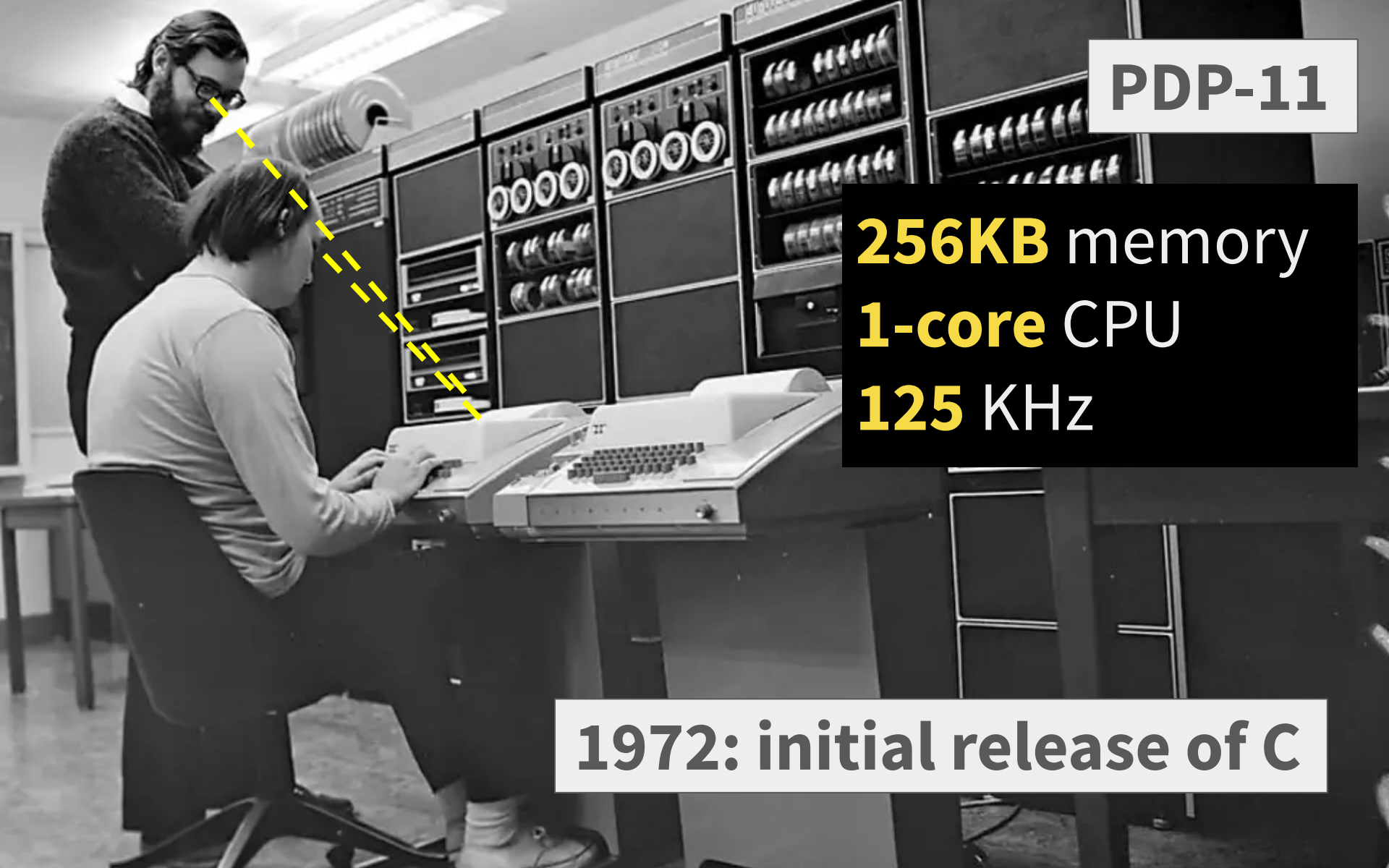

Ruby


android


Linux


mac
OS


iOS


PDP-11
256KB memory
1-core CPU
125 KHz
1972: initial release of C

Why is C still popular?

Maximum possible performance

thanks to Zero-overhead programming

Much more ergonomic than Assembly

Much simpler than C++

Why is C still popular?

Local Static HTTP Server Perf Olympics

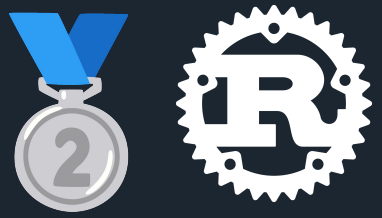
Time
9 ms
5 ms
3 ms
4 ms
5 ms
3 ms
5 ms
2 ms

http-server

Weekly Downloads

2,360,975

Time
6 ms
1 ms
4 ms
3 ms
2 ms
2 ms
2 ms
2 ms



simple-http-server

52,089

Downloads all time

Time
2 ms
3 ms
2 ms
2 ms
1 ms
3 ms
2 ms
1 ms

the ~300 lines of C code we write in this workshop using only the stdlib

What is C?

“Portable assembly”

Zero-overhead programming

Almost zero-safety programming

The language other languages use to talk to each other

nodejs.org/api/addons.html

C++ addons

Addons are dynamically-linked shared objects written in C++. The `require()` function can load addons as ordinary Node.js modules. Addons provide an interface between JavaScript and C/C++ libraries.

There are three options for implementing addons:

- Node-API
- `nan` (Native Abstractions for Node.js)
- direct use of internal V8, libuv, and Node.js libraries

Unless there is a need for direct access to functionality which is not exposed by Node-API, use Node-API. Refer to C/C++ addons with Node-API for more information on Node-API.

When not using Node-API, implementing addons becomes more complex, requiring knowledge of multiple components and APIs:

- V8: the C++ library Node.js uses to provide the JavaScript implementation. It provides the mechanisms for creating objects, calling functions, etc. The V8's API is documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node.js source tree), and is also available online.

- libuv: The C library that implements the Node.js event loop, its worker threads and all of the

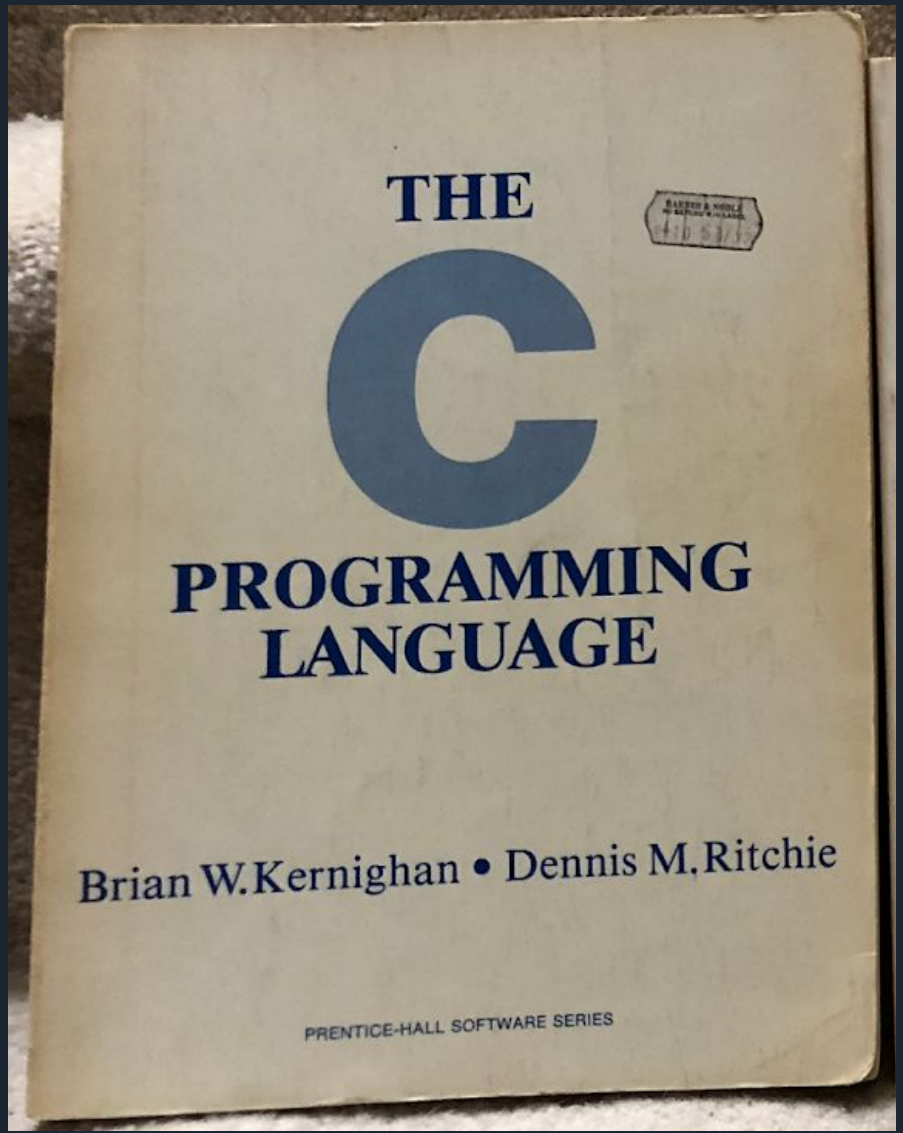
Buiding Large C(++) Projects

“How to Build Software From Source”

Andrew Kelley (Zig language creator)

2023 *Software You Can Love*

youtu.be/CwXixVcliP0



Brian **K**ernighan

Dennis **R**itchie

“**K**&**R**”

Software optimization resources

See also my blog

Contents

agner.org/optimize

Optimization manuals

This series of five manuals describes everything you need to know about optimizing code for x86 and x86-64 family microprocessors, including optimization advices for C++ and assembly language, details about the microarchitecture and instruction timings of most Intel, AMD and VIA processors, and details about different compilers and calling conventions.

Operating systems covered: DOS, Windows, Linux, BSD, Mac OS X Intel based, 32 and 64 bits.

Note that these manuals are not for beginners.

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms

This is an optimization manual for advanced C++ programmers. Topics include: The choice of platform and operating system. Choice of compiler and framework. Finding performance bottlenecks. The efficiency of different C++ constructs. Multi-core systems. Parallelization with vector operations. CPU dispatching. Efficient container class templates. Etc.

File name: optimizing_cpp.pdf, size: 1798079, last modified: 2024-Mar-15.
Download.

Performance-Aware Programming

computerenhance.com

Low-overhead C alternatives

JAI

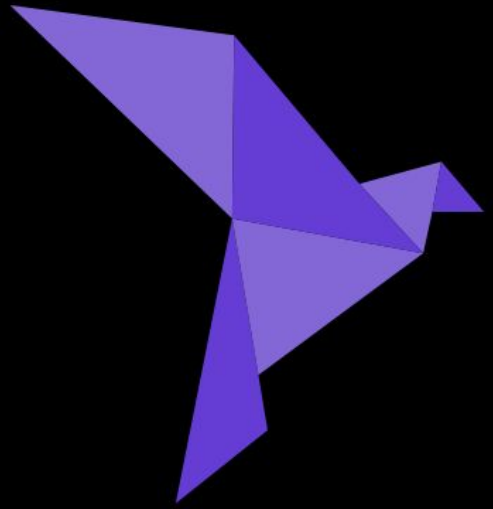
Low-overhead C alternatives

all support C interop

because C is **the** language that

other languages use to talk to each other

C is the most **universal** language!



I host a podcast!

roc-lang.org

software-unscripted.com