


# Linux Plumbers Conference

Dublin, Ireland September 12-14, 2022

A decorative graphic of a green pipe network with various fittings, valves, and elbows, set against a white background. The pipes are arranged in a complex, interconnected pattern, with some sections being vertical and others horizontal or diagonal.

# How I started chasing speculative type confusion bugs in the kernel and ended up with 'real' ones

Jakob Koschel


PhD student @ Vrije Universiteit Amsterdam



Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



VUsec



LWN: "The Linux kernel could upgrade from C89 to C11 as early as the 5.18 release"

"Alright, just gonna to patch a few of these Spectre gadgets that we found"



Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of a green pipe network with various fittings, elbows, and valves, framing the text on the slide.

We'll start with some **Spectre**  
background

followed by an **interesting case study**

revealing more 'real' bugs in the **list  
iterators**



Linux

Plumbers Conference | Dublin, Ireland **Sept. 12-14, 2022**

# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```

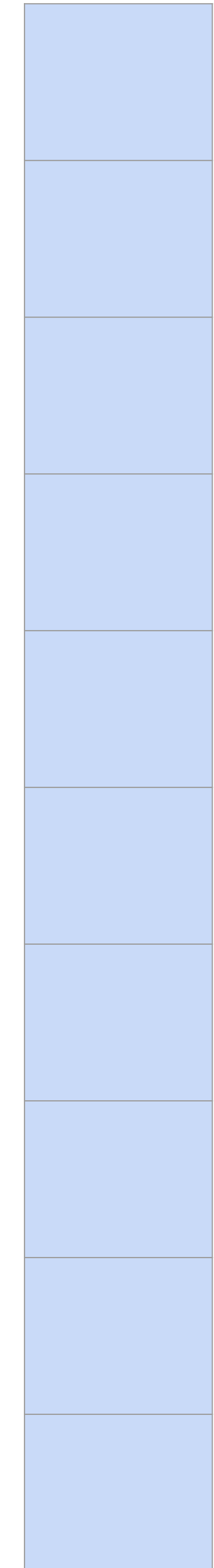
**i = 0**

# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```

$i = 0$

Cache

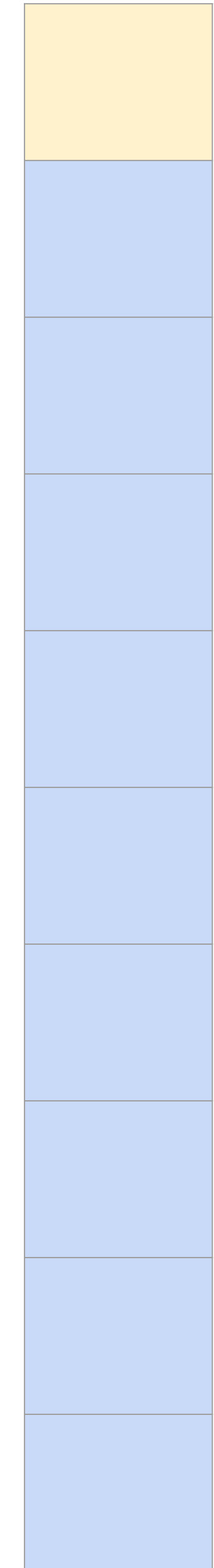


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

`i = 0`

Cache

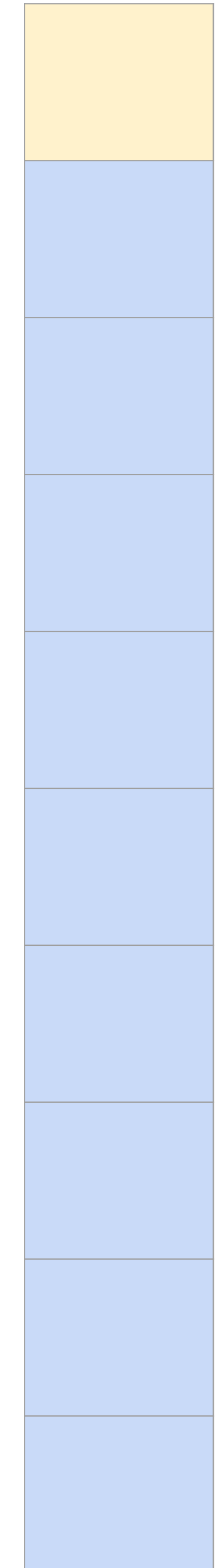


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```

`i = 1`

Cache



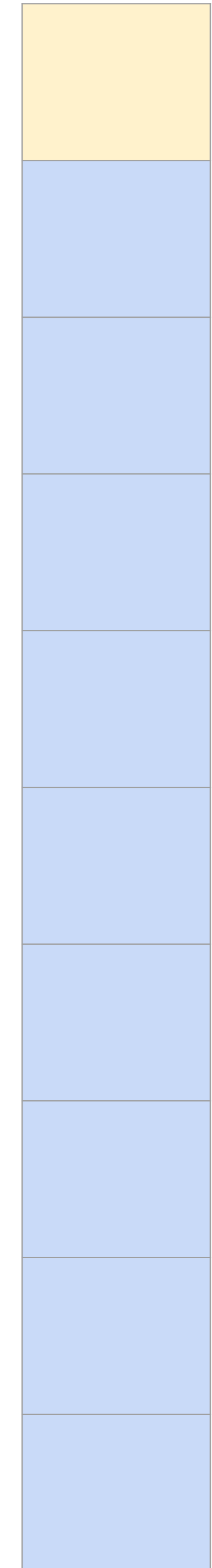


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

`i = 2`

Cache



# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

**i = 3**

Cache

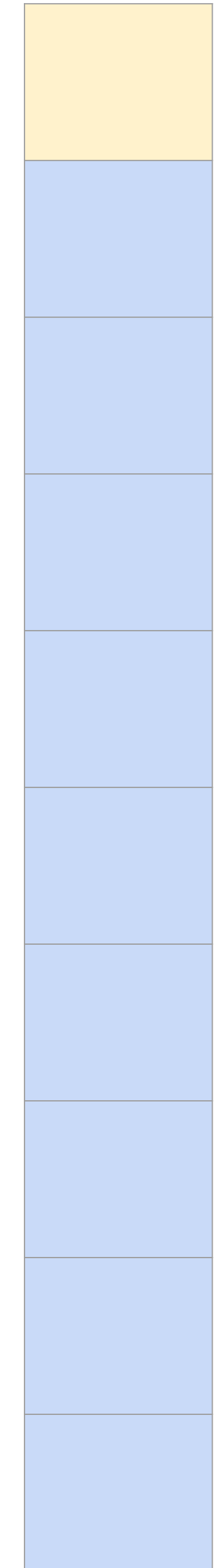


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 128; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

**i = 4**

Cache

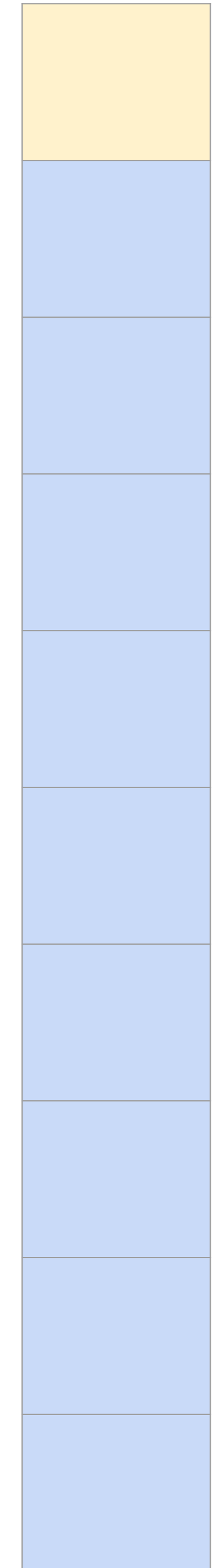


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```

$i = 5$

Cache

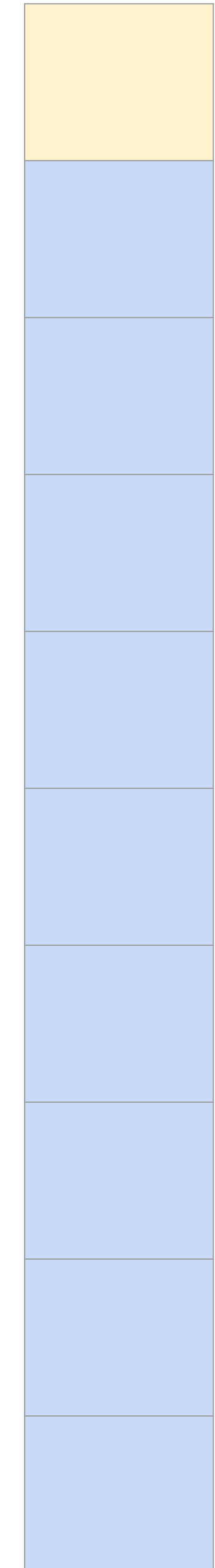


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```

**i = 6**

Cache

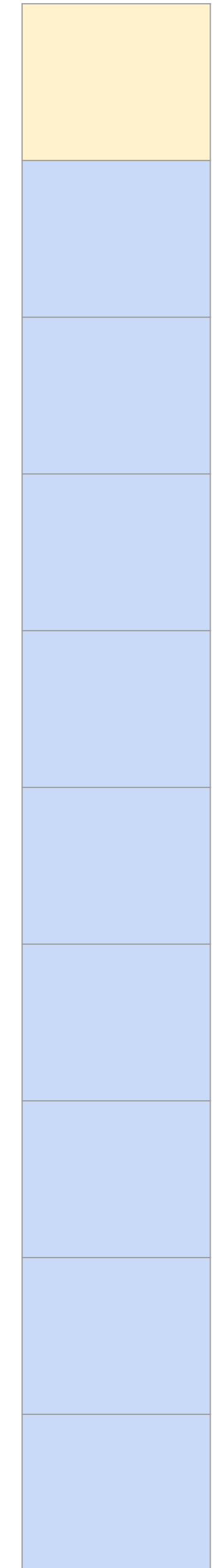


# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```

**i = 63**

Cache



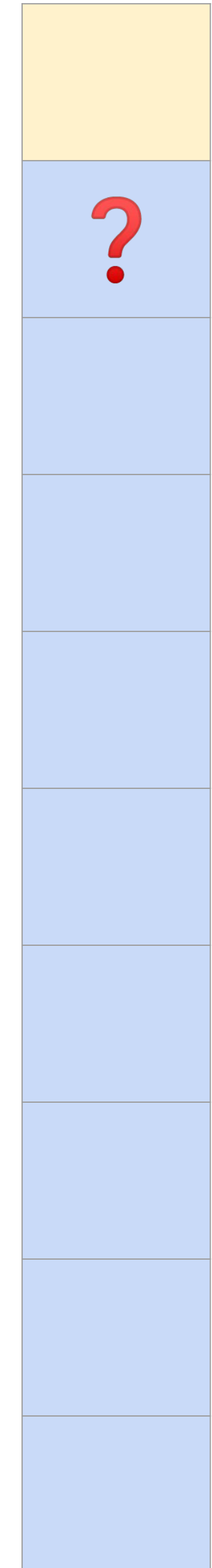
# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```



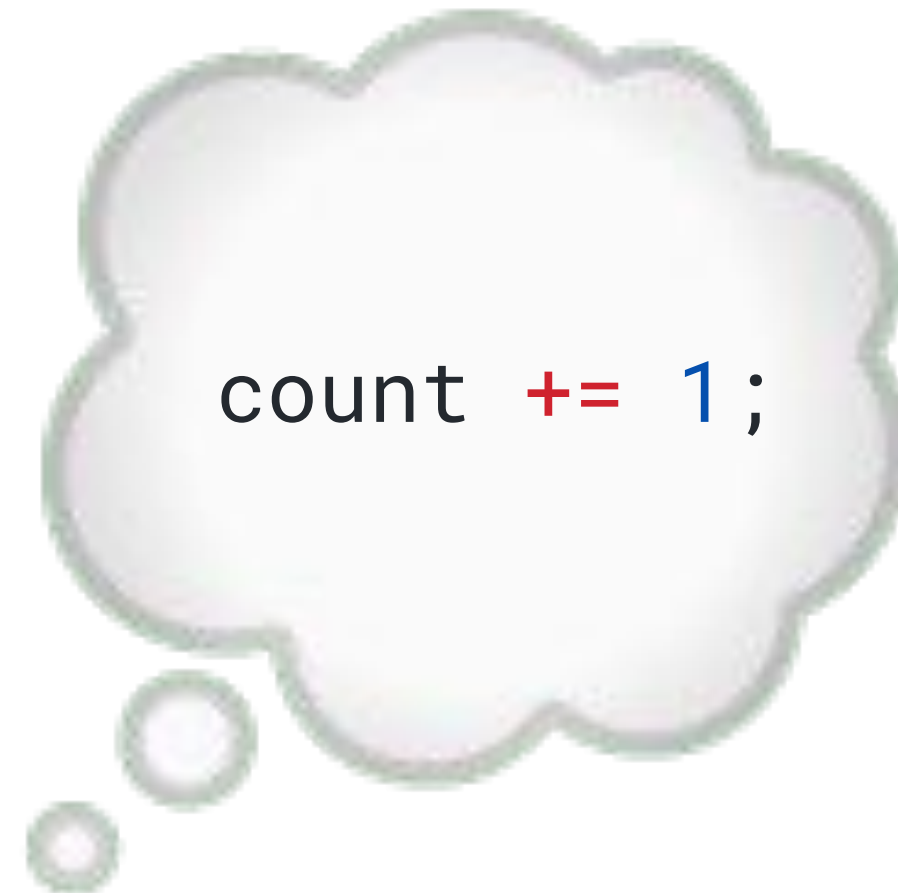
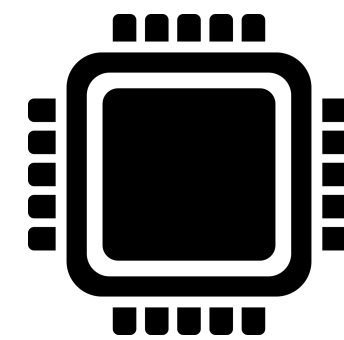
`i = 64`

Cache



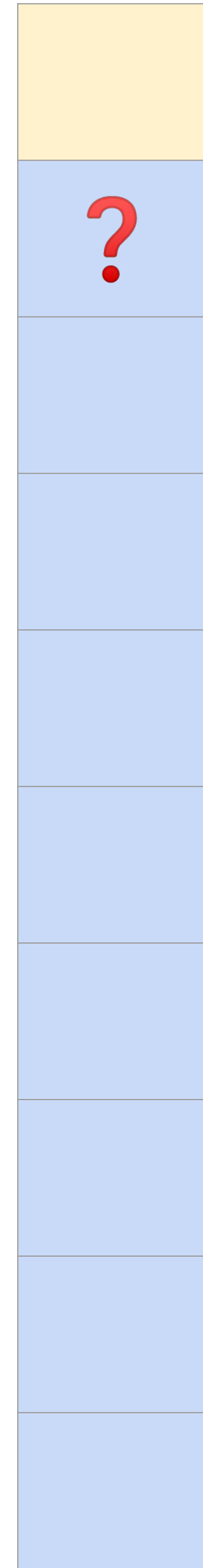
# Speculative Execution & Branch Predictor

```
char msg[128] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 128; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```



i = 64

Cache





A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

But what if the CPU is not right?

Can we fool the **Branch  
Predictor?**



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# Misprediction

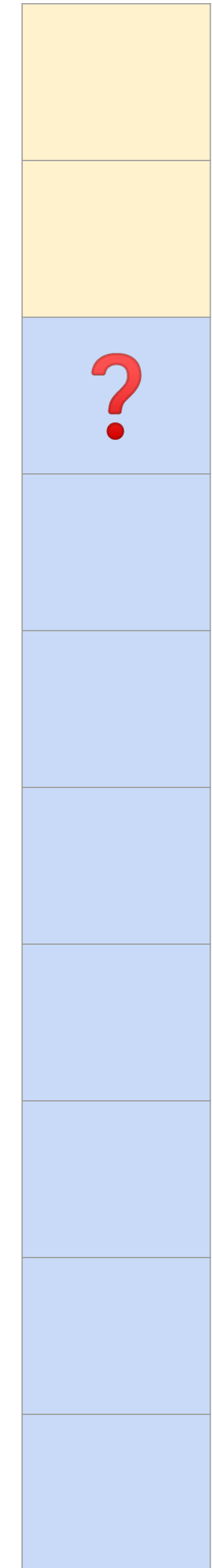
```
char msg[129] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 129; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

# Misprediction

```
char msg[129] = "LPCLPCLPCLPCLPC...\0";
int count = 0;
// calculate length of string
for (int i = 0; i < 129; i++) {
    if (msg[i] != '\0') {
        count += 1;
    } else {
        break;
    }
}
```

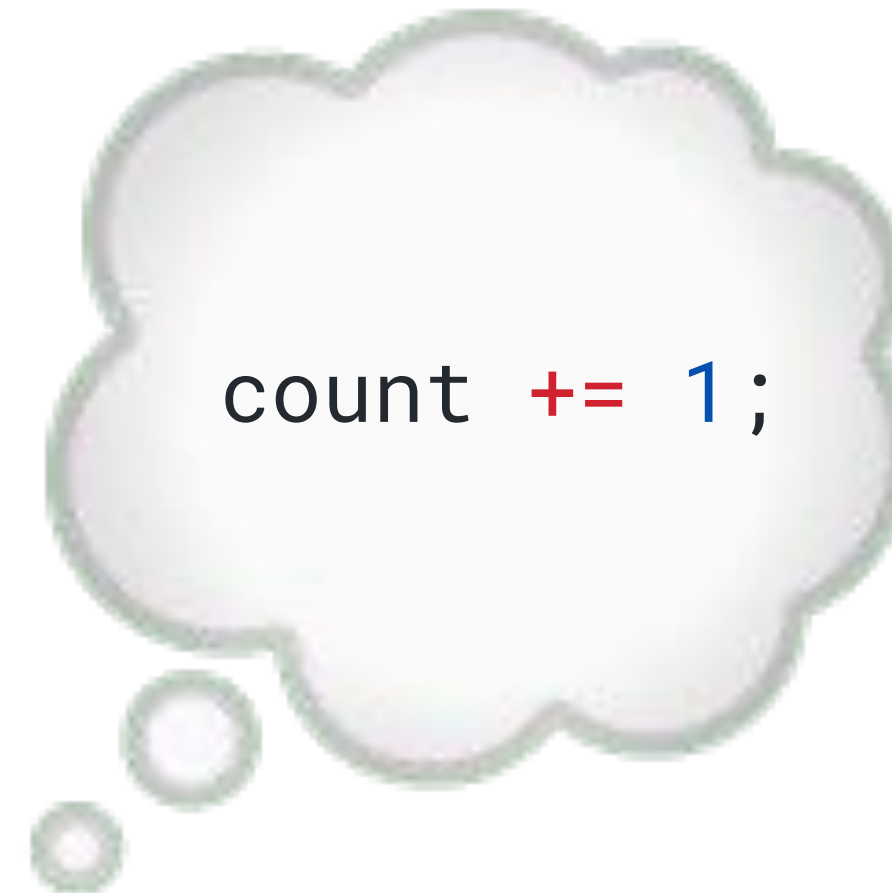
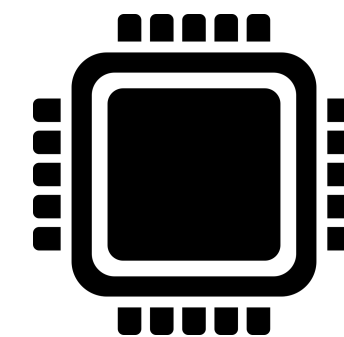
`i = 129`

Cache



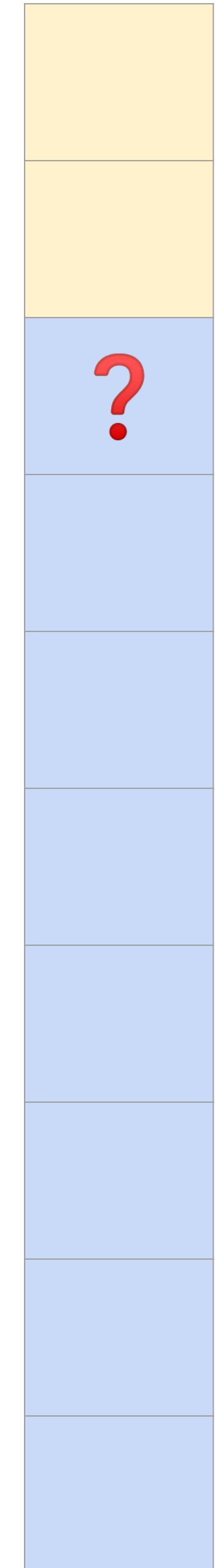
# Misprediction

```
char msg[129] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 129; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```



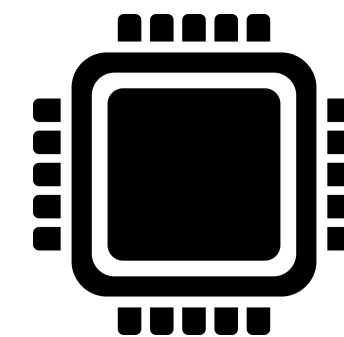
`i = 129`

Cache



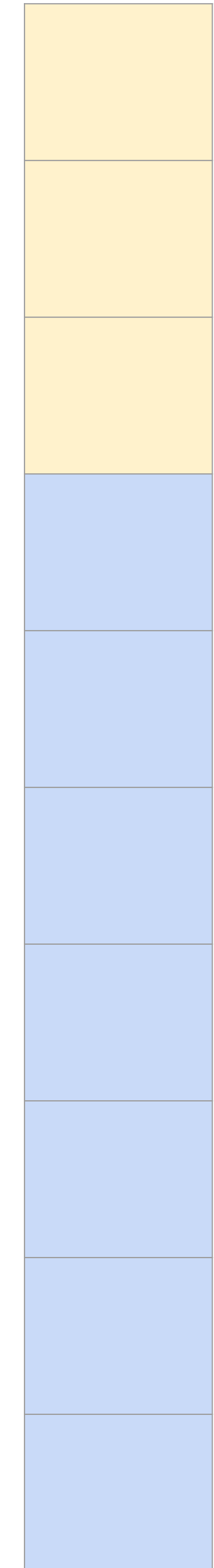
# Misprediction

```
char msg[129] = "LPCLPCLPCLPCLPC...\0";  
int count = 0;  
// calculate length of string  
for (int i = 0; i < 129; i++) {  
    if (msg[i] != '\0') {  
        count += 1;  
    } else {  
        break;  
    }  
}
```



i = 129

Cache



# A Spectre V1 gadget

```
x = get_user(ptr);  
if (x < size) {  
    y = arr1[x];  
    z = arr2[y];  
}
```


# A Spectre V1 gadget



```
x = get_user(ptr);  
if (x < size) {  
    y = arr1[x];  
    z = arr2[y];  
}
```

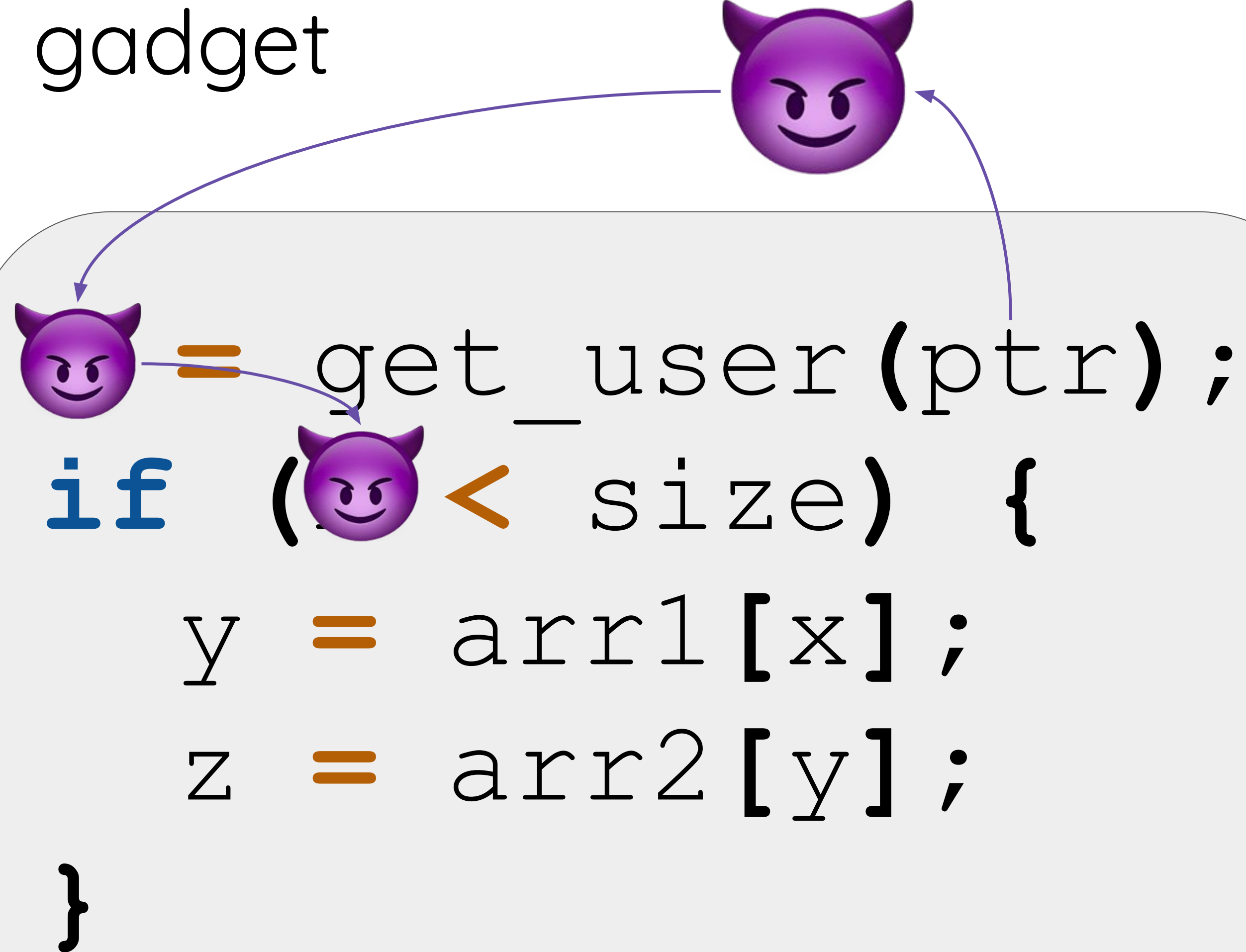
# A Spectre V1 gadget



```
 = get_user(ptr);  
if (x < size) {  
    y = arr1[x];  
    z = arr2[y];  
}
```



# A Spectre V1 gadget

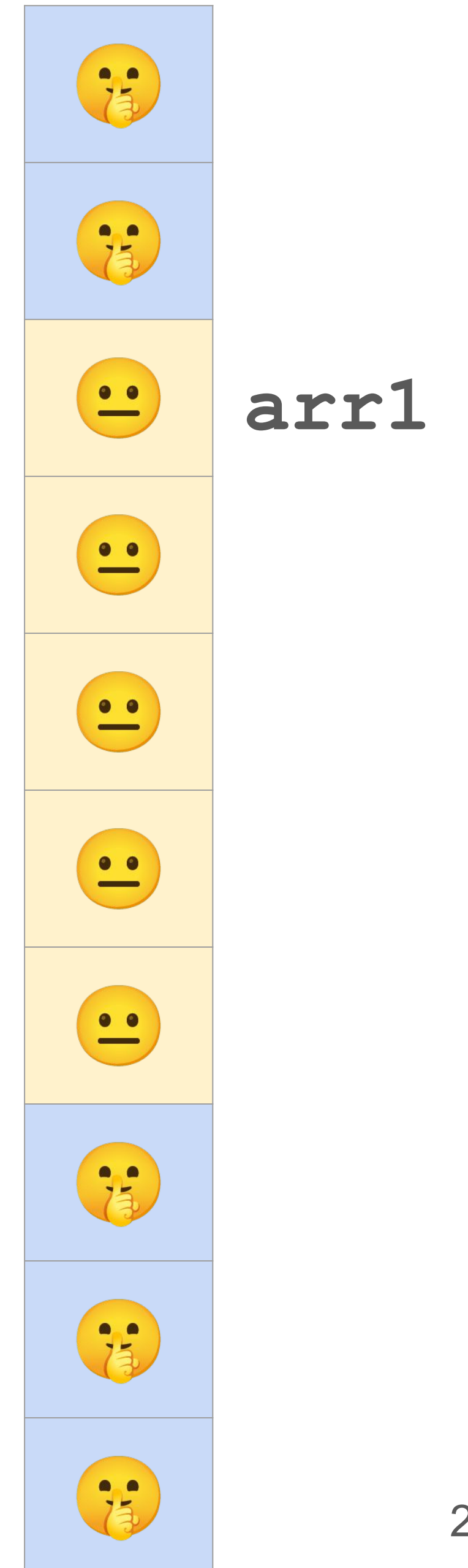


```
user = get_user(ptr);  
if (user < size) {  
    y = arr1[x];  
    z = arr2[y];  
}
```

# A Spectre V1 gadget

```
👹 = get_user(ptr);  
if (👹 < size) {  
    y = arr1[x];  
    z = arr2[y];  
}
```

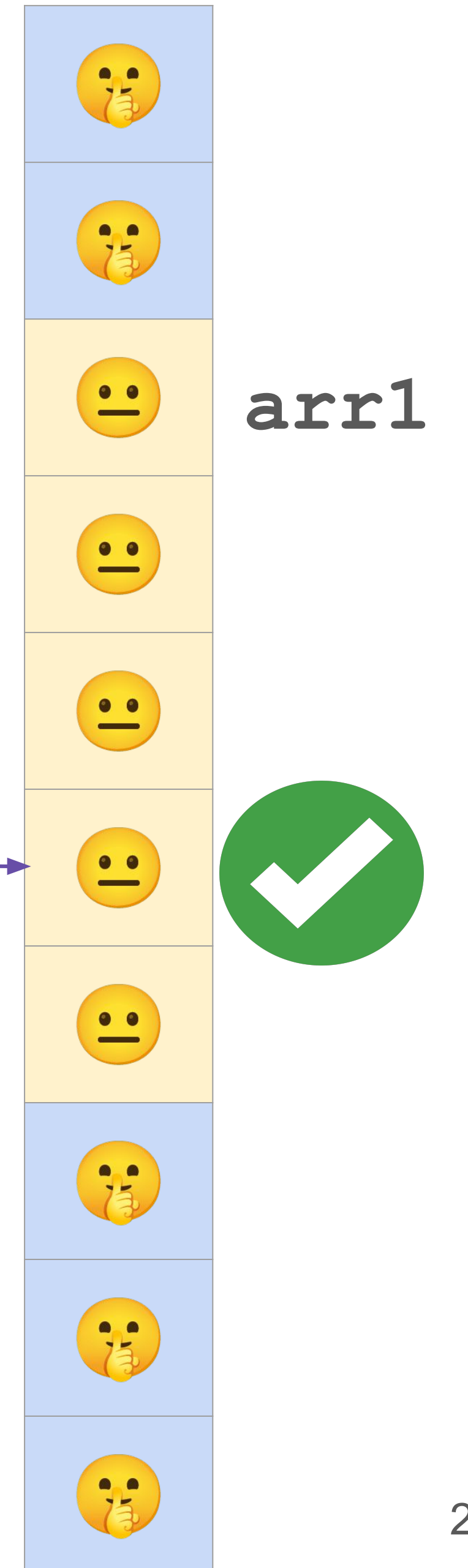
Kernel memory



# A Spectre V1 gadget

```
    🍆 = get_user(ptr);  
    if (🍆 < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }
```

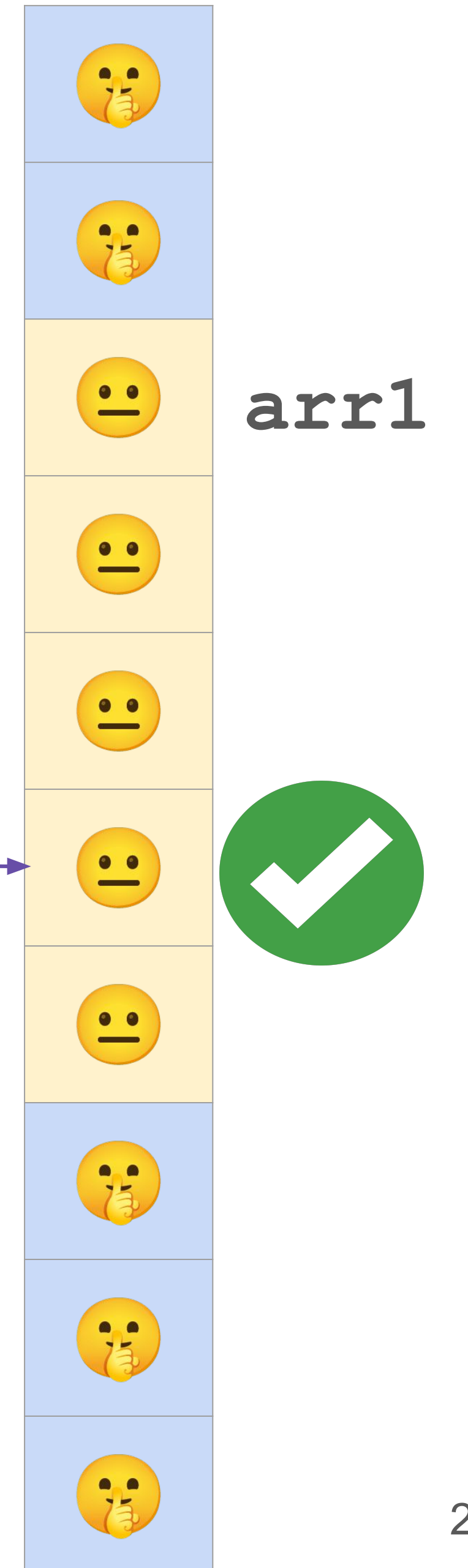
Kernel memory



# A Spectre V1 gadget

```
    🍆 = get_user(ptr);  
    if (🍆 < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }
```

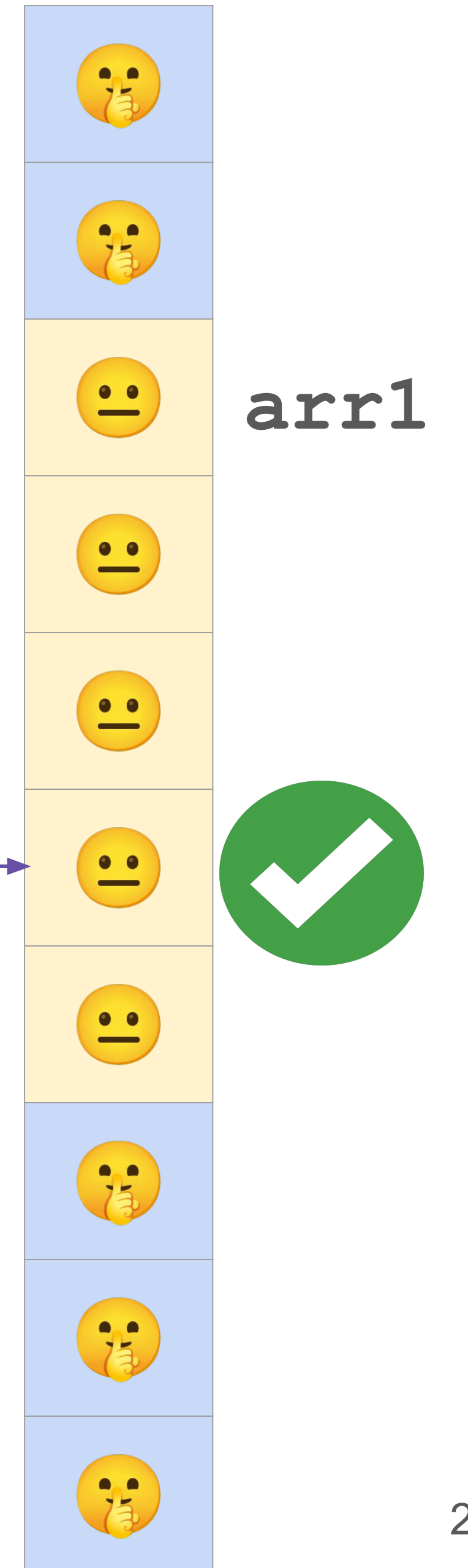
Kernel memory



# A Spectre V1 gadget

```
    = get_user(ptr);  
if ( < size) {  
    y = arr1 [ ];  
    z = arr2 [y];  
}
```

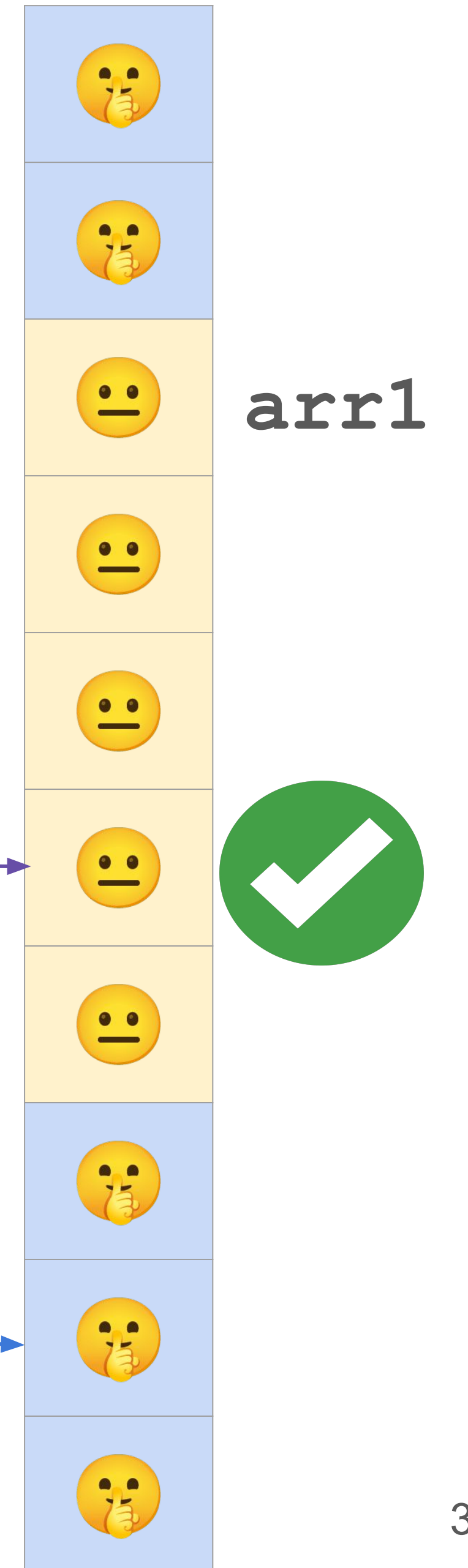
Kernel memory



# A Spectre V1 gadget

```
    = get_user(ptr);  
    if ( < size) {  
        y = arr1[ ];  
        z = arr2[y];  
    }
```

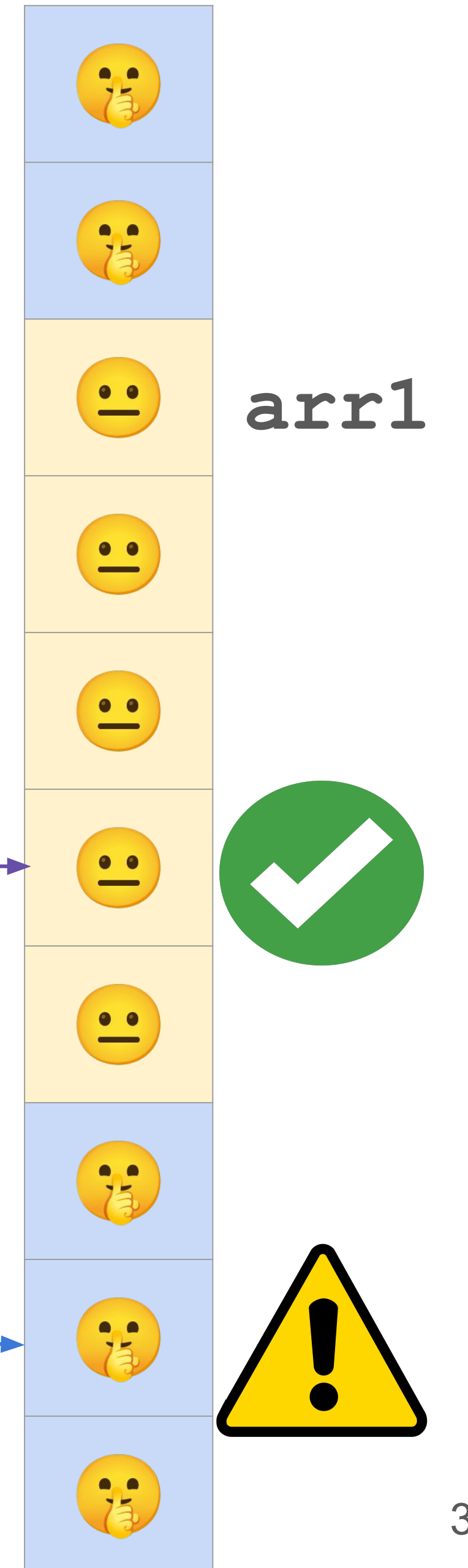
Kernel memory



# A Spectre V1 gadget

```
    = get_user(ptr);  
    if ( < size) {  
        y = arr1 [ ];  
        z = arr2[y];  
    }
```

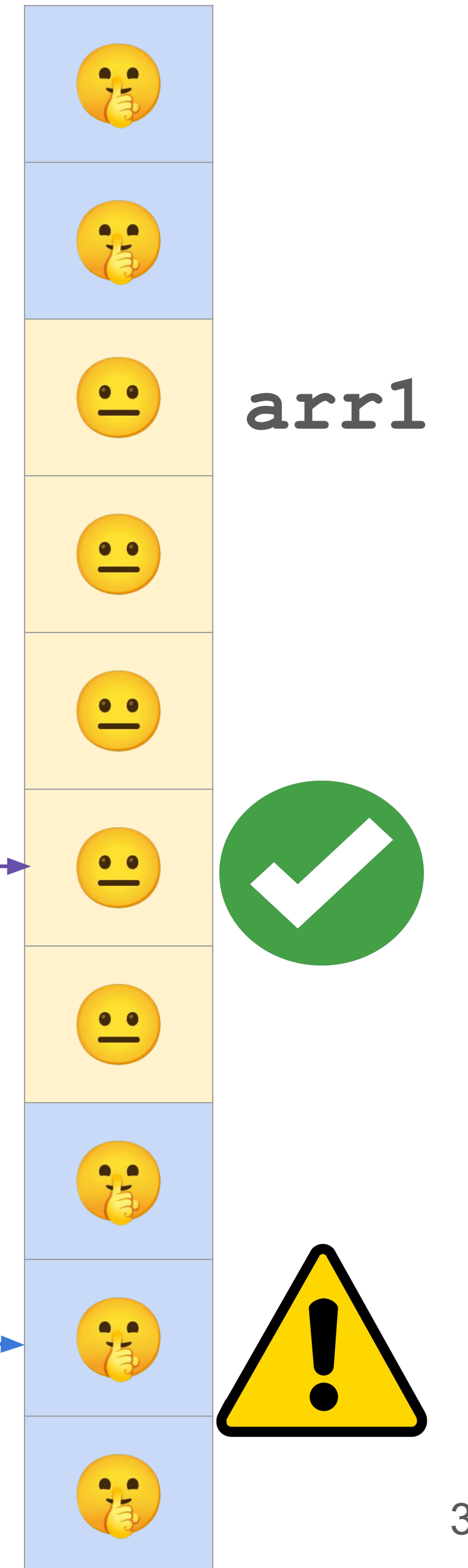
Kernel memory



# A Spectre V1 gadget

```
    🐱 = get_user(ptr);  
    if (🐱 < size) {  
        🤫 = arr1[👻];  
        z = arr2[y];  
    }
```

Kernel memory

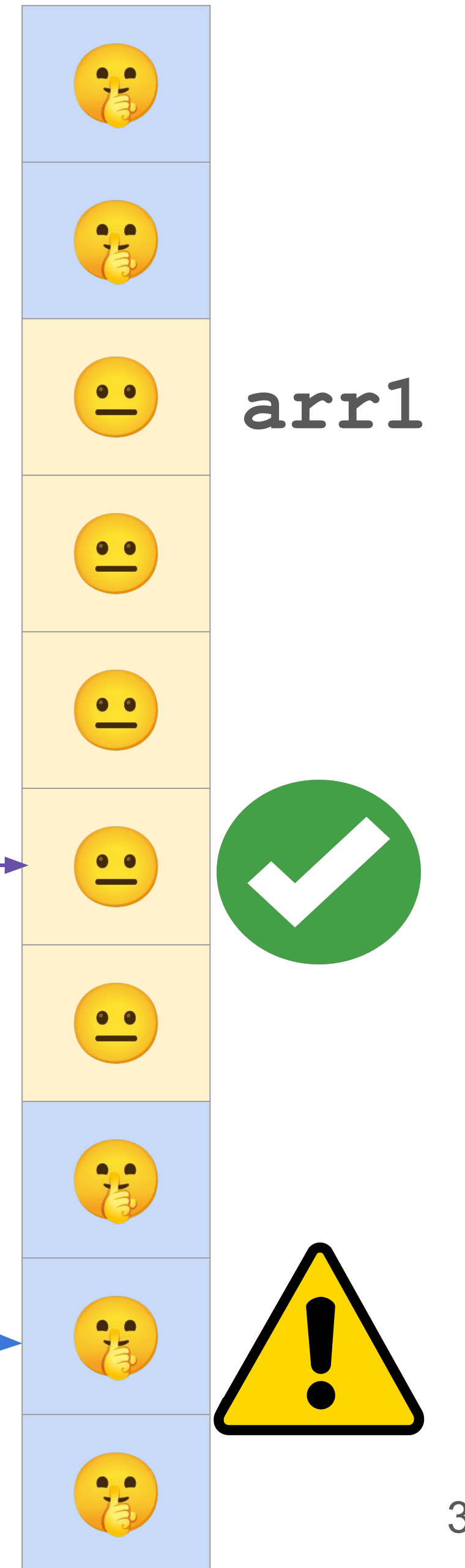




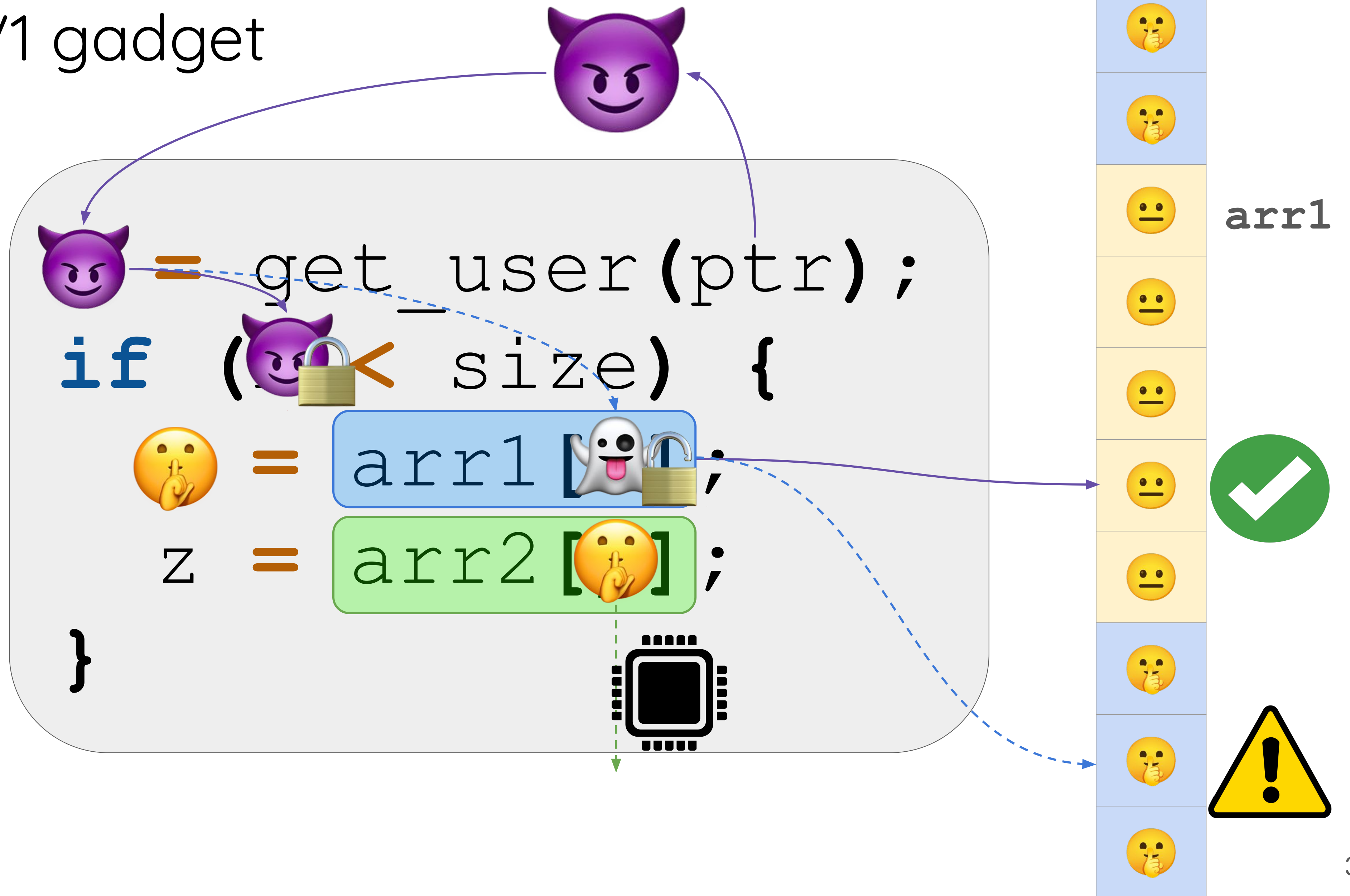
# A Spectre V1 gadget

```
    🐱 = get_user(ptr);  
    if (🐱 < size) {  
        🧠 = arr1[👻];  
        z = arr2[🧠];  
    }
```

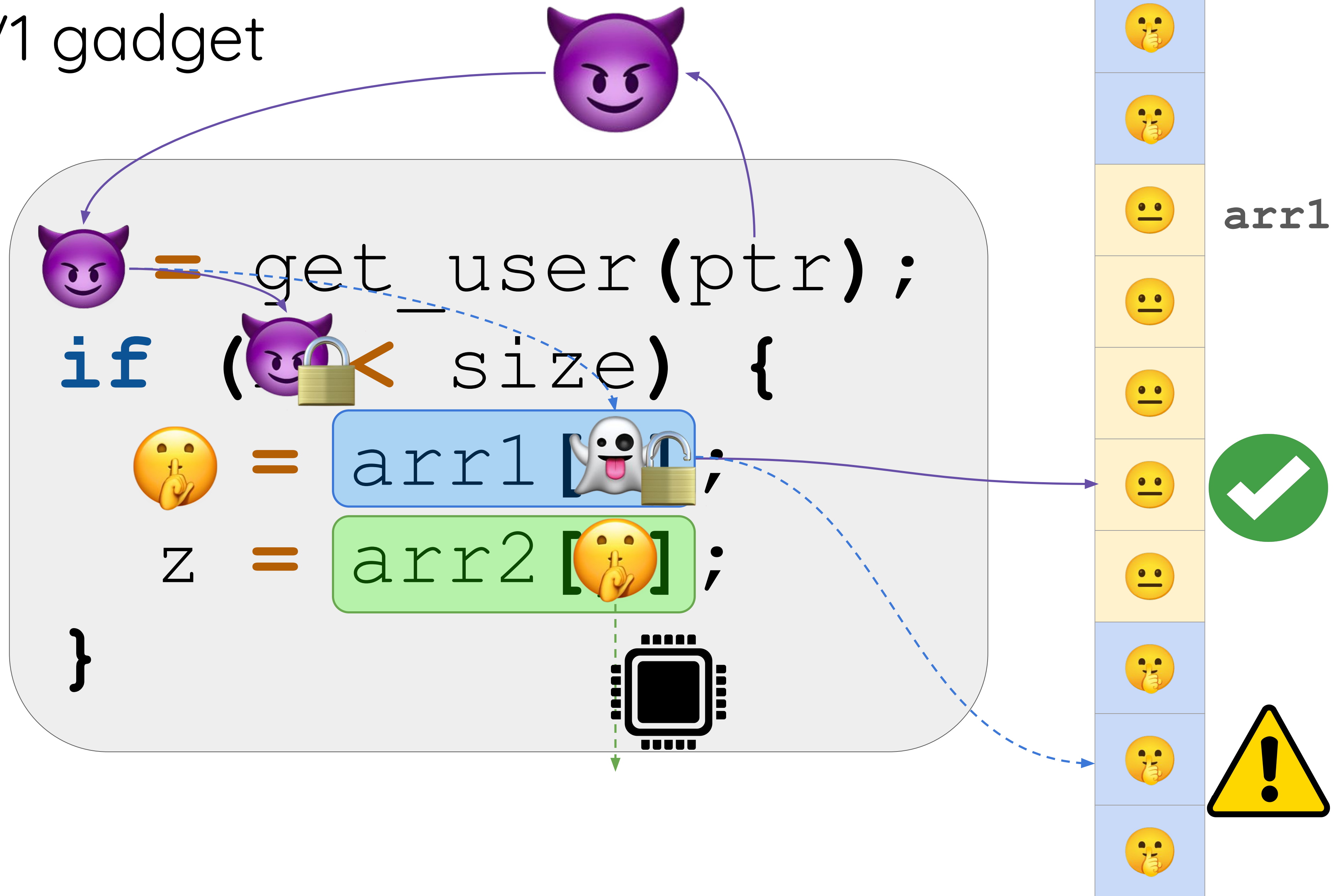
Kernel memory



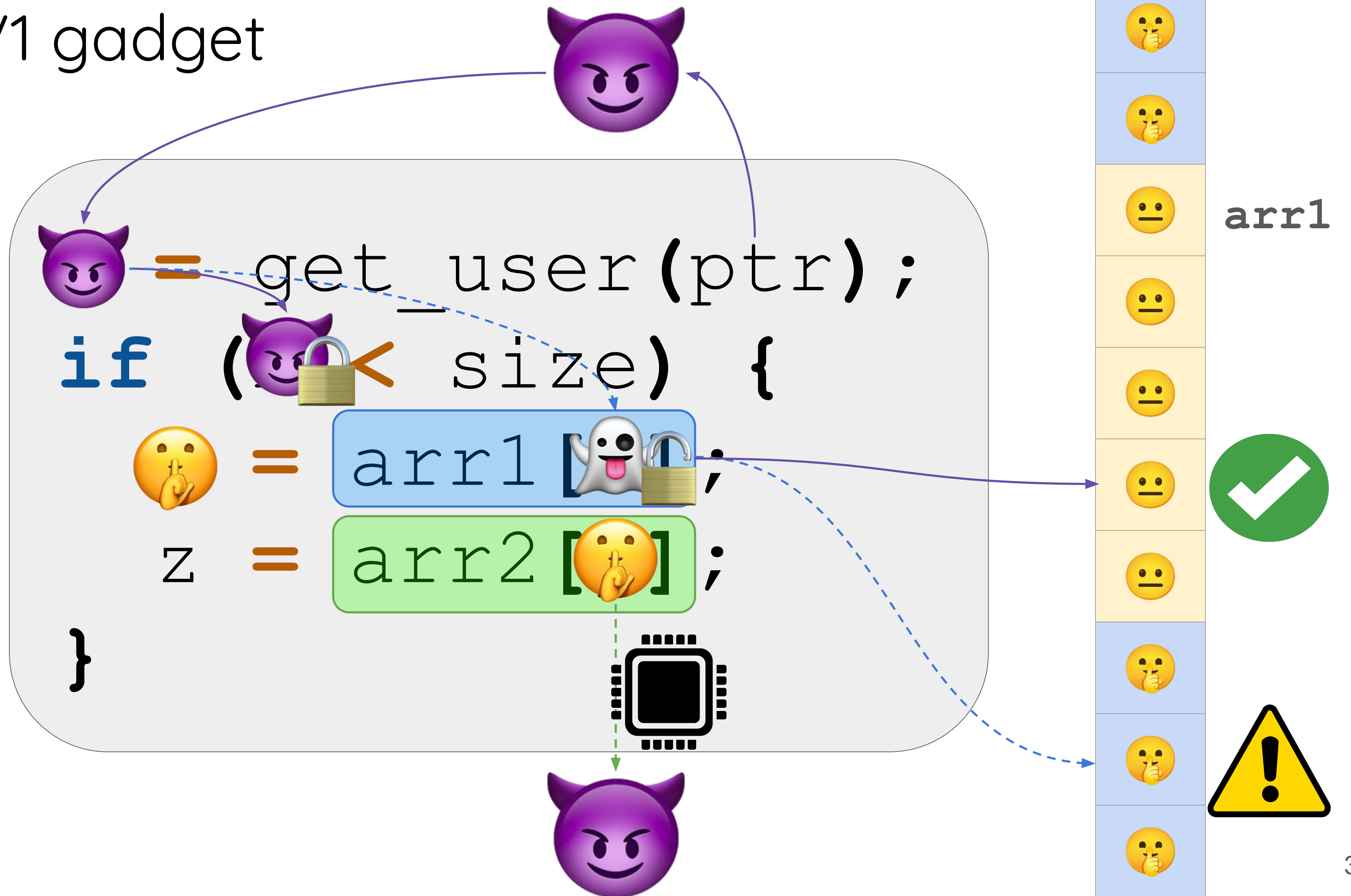
# A Spectre V1 gadget

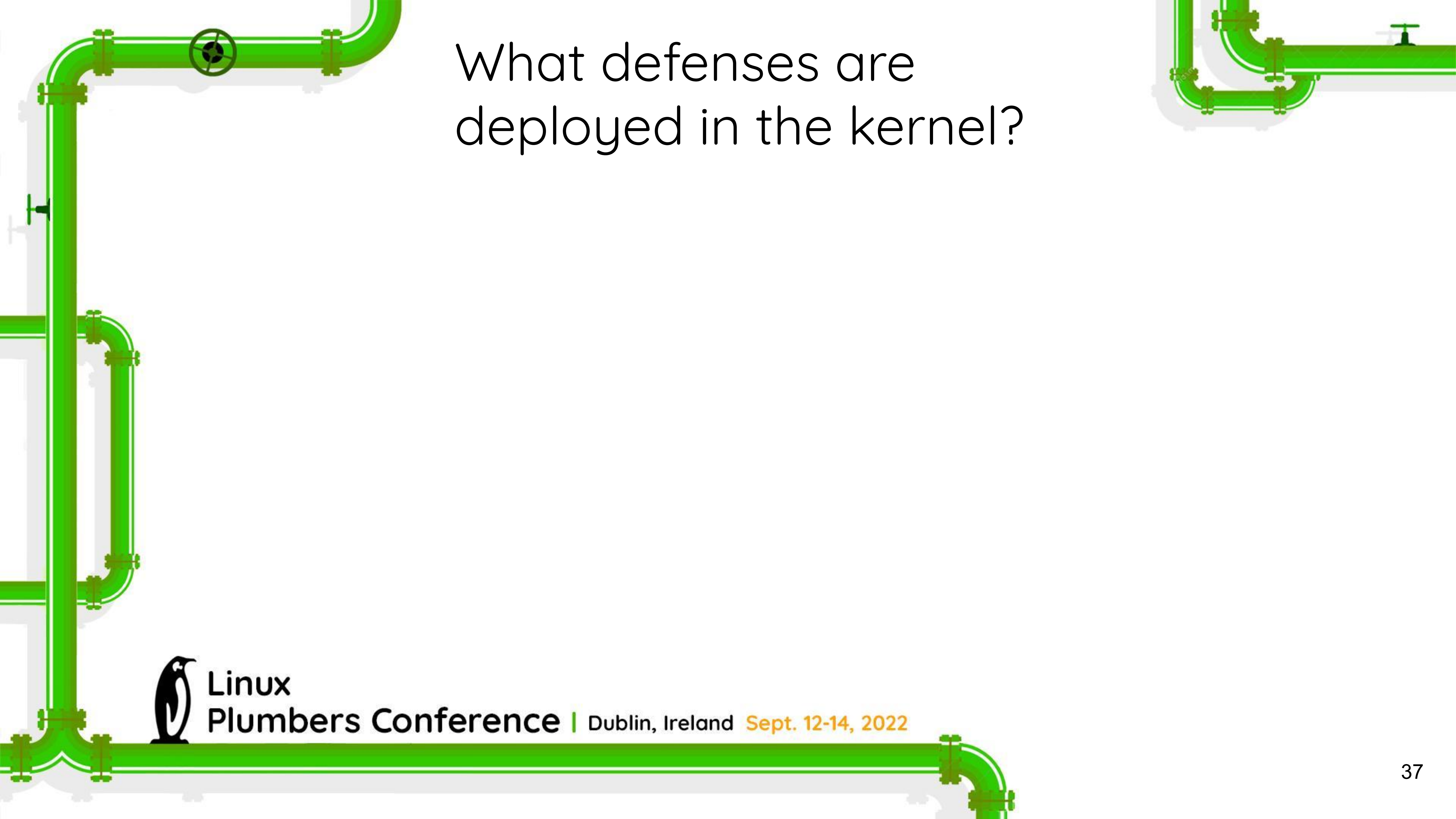


# A Spectre V1 gadget



# A Spectre V1 gadget



A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text. The pipes are bright green and have a slight shadow effect.

What defenses are  
deployed in the kernel?



Linux

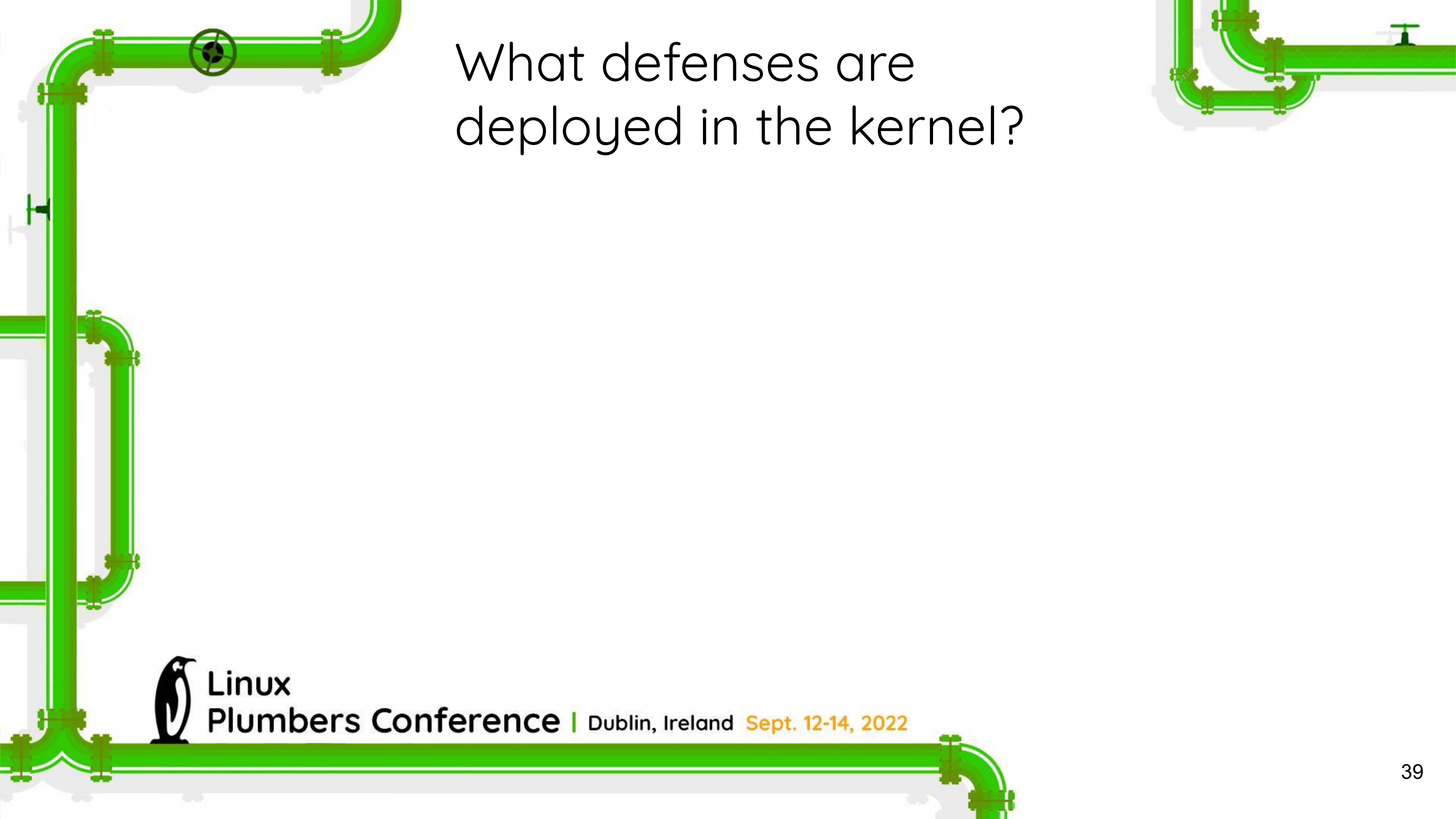
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# What defenses are deployed in the kernel?

lfence on copy-from-user:

```
static bool user_access_begin(const void __user *ptr, size_t len)
{
    if (unlikely(!access_ok(ptr, len)))
        return 0;
    __uaccess_begin_nospec();
    return 1;
}
```



A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text. The pipes are bright green with a slight shadow effect.

What defenses are  
deployed in the kernel?



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

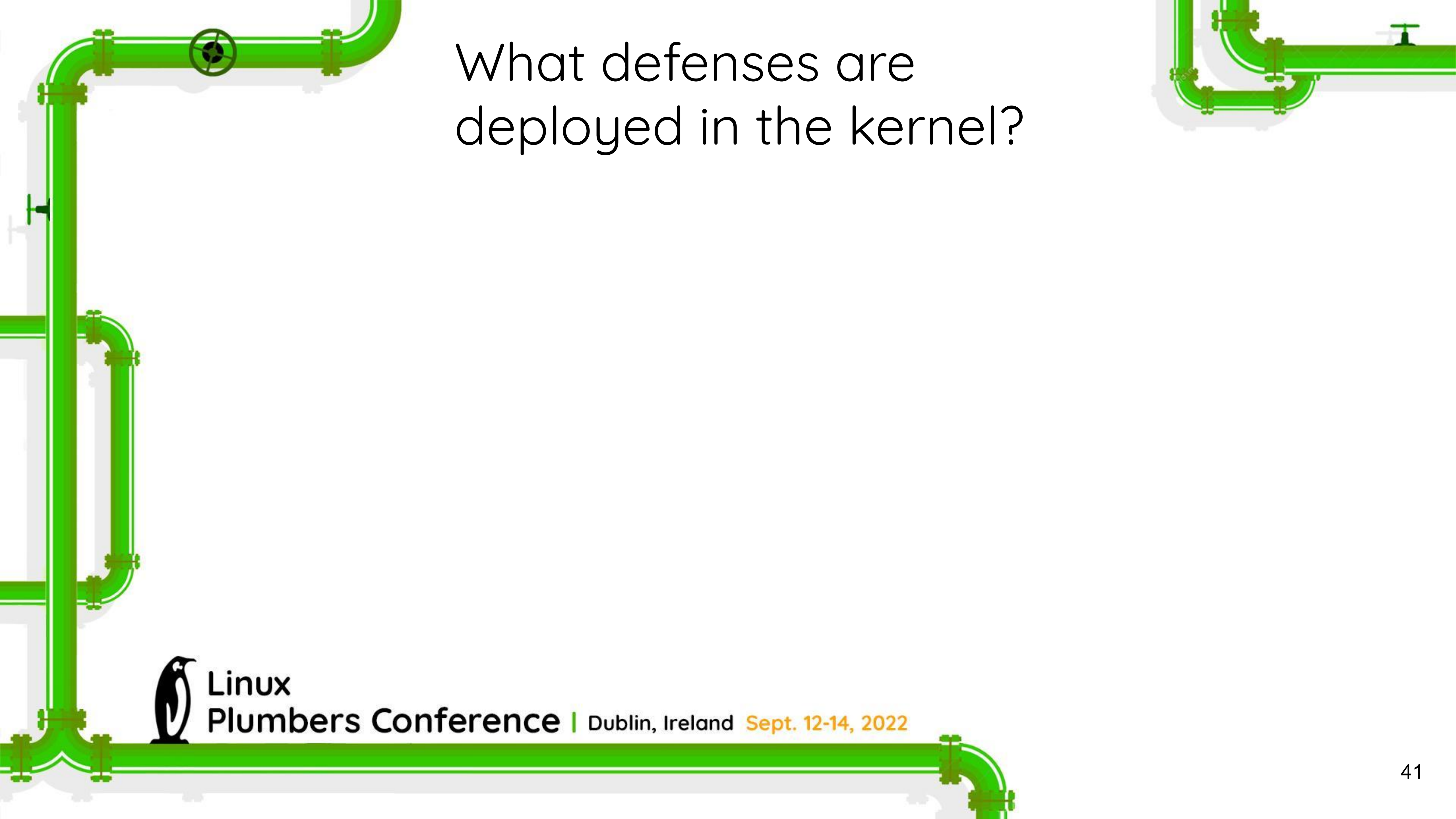
# What defenses are deployed in the kernel?

```
static __always_inline bool do_syscall_x64(struct pt_regs *regs, int nr)
{
    unsigned int unr = nr;

    if (likely(unr < NR_syscalls)) {
        unr = array_index_nospec(unr, NR_syscalls);
        regs->ax = sys_call_table[unr](regs);
        return true;
    }
    return false;
}
```





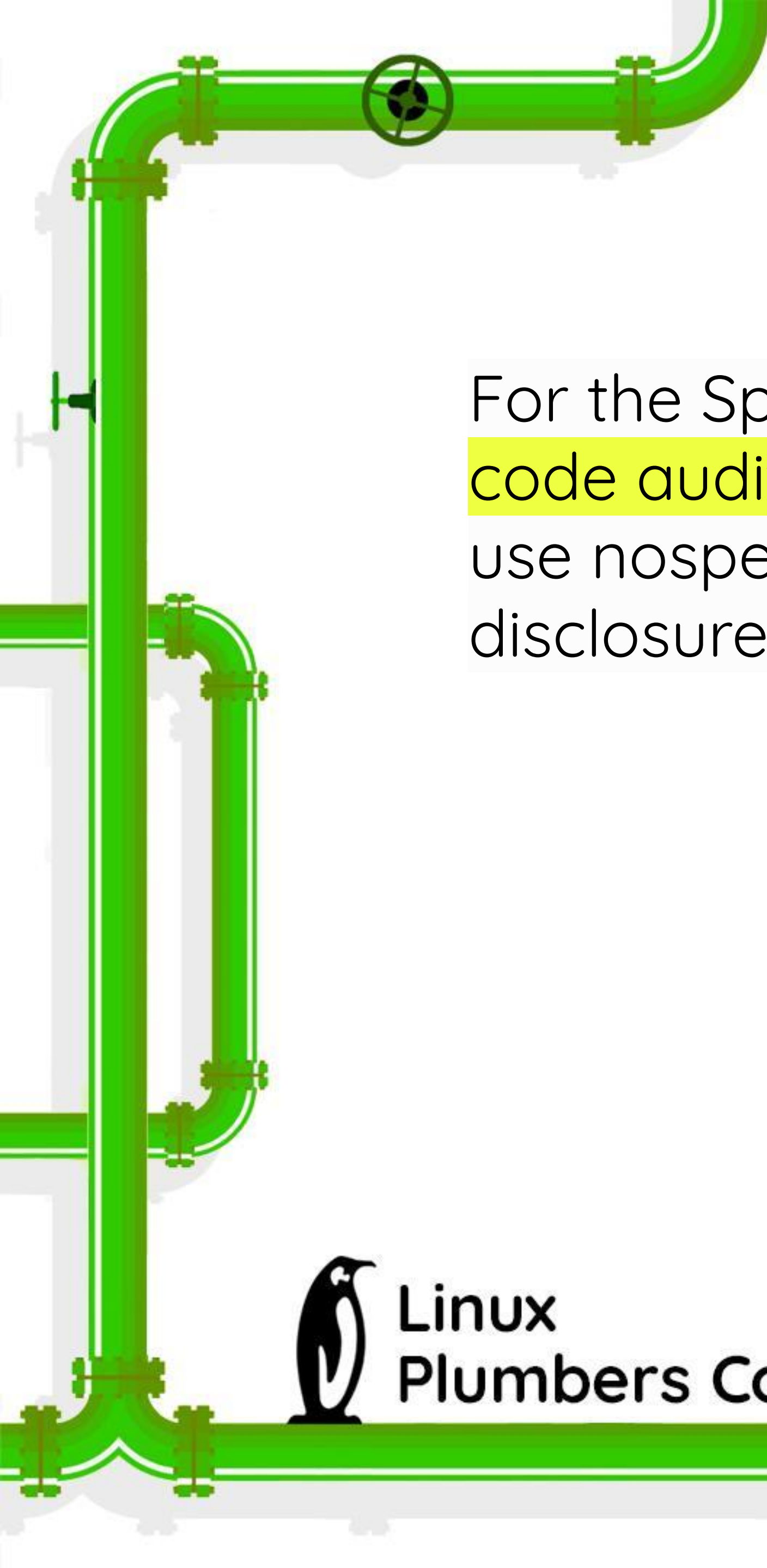
A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text. The pipes are bright green and have a slight shadow effect.

What defenses are  
deployed in the kernel?



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of green pipes with valves and elbows, running vertically on the left side of the slide and curving at the top and bottom.

# What defenses are deployed in the kernel?

For the Spectre variant 1, vulnerable kernel code (as determined by code audit or scanning tools) is annotated on a case by case basis to use nospec accessor macros for bounds clipping to avoid any usable disclosure gadgets.



# What defenses are deployed in the kernel?

For the Spectre variant 1, vulnerable kernel code (as determined by code audit or scanning tools) is annotated on a case by case basis to use nospec accessor macros for bounds clipping to avoid any usable disclosure gadgets. However, it may not cover all attack vectors for Spectre variant 1.





We can do **better**.

So Brian Johannesmeyer and I started with a **dynamic analysis approach** in 2019.



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

We're using something called  
**Dynamic Taint Analysis**, but what  
is it?



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# Dynamic Taint Analysis

```
int main(int argc, char *argv[]) {  
  
    char *prog = malloc(100);  
    strcpy(prog, argv[1]);  
  
    execve(prog,  
           (char *[]){prog, 0},  
           environ);  
}
```

# Dynamic Taint Analysis

```
int main(int argc, char *argv[]) {  
  
    char *prog = malloc(100);  
    strcpy(prog, argv[1]);  
  
    execve(prog,  
           (char *[]){prog, 0},  
           environ);  
}
```



# Dynamic Taint Analysis

Taint Source

```
int main(int argc, char *argv[]) {  
    dfsan_add_label(user, argv[1],  
                    strlen(argv[1]));  
    char *prog = malloc(100);  
    strcpy(prog, argv[1]);  
  
    execve(prog,  
           (char *[]){prog, 0},  
           environ);  
}
```



# Dynamic Taint Analysis

Taint Source

```
int main(int argc, char *argv[]) {  
    dfsan_add_label(user, argv[1],  
                    strlen(argv[1]));  
    char *prog = malloc(100);  
    strcpy(prog, argv[1]);  
  
    execve(prog,  
            (char *[]){prog, 0},  
            environ);  
}
```

Taint Propagation

# Dynamic Taint Analysis

Taint Source

```
int main(int argc, char *argv[]) {  
    dfsan_add_label(user, argv[1],  
                    strlen(argv[1]));  
    char *prog = malloc(100);  
    strcpy(prog, argv[1]);  
}
```

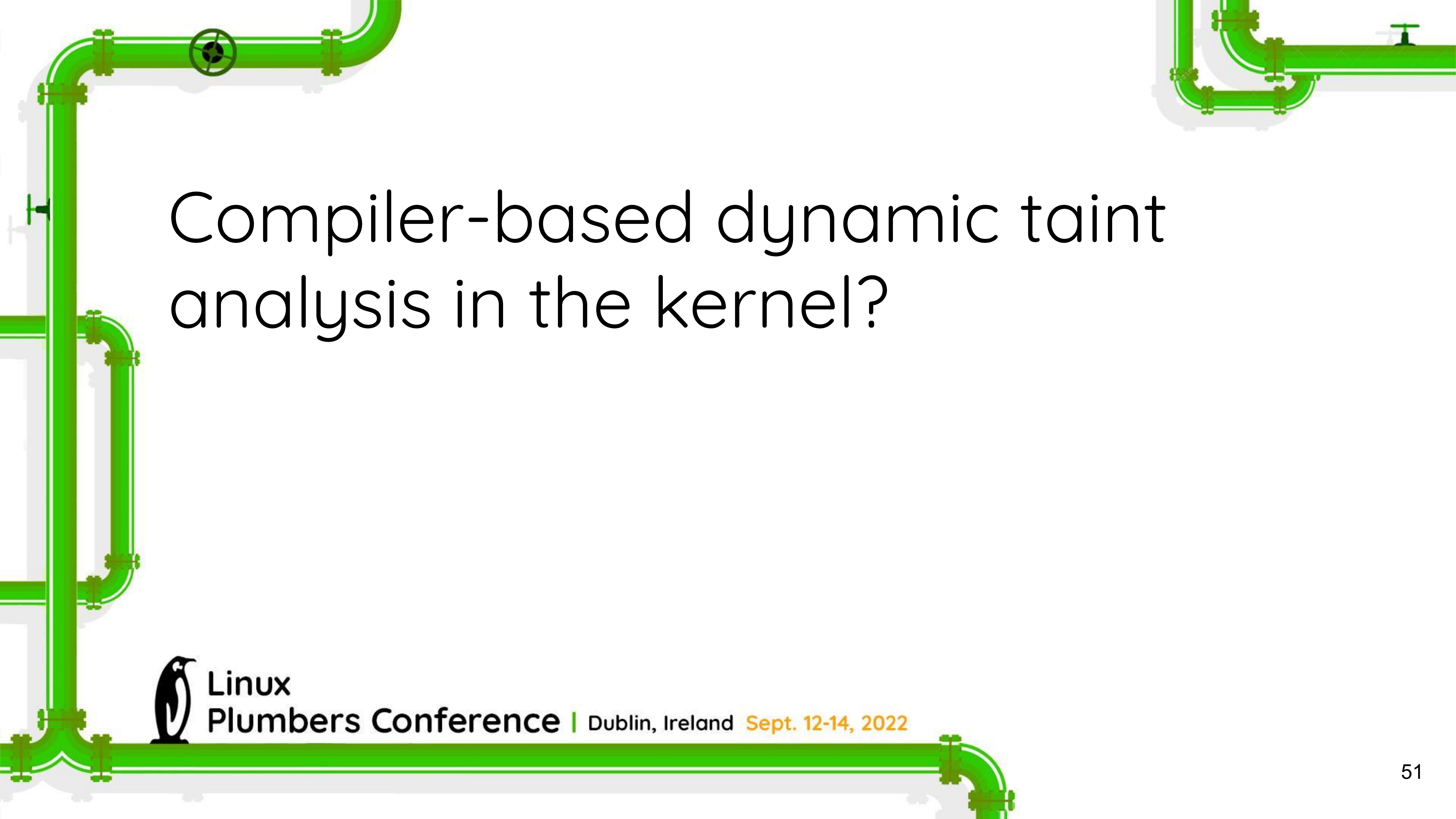
Taint Sink

```
execve(prog,  
        (char *[]){prog, 0},  
        environ);
```

Taint Propagation

Violation detected!



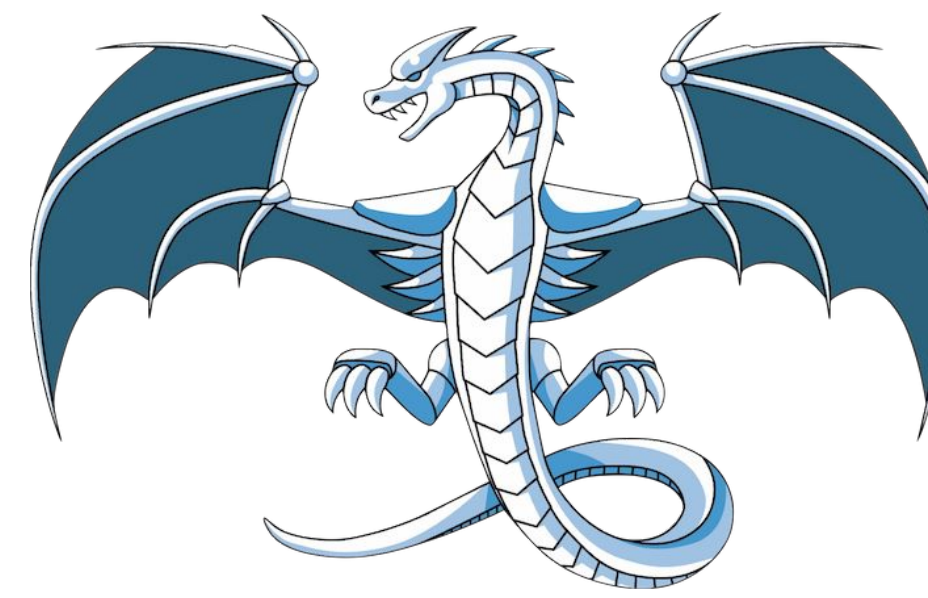
A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

# Compiler-based dynamic taint analysis in the kernel?



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



Compiler-based dynamic taint analysis in the kernel?

We've built **KDFSAN** for this project!

<https://github.com/vusec/kdfsan-linux/tree/kdfsan-linux-v5.13.7>



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

Our approach:

# Our approach:

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. **Fuzz** the syscall interface

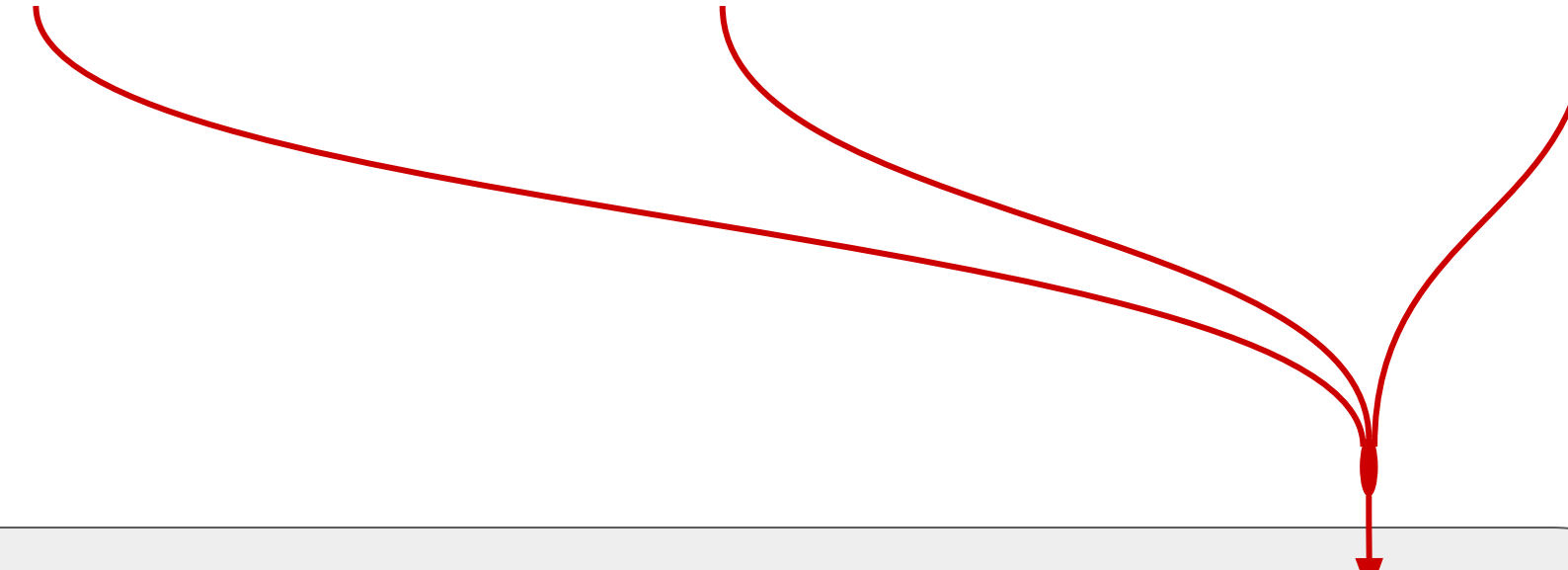
```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. **Fuzz** the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```






# Our approach:

1. **Fuzz** the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**



```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```



# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

# Our approach:

1. **Fuzz** the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an **attacker** label

3. Start **speculative emulation**

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. **Memory error detector** identifies unsafe access

# Our approach:

1. Fuzz the syscall interface

**x = -7**

**x = 3**

**x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. Memory error detector identifies unsafe access

5. Add a secret label

# Our approach:

1. Fuzz the syscall interface

**x = -7**

**x = 3**

**x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. Memory error detector identifies unsafe access

5. Add a secret label

# Our approach:

1. Fuzz the syscall interface

**x = -7**                      **x = 3**                      **x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. Memory error detector identifies unsafe access

5. Add a secret label

# Our approach:

1. Fuzz the syscall interface

**x = -7**

**x = 3**

**x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. Memory error detector identifies unsafe access

5. Add a secret label

6. Cache interference detector identifies gadget

# Our approach:

1. Fuzz the syscall interface

**x = -7**

**x = 3**

**x = 100000**

2. Add an attacker label

3. Start speculative emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. Memory error detector identifies unsafe access

5. Add a secret label

6. Cache interference detector identifies gadget

# Our approach:

1. **Fuzz** the syscall interface

**x = -7**

**x = 3**

**x = 100000**

2. Add an **attacker** label

3. Start **speculative** emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. **Memory error detector** identifies unsafe access

5. Add a **secret** label

6. **Cache interference detector** identifies gadget

7. **Revert** speculative operations



# Our approach:

1. **Fuzz** the syscall interface

**x = -7**

**x = 3**

**x = 100000**

2. Add an **attacker** label

3. Start **speculative** emulation

```
void syscall_handler(int x) {  
    ...  
    if (x < size) {  
        y = arr1[x];  
        z = arr2[y];  
    }  
}
```

4. **Memory error detector** identifies unsafe access

5. Add a **secret** label

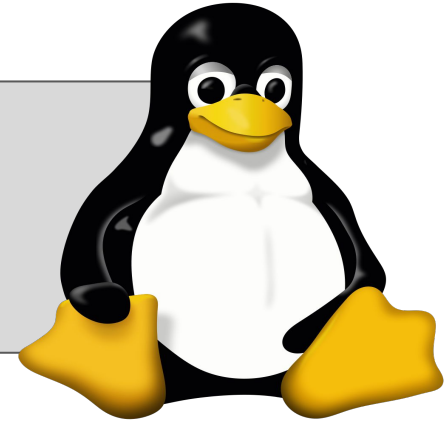
6. **Cache interference detector** identifies gadget

7. **Revert** speculative operations

Our implementation: KASPER

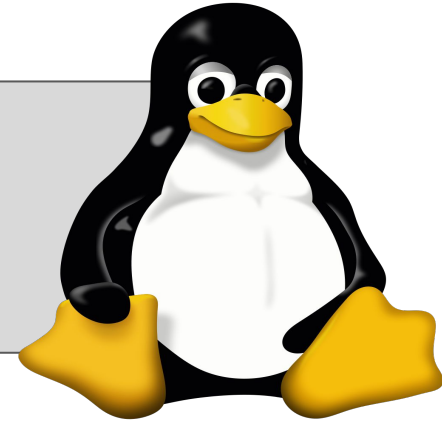
# Our implementation: KASPER

Linux kernel



# Our implementation: KASPER

Linux kernel



KASPER runtime libraries

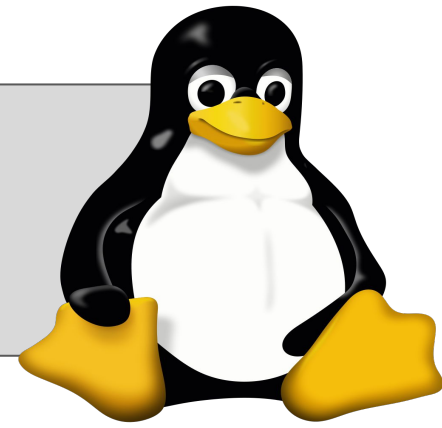


# Our implementation: KASPER



# Our implementation: KASPER

Linux kernel



KASPER runtime libraries



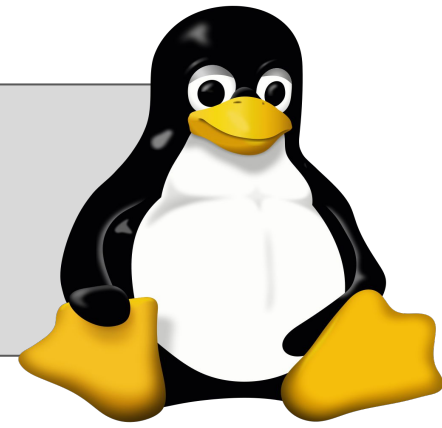
**Build the kernel** with KASPER's LLVM passes

KASPER-instrumented kernel



# Our implementation: KASPER

Linux kernel



KASPER runtime libraries



**Build the kernel** with KASPER's LLVM passes

KASPER-instrumented kernel




**Fuzz the kernel** so that **KASPER reports gadgets at runtime**

# Our implementation: KASPER

Linux kernel 

KASPER runtime libraries 

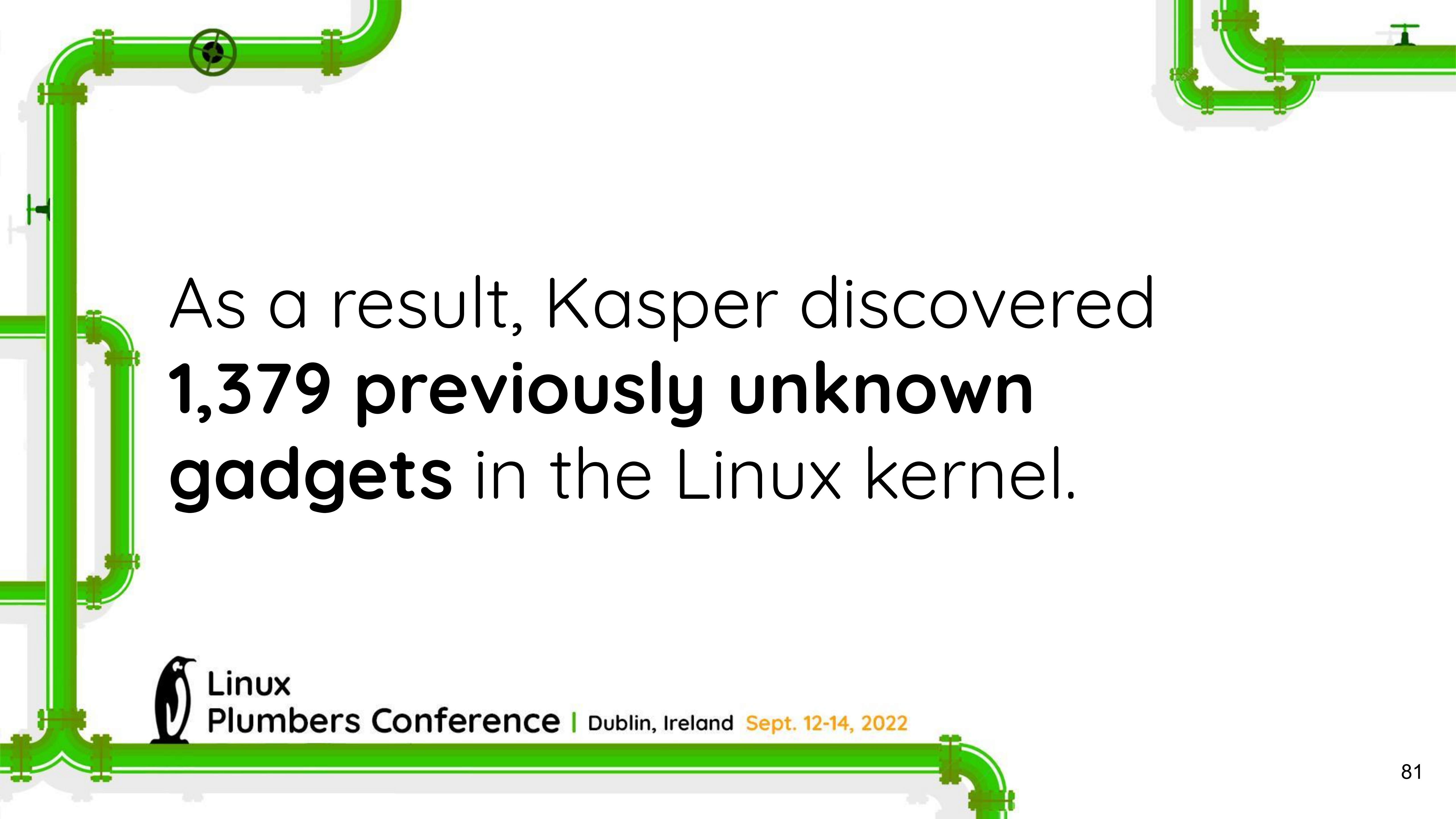
**Build the kernel** with KASPER's LLVM passes

KASPER-instrumented kernel 

**Fuzz the kernel** so that **KASPER reports gadgets at runtime**

Gadgets statistics 



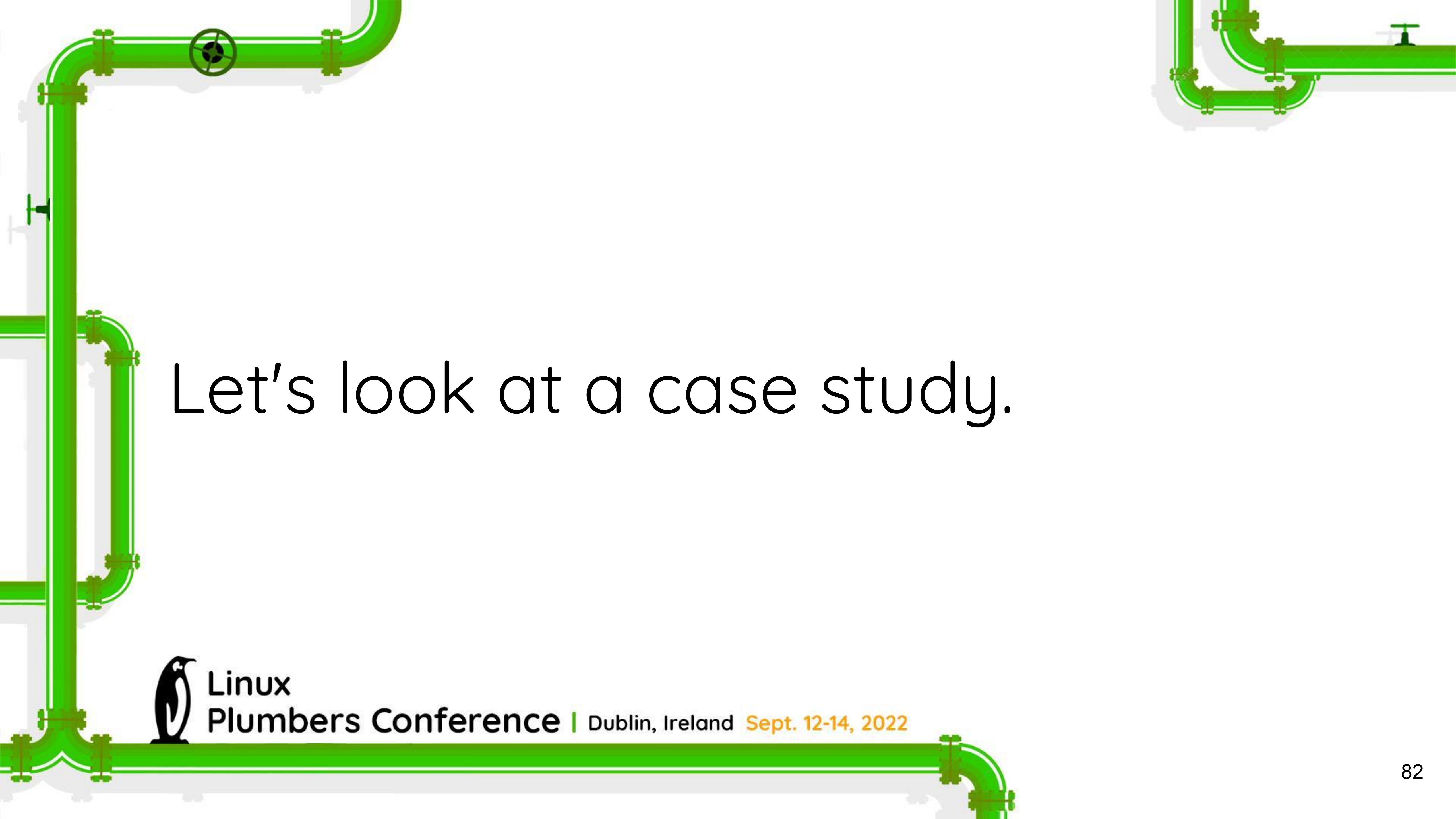
A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

As a result, Kasper discovered  
**1,379 previously unknown  
gadgets** in the Linux kernel.



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A stylized green pipe network graphic is overlaid on the slide. It features a vertical pipe on the left side with several horizontal pipes branching off. At the top, a horizontal pipe has a circular valve. At the bottom, a horizontal pipe runs across the width of the slide. In the top right corner, there is a separate section of pipe with a valve. The pipes are connected by various fittings and elbows.

Let's look at a case study.



**Linux**  
**Plumbers Conference** | Dublin, Ireland **Sept. 12-14, 2022**

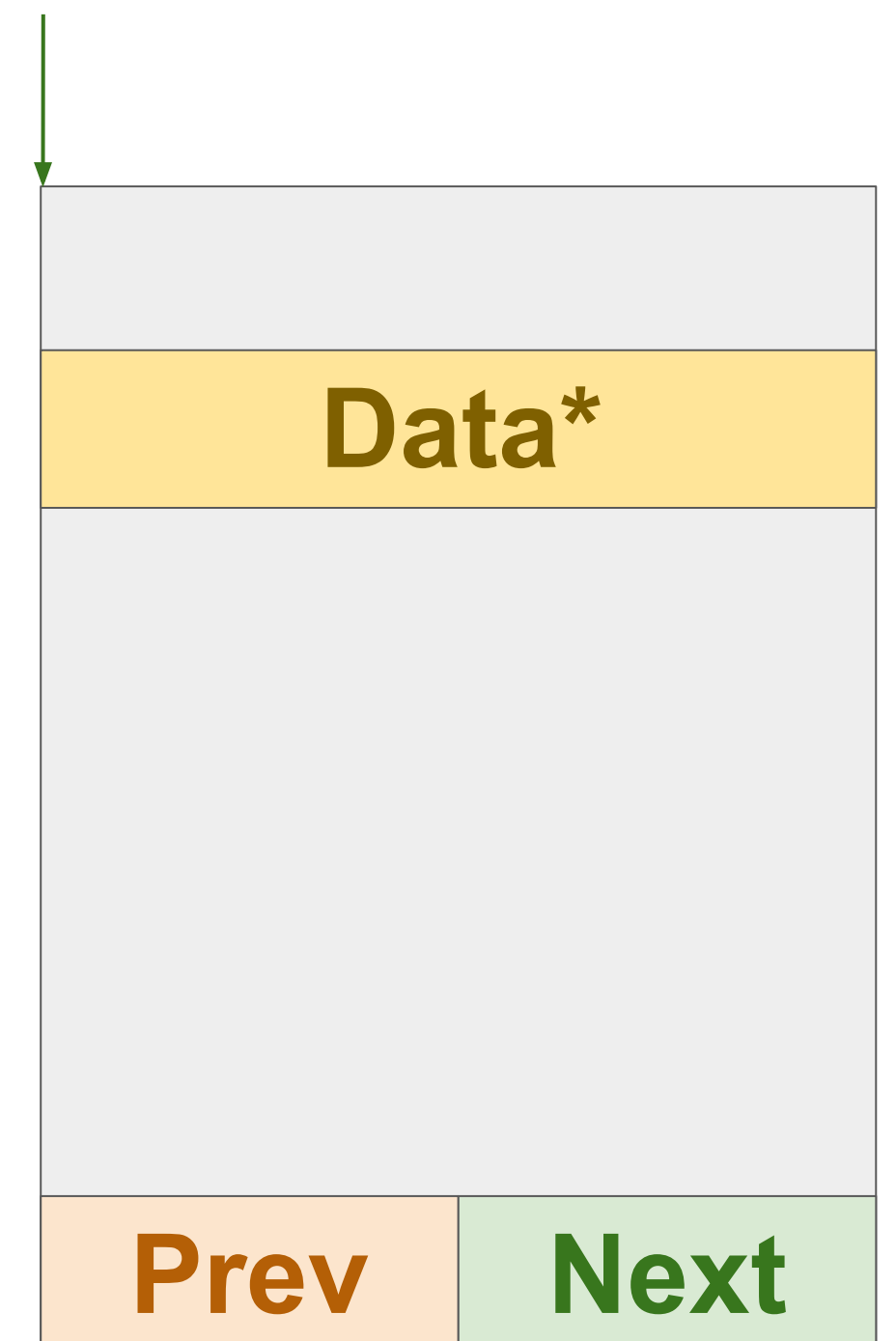
# Background: the list iterator

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

# Background: the list iterator

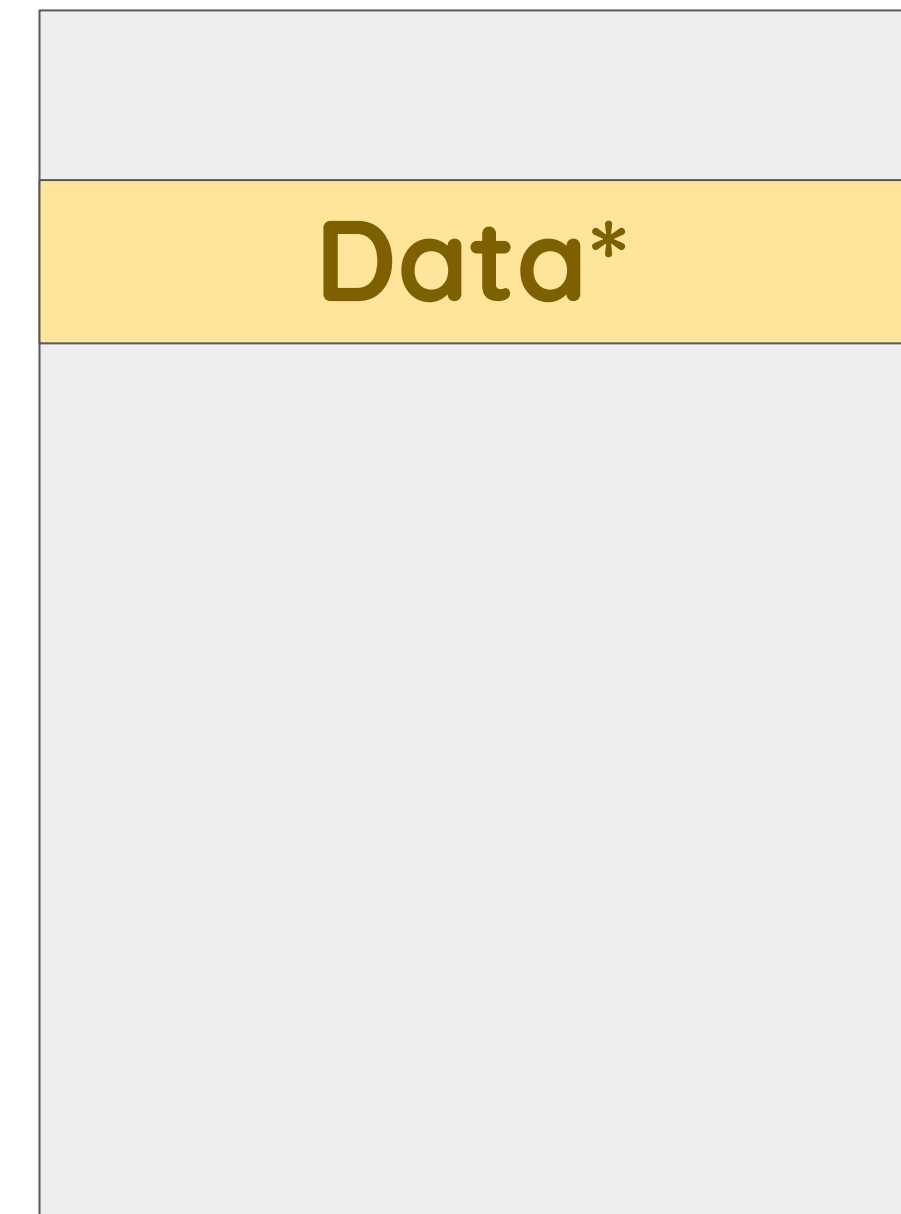
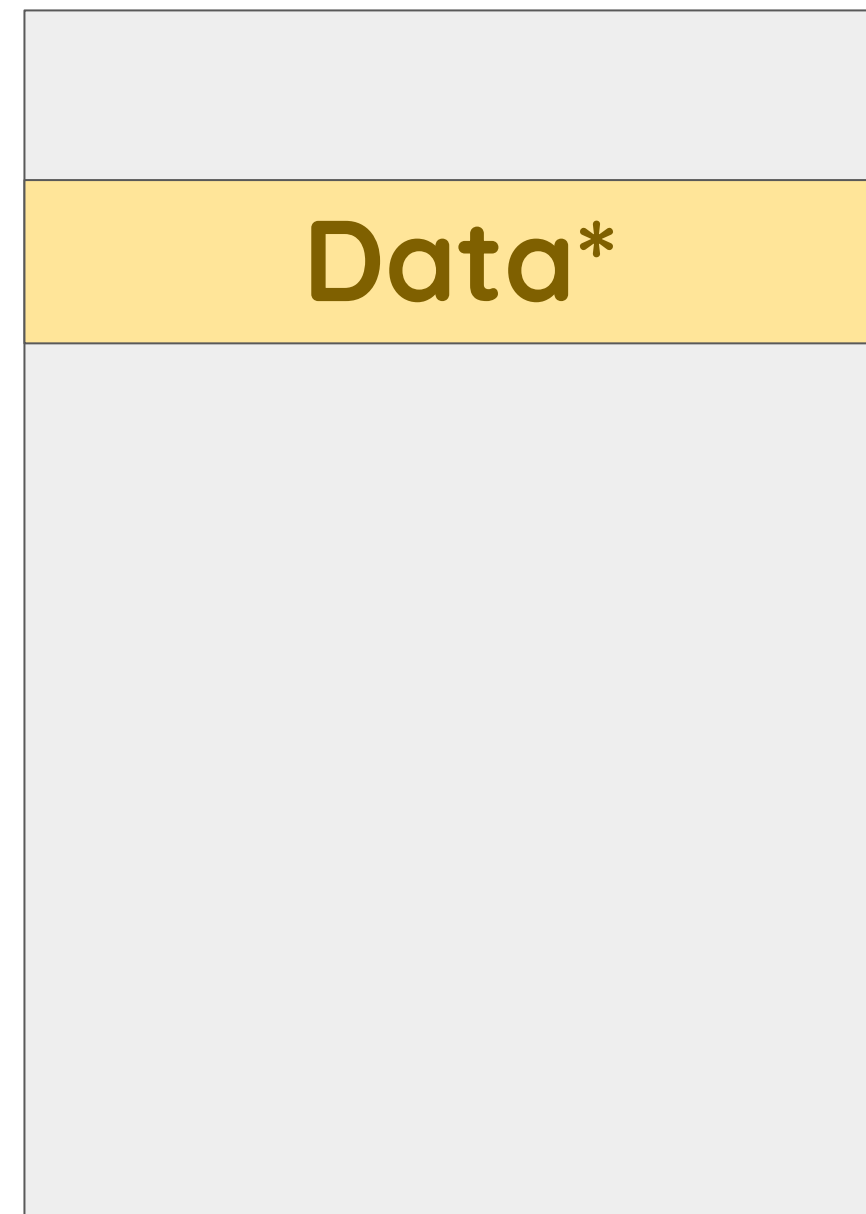
```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

**pos**



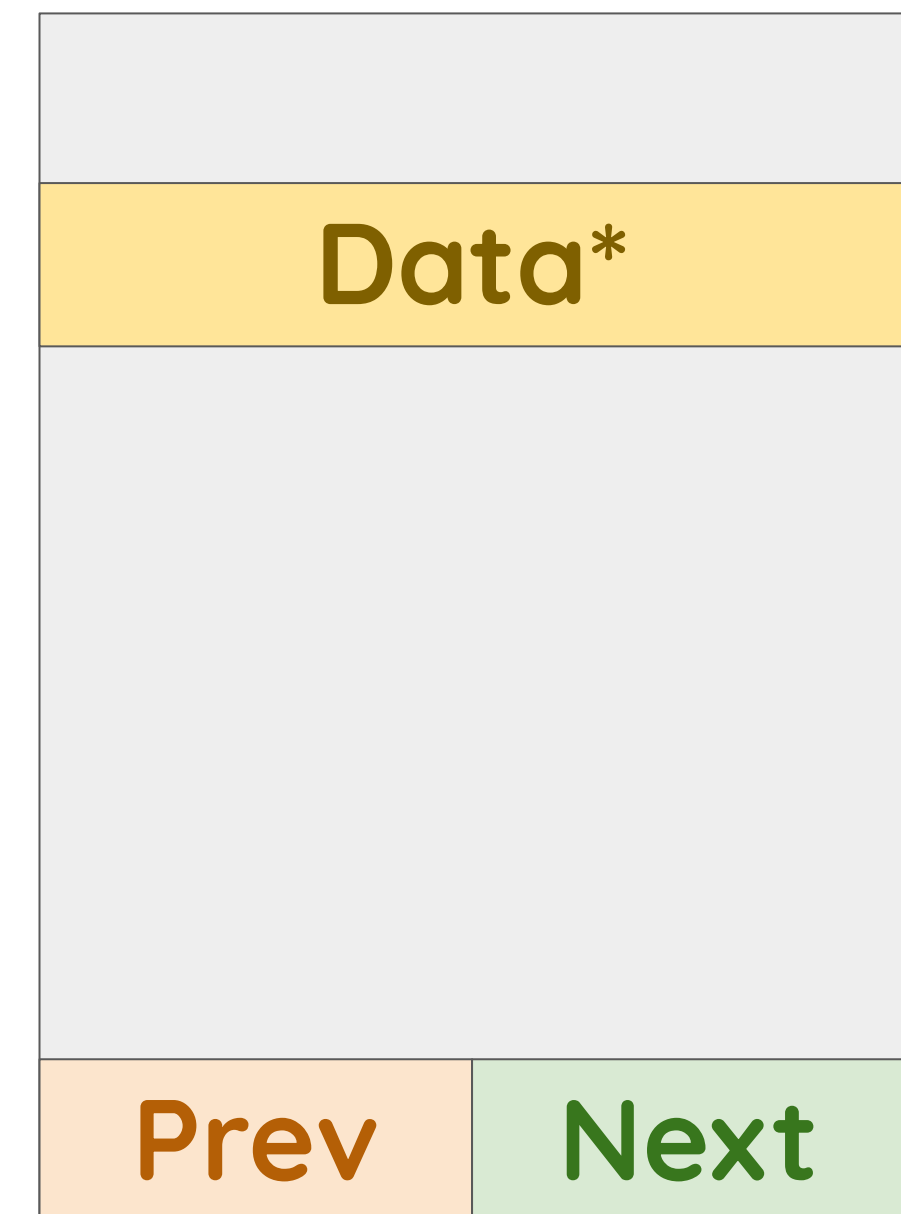
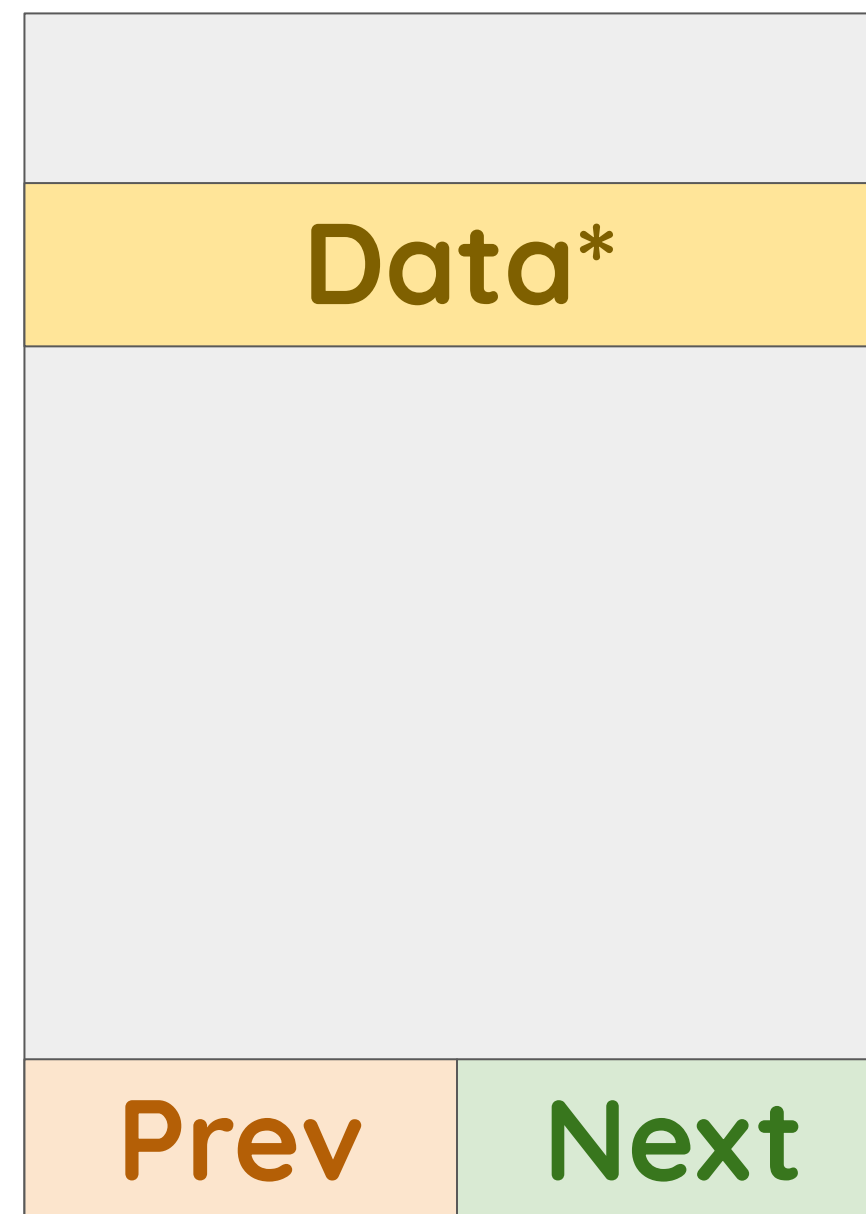
# Case study: list iterator

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```



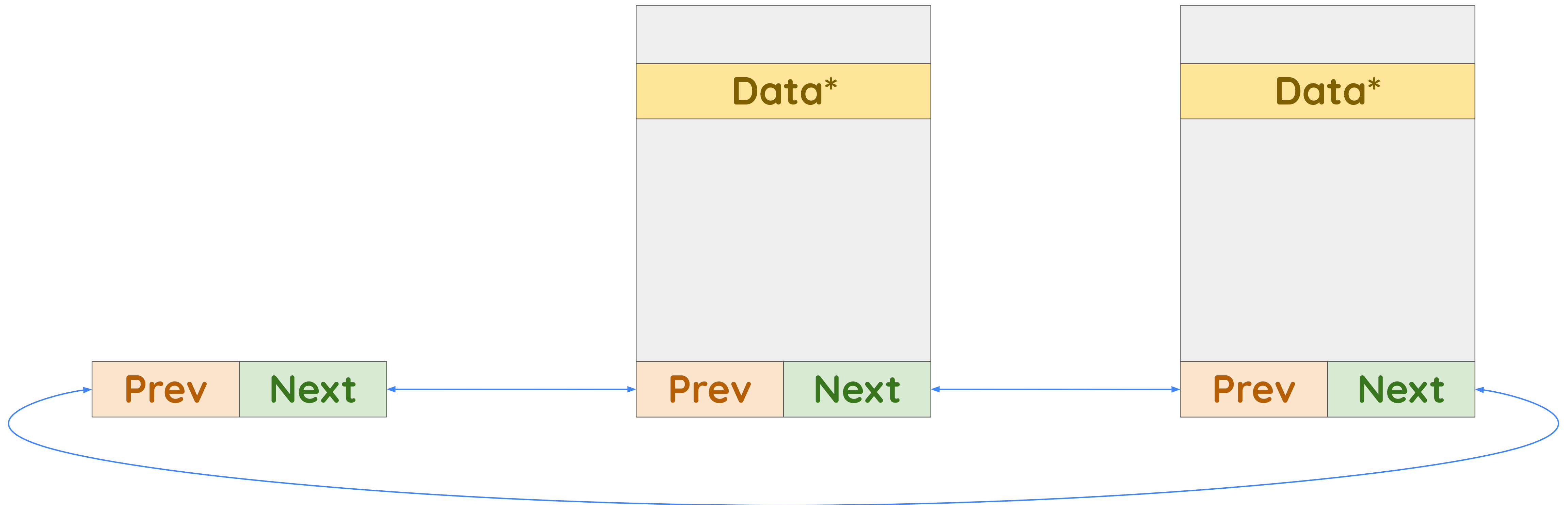
# Case study: list iterator

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```



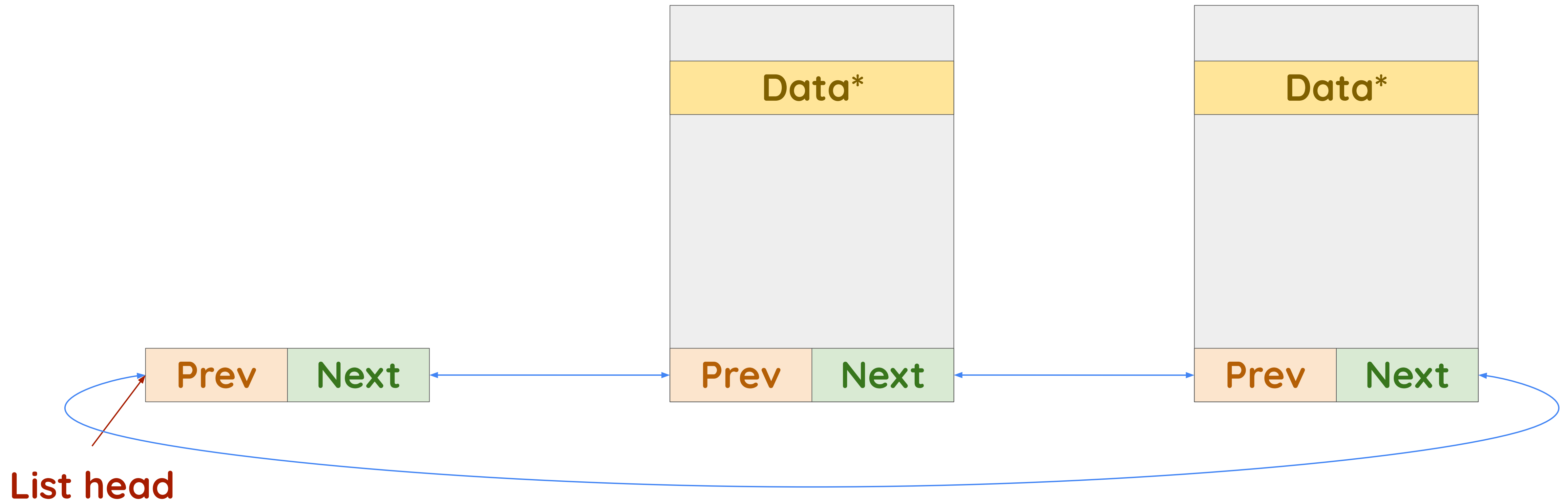
# Case study: list iterator

```
#define list_for_each_entry(pos, head, member)  
for (pos = list_first_entry(head,  
    typeof(*pos), member);  
    !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```



# Case study: list iterator

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

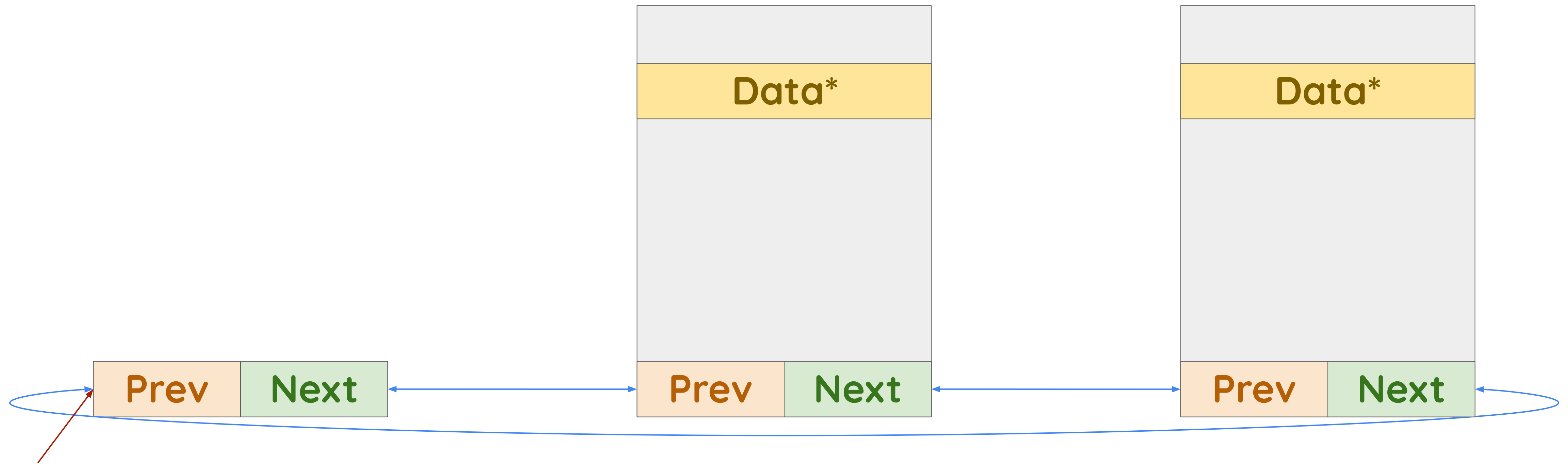




# Case study: list iterator

Iteration 1

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

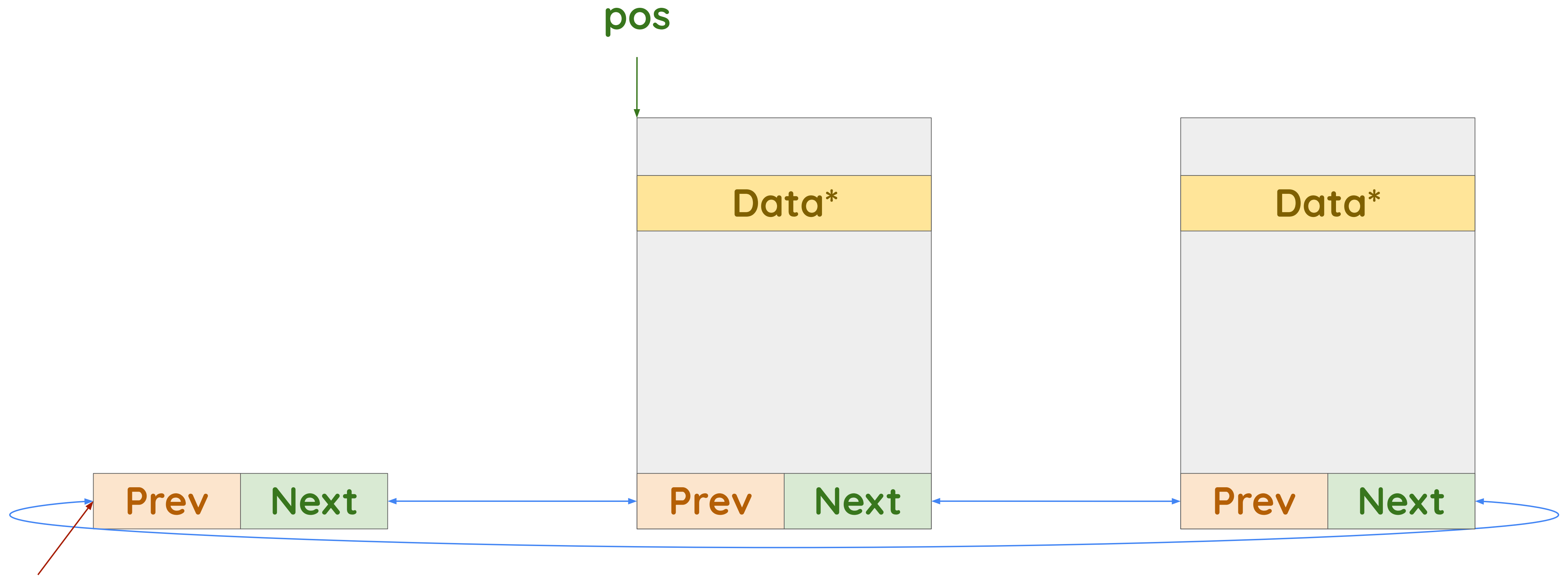


List head

# Case study: list iterator

Iteration 1

```
#define list_for_each_entry(pos, head, member)  
for (pos = list_first_entry(head,  
    typeof(*pos), member);  
    !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```

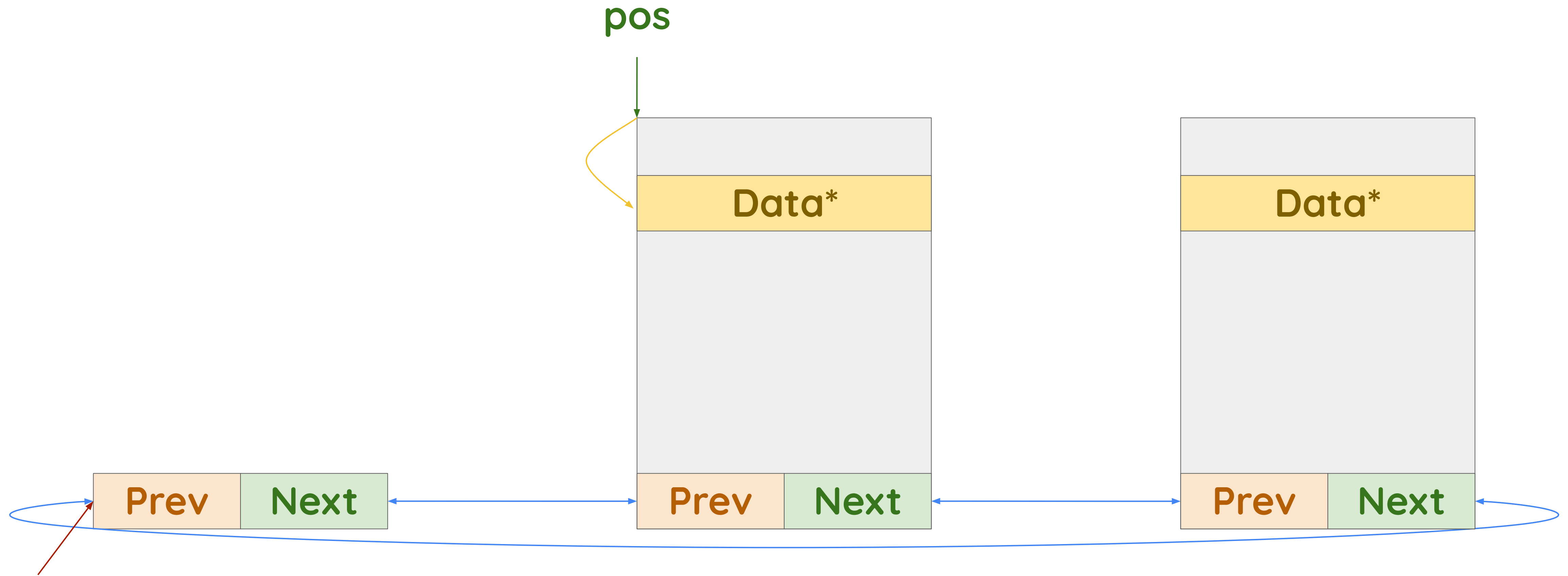


List head

# Case study: list iterator

Iteration 1

```
#define list_for_each_entry(pos, head, member)  
for (pos = list_first_entry(head,  
    typeof(*pos), member);  
    !list_entry_is_head(pos, head, member);  
    pos = list_next_entry(pos, member))
```

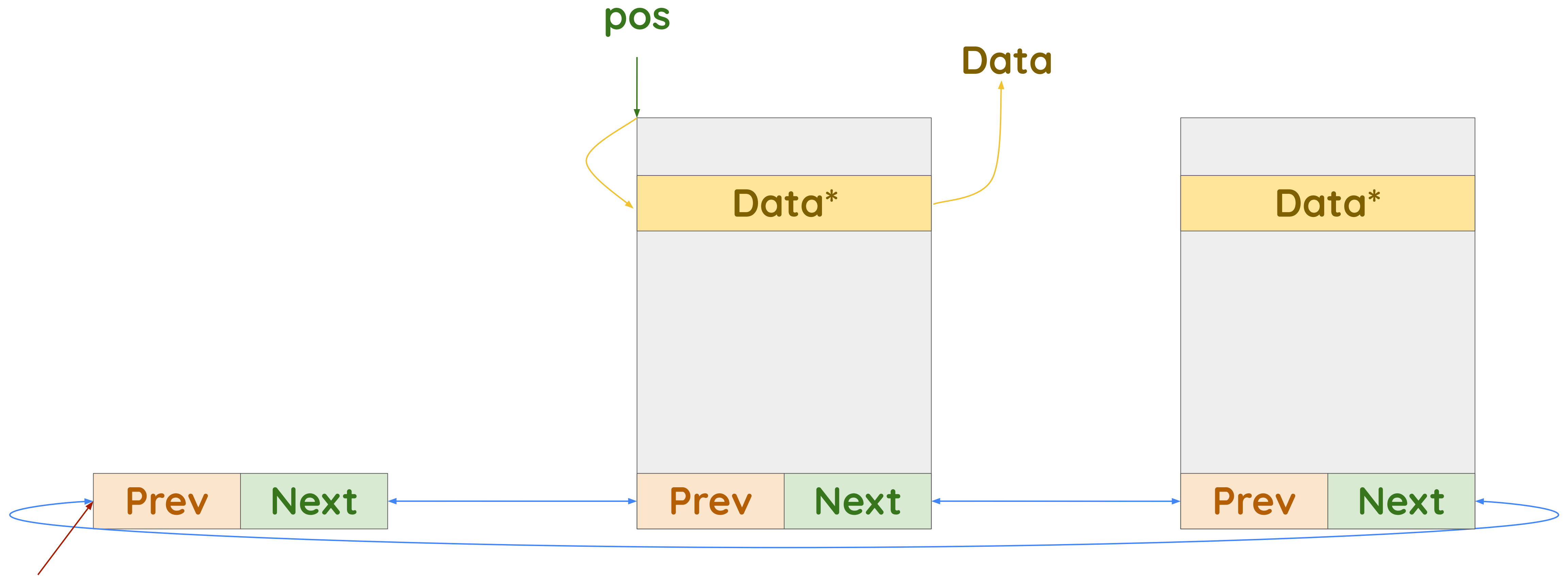


List head

# Case study: list iterator

Iteration 1

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typeof(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

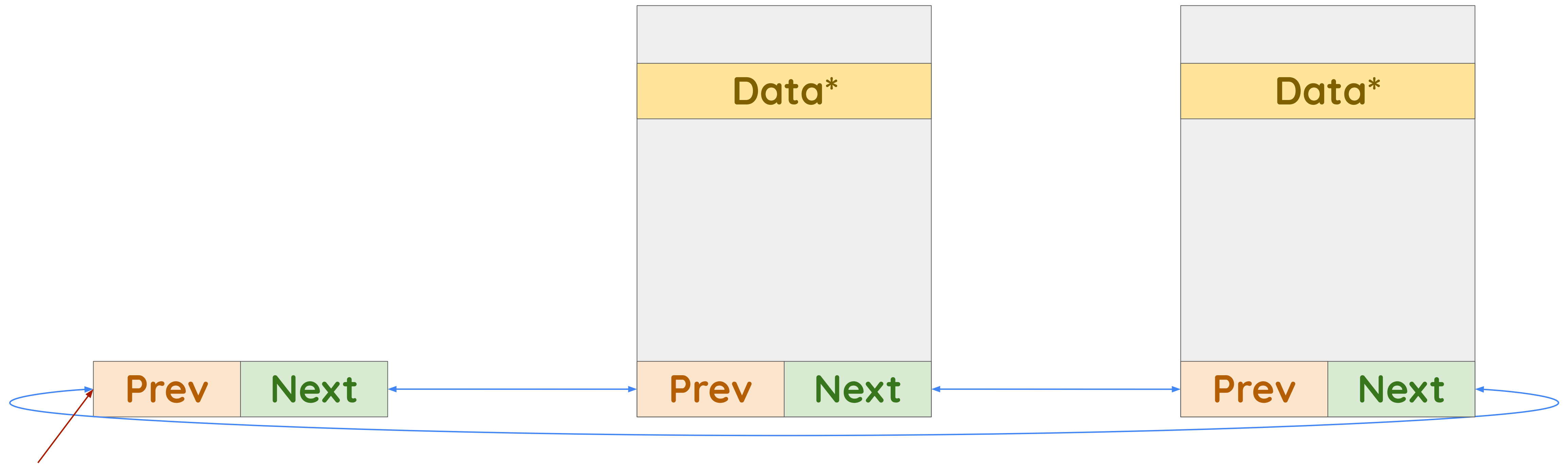


List head

# Case study: list iterator

Iteration 2

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

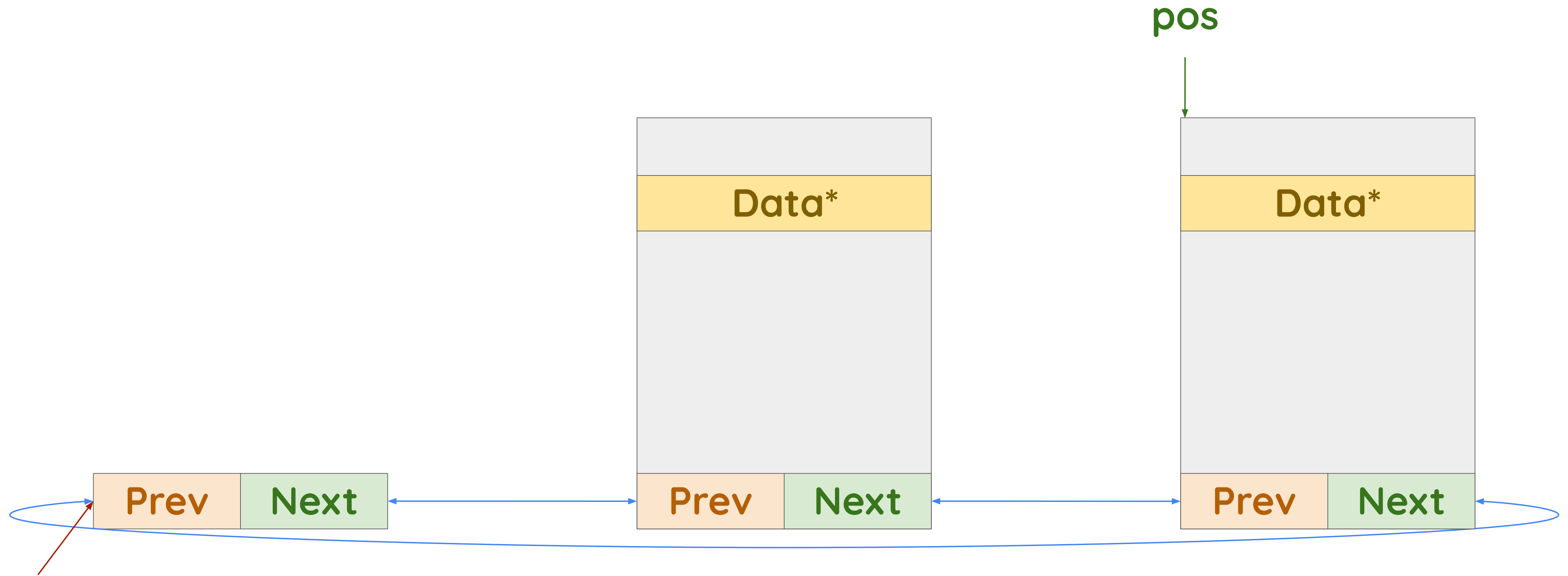


List head

# Case study: list iterator

Iteration 2

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

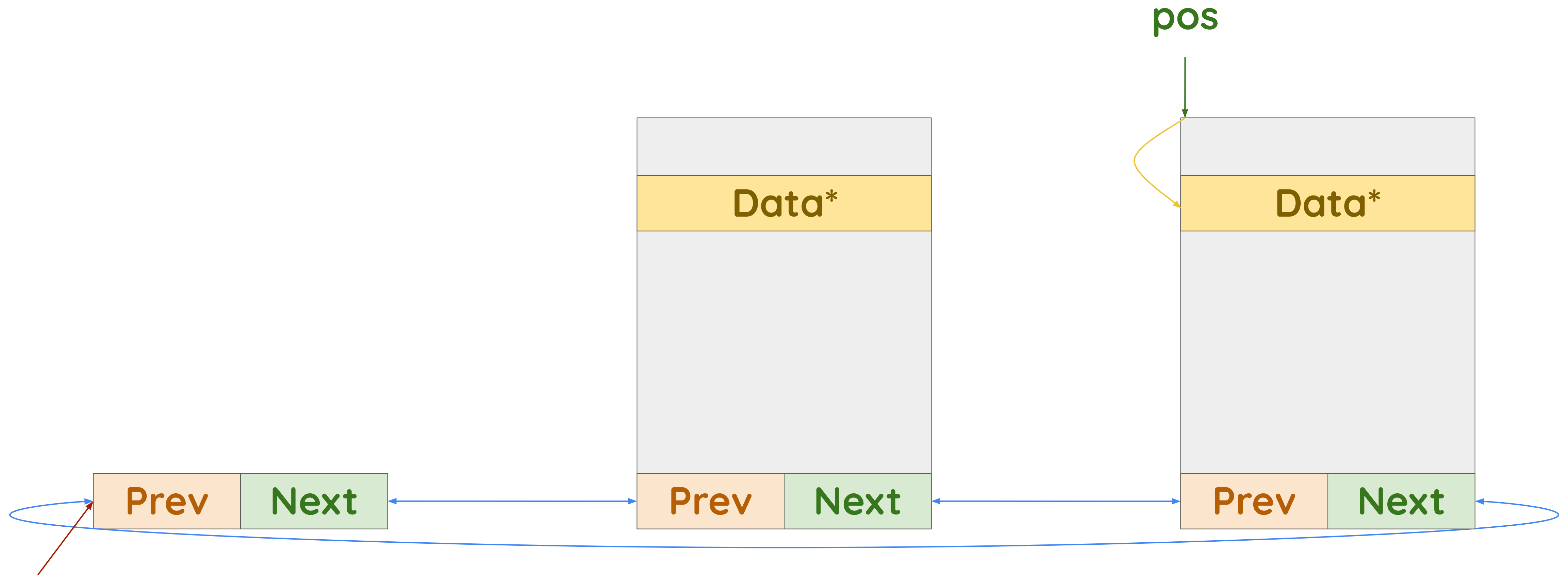


List head

# Case study: list iterator

Iteration 2

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

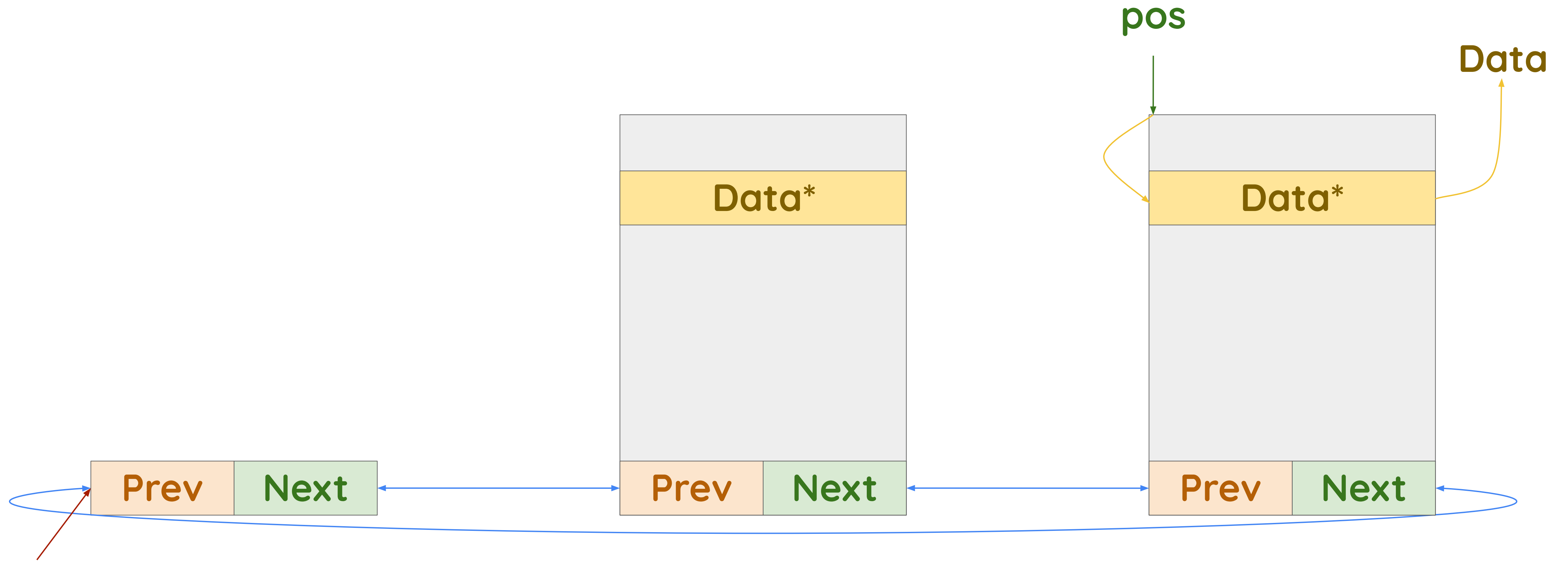


List head

# Case study: list iterator

Iteration 2

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```



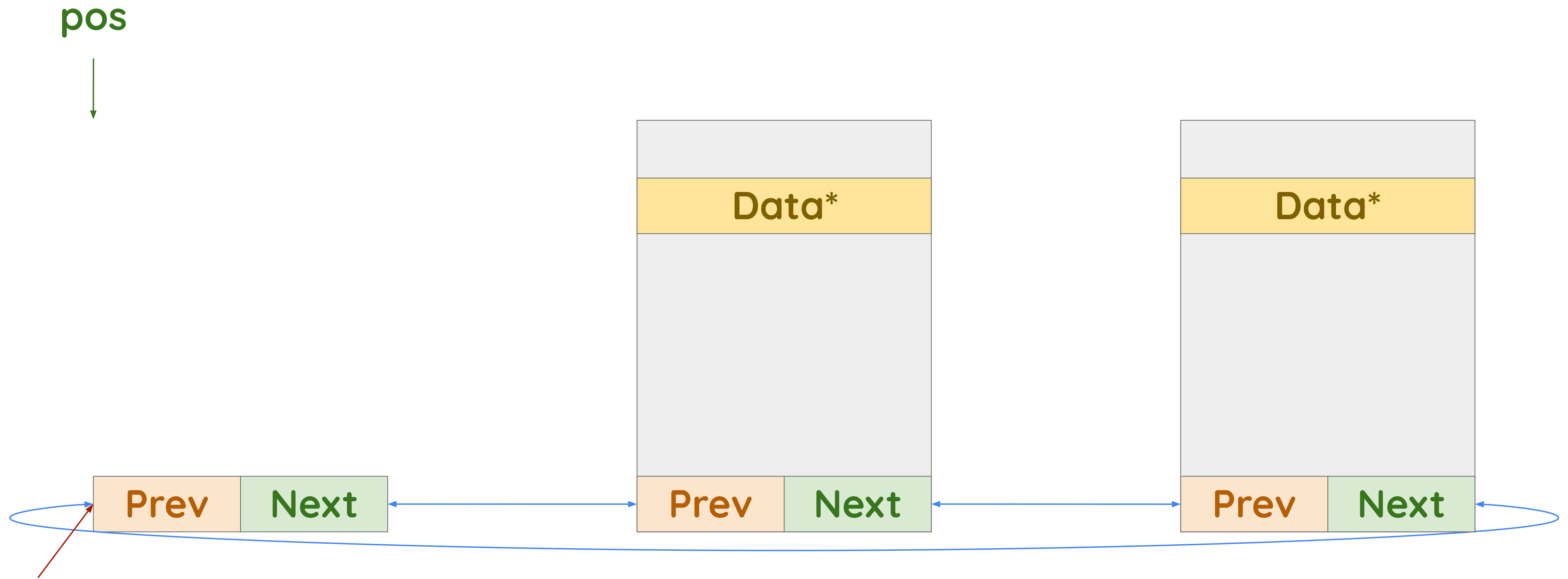
List head



# Case study: list iterator

Iteration 3 (termination)

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

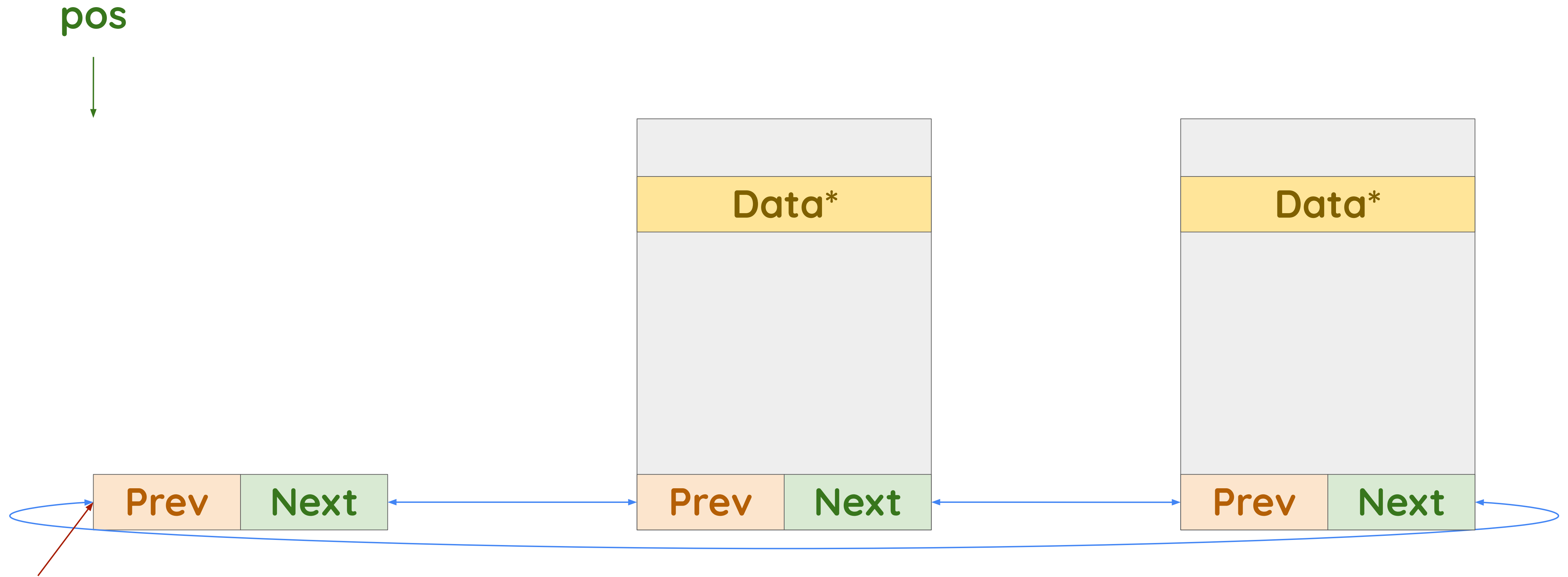


List head

# Case study: list iterator

Iteration 3 (misprediction)

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```



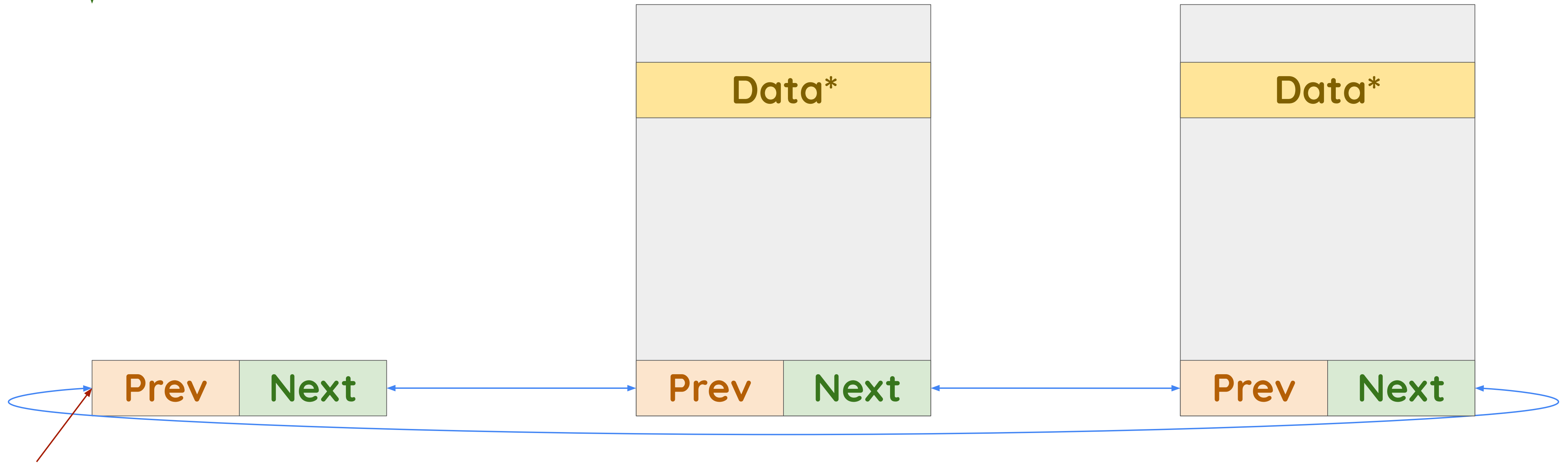
List head

# Case study: list iterator

Iteration 3 (misprediction)



pos



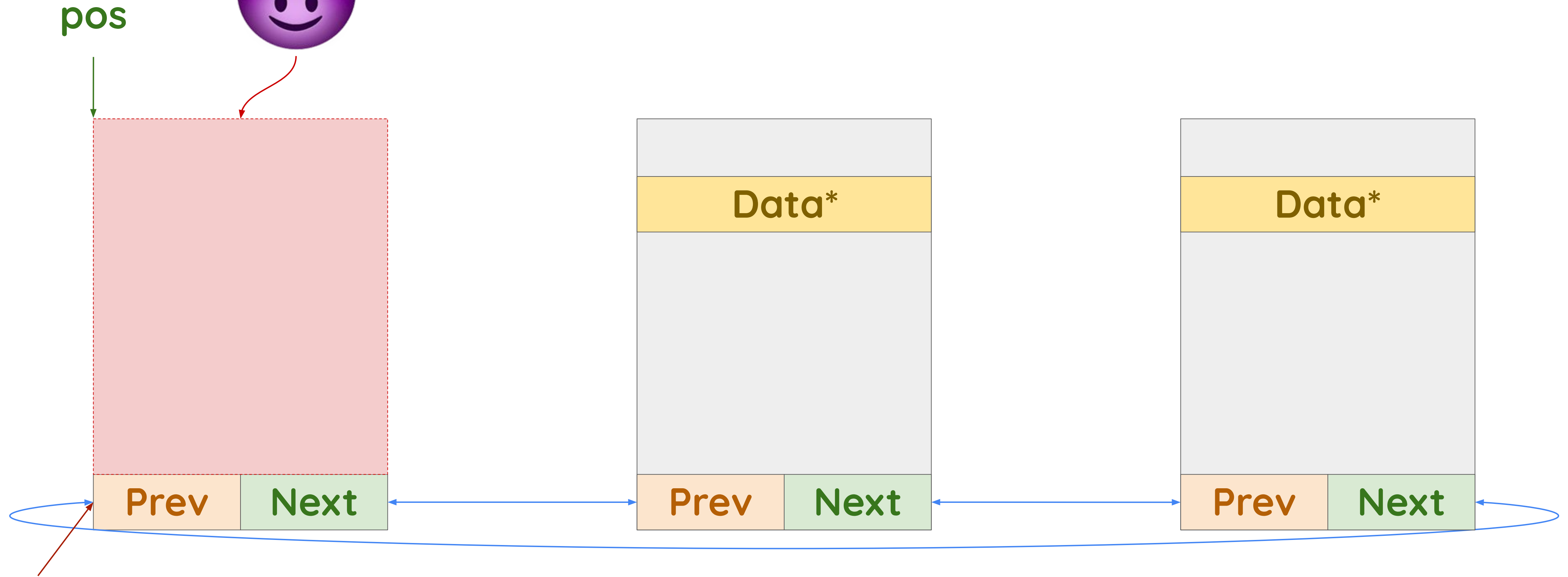
```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

List head

# Case study: list iterator

Iteration 3 (misprediction)

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```

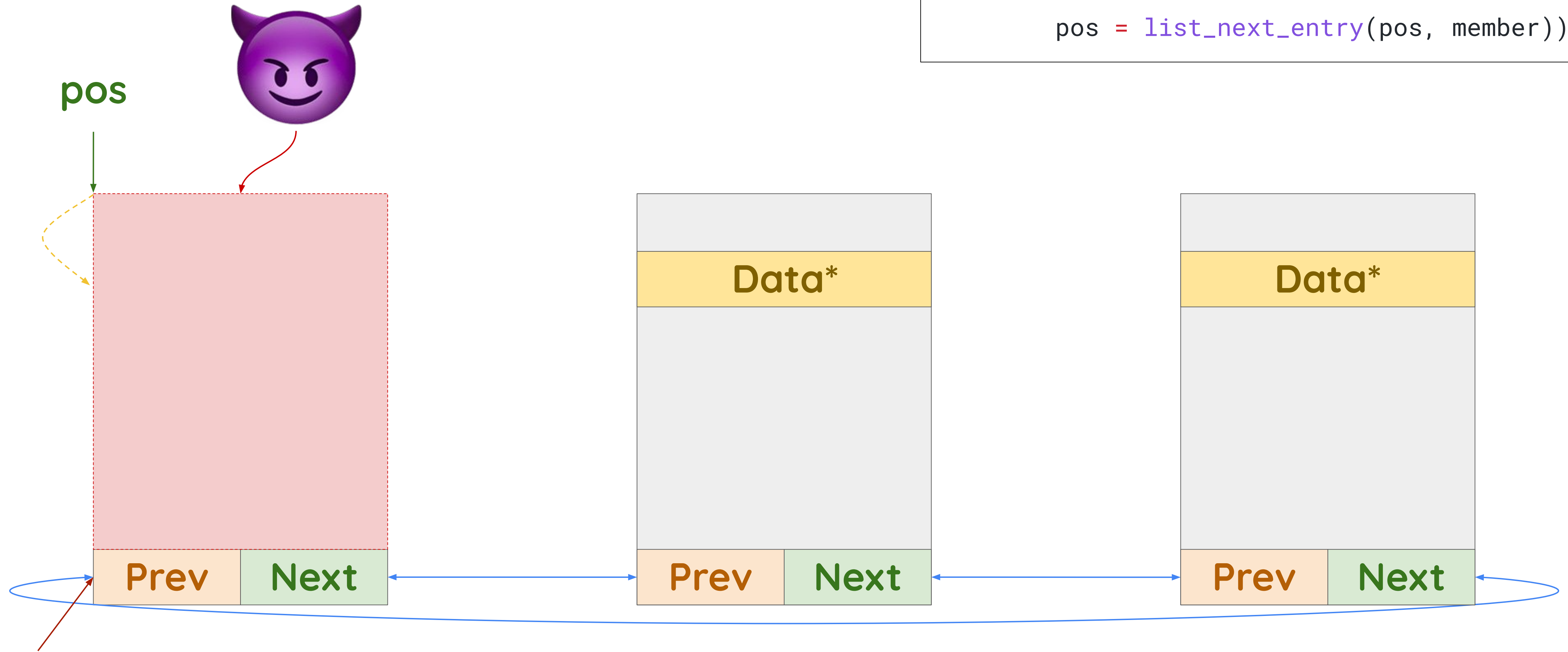


List head

# Case study: list iterator

Iteration 3 (misprediction)

```
#define list_for_each_entry(pos, head, member)
for (pos = list_first_entry(head,
        typedef(*pos), member);
    !list_entry_is_head(pos, head, member);
    pos = list_next_entry(pos, member))
```

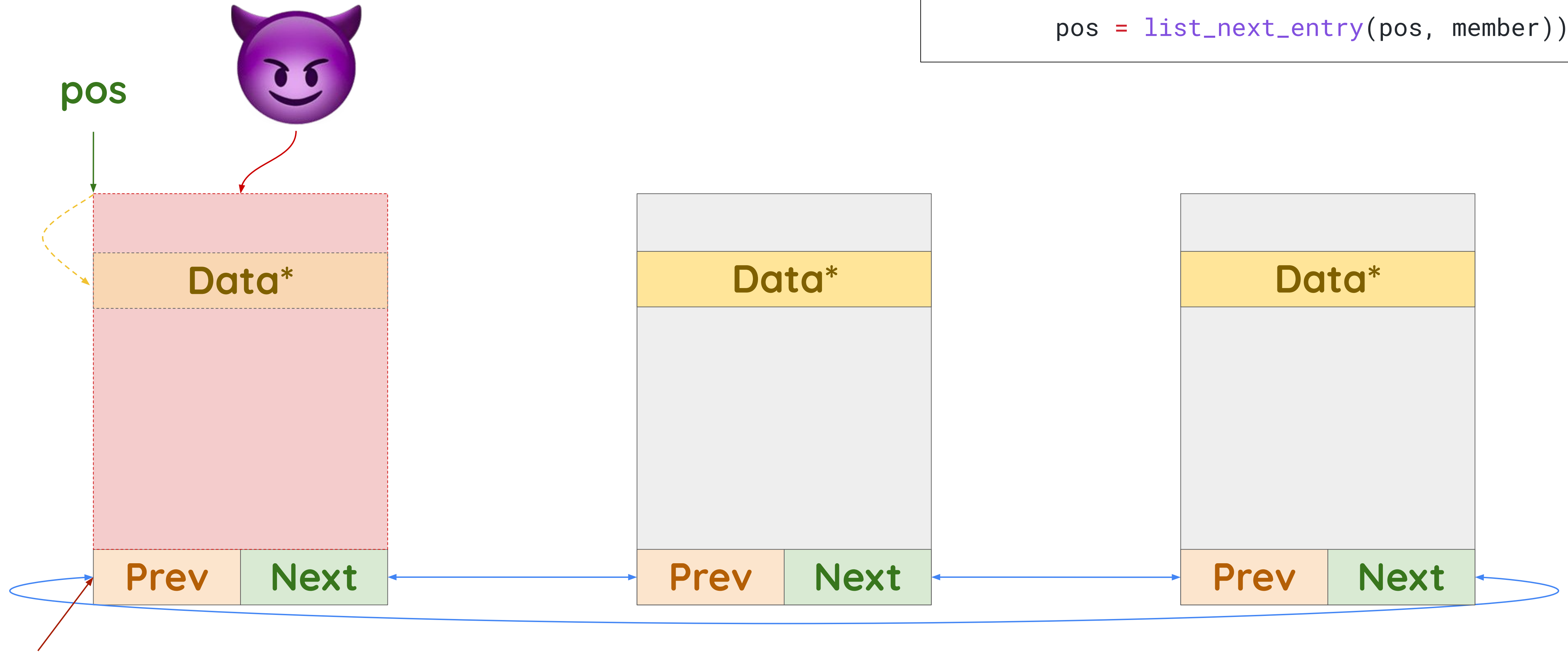


List head

# Case study: list iterator

Iteration 3 (misprediction)

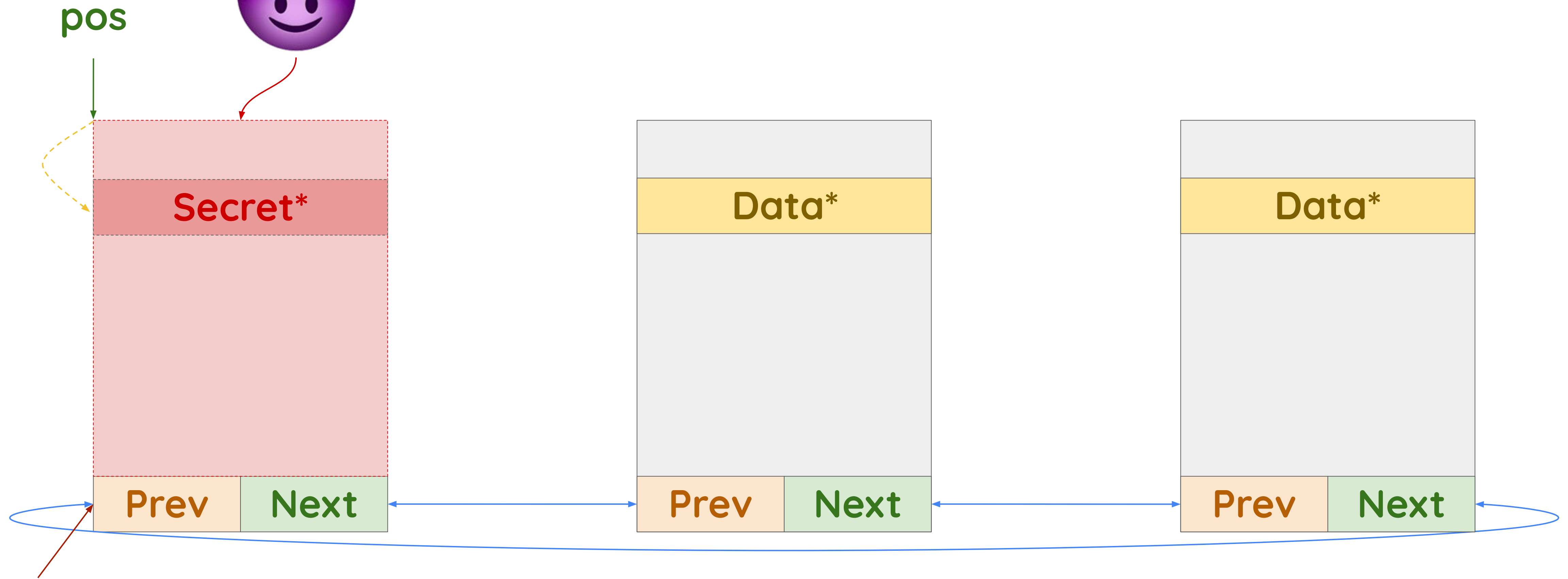
```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head,
        typedef(*pos), member);
        !list_entry_is_head(pos, head, member);
        pos = list_next_entry(pos, member))
```



List head

# Case study: list iterator

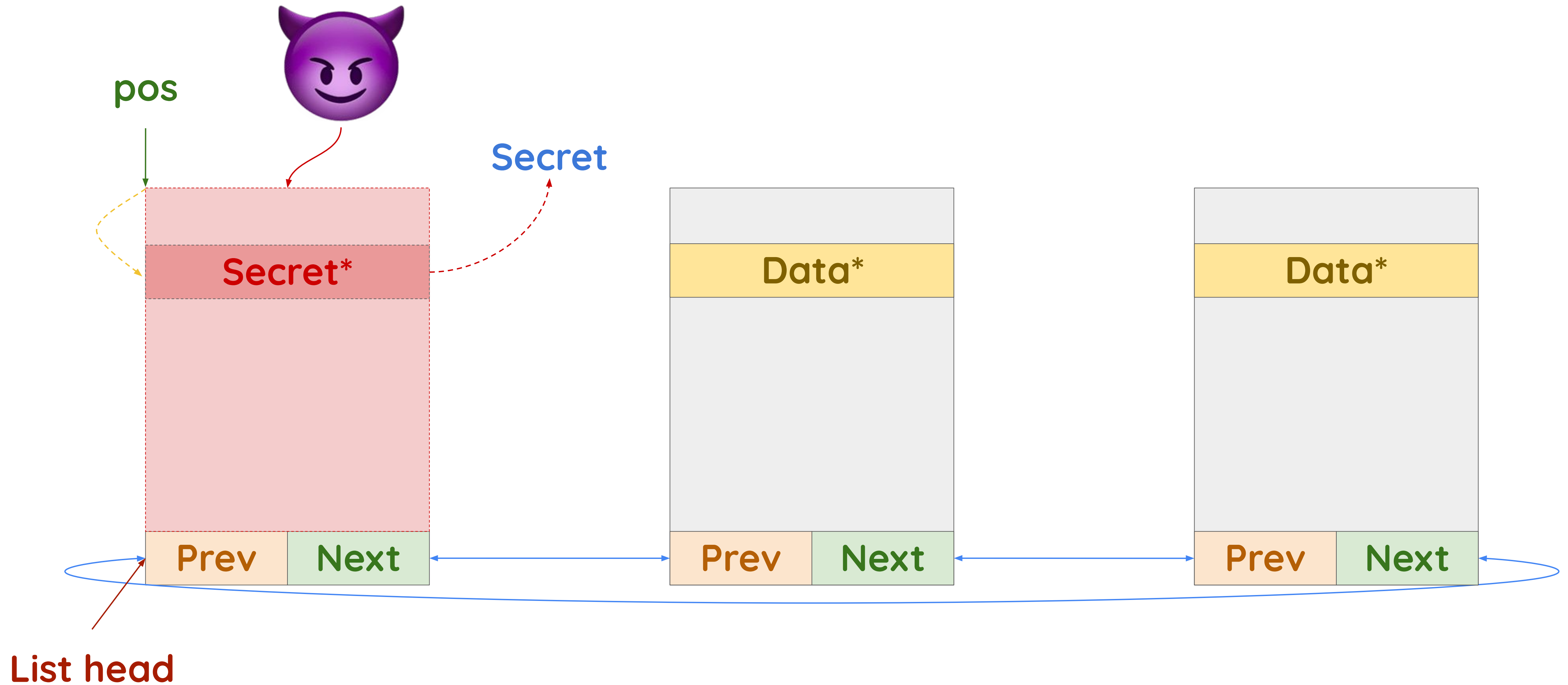
Iteration 3 (misprediction)



List head

# Case study: list iterator

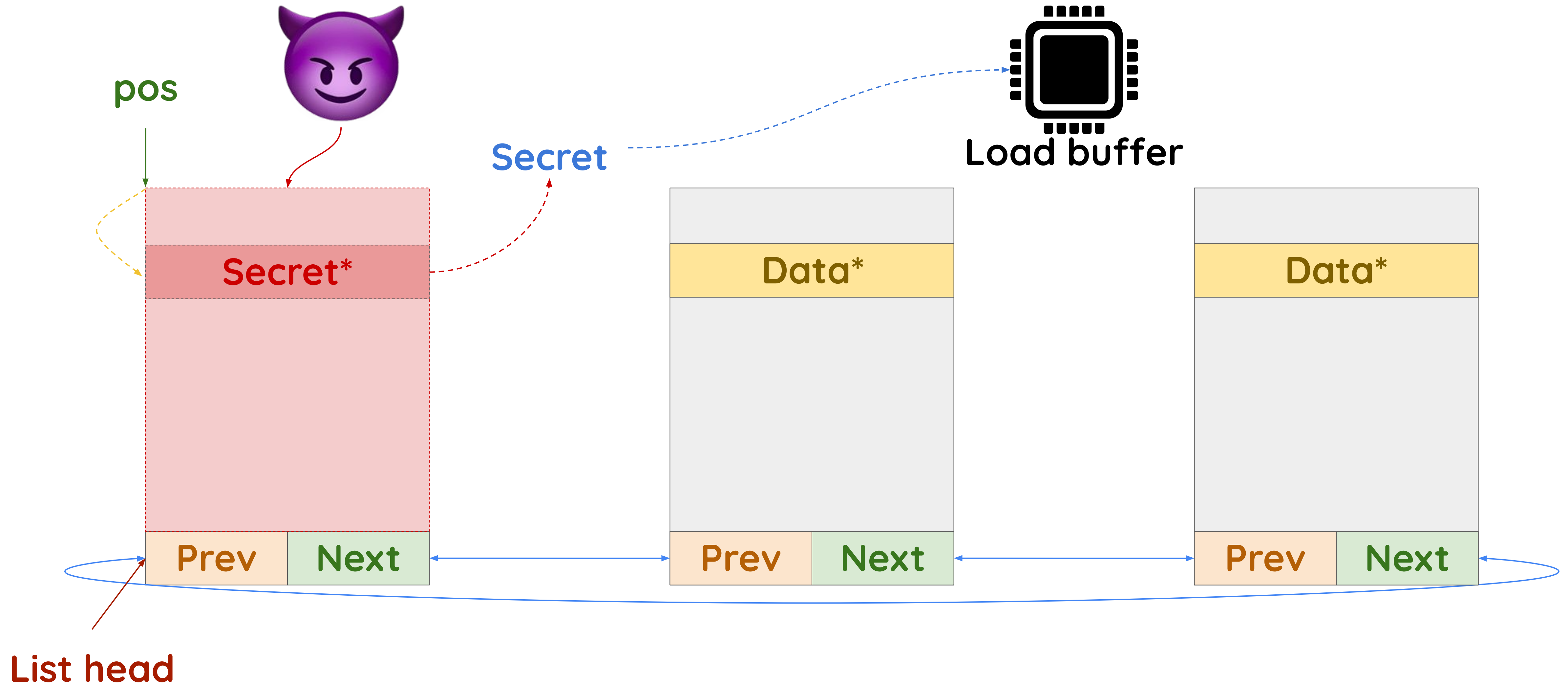
Iteration 3 (misprediction)





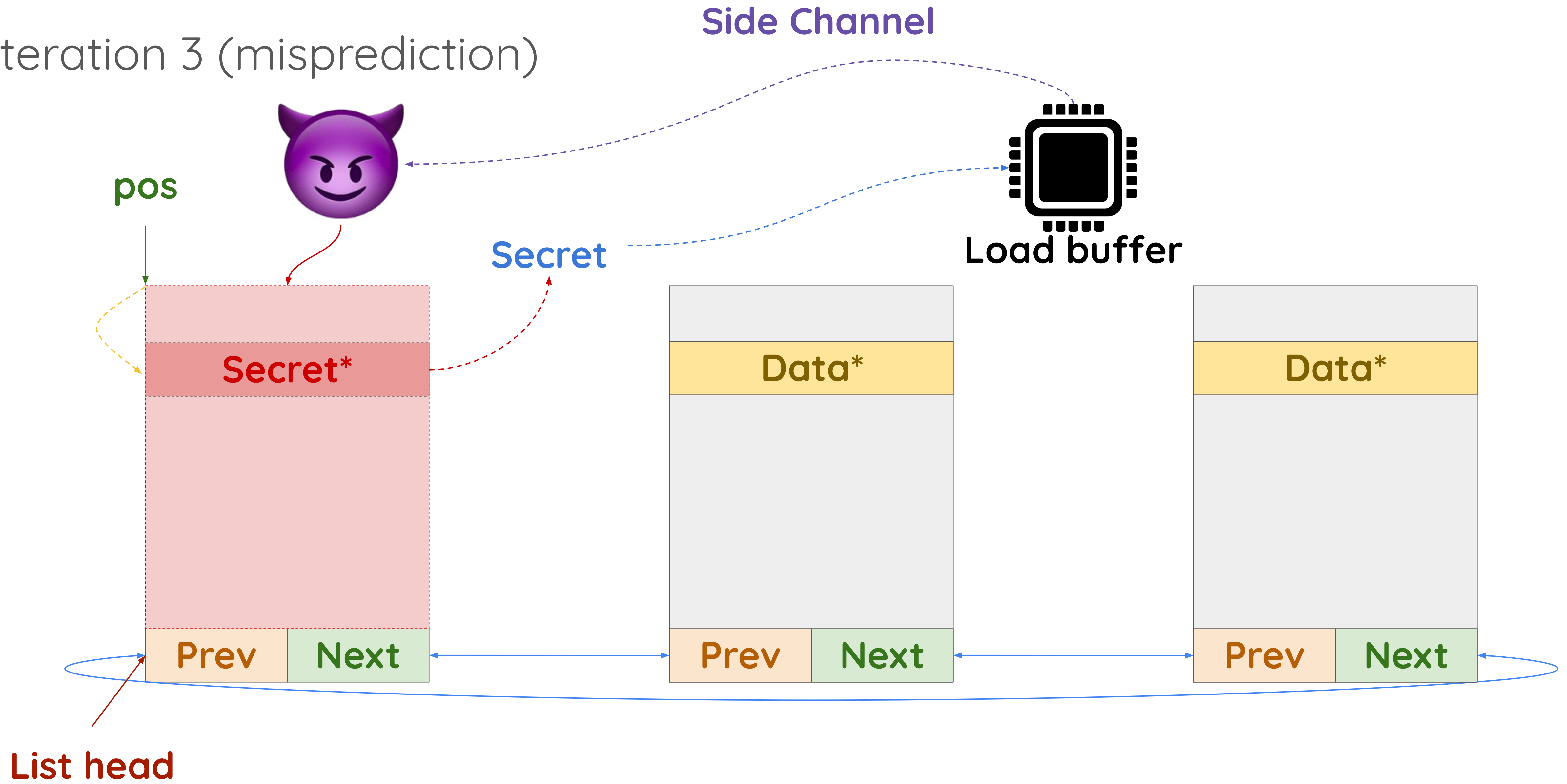
# Case study: list iterator

Iteration 3 (misprediction)



# Case study: list iterator

Iteration 3 (misprediction)





Finally, we implemented a **proof-of-concept exploit** of an instance of this gadget.

But that's just the **beginning of the story...**



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

The issue **cannot** be **solved** with a simple `array_index_nospec()`



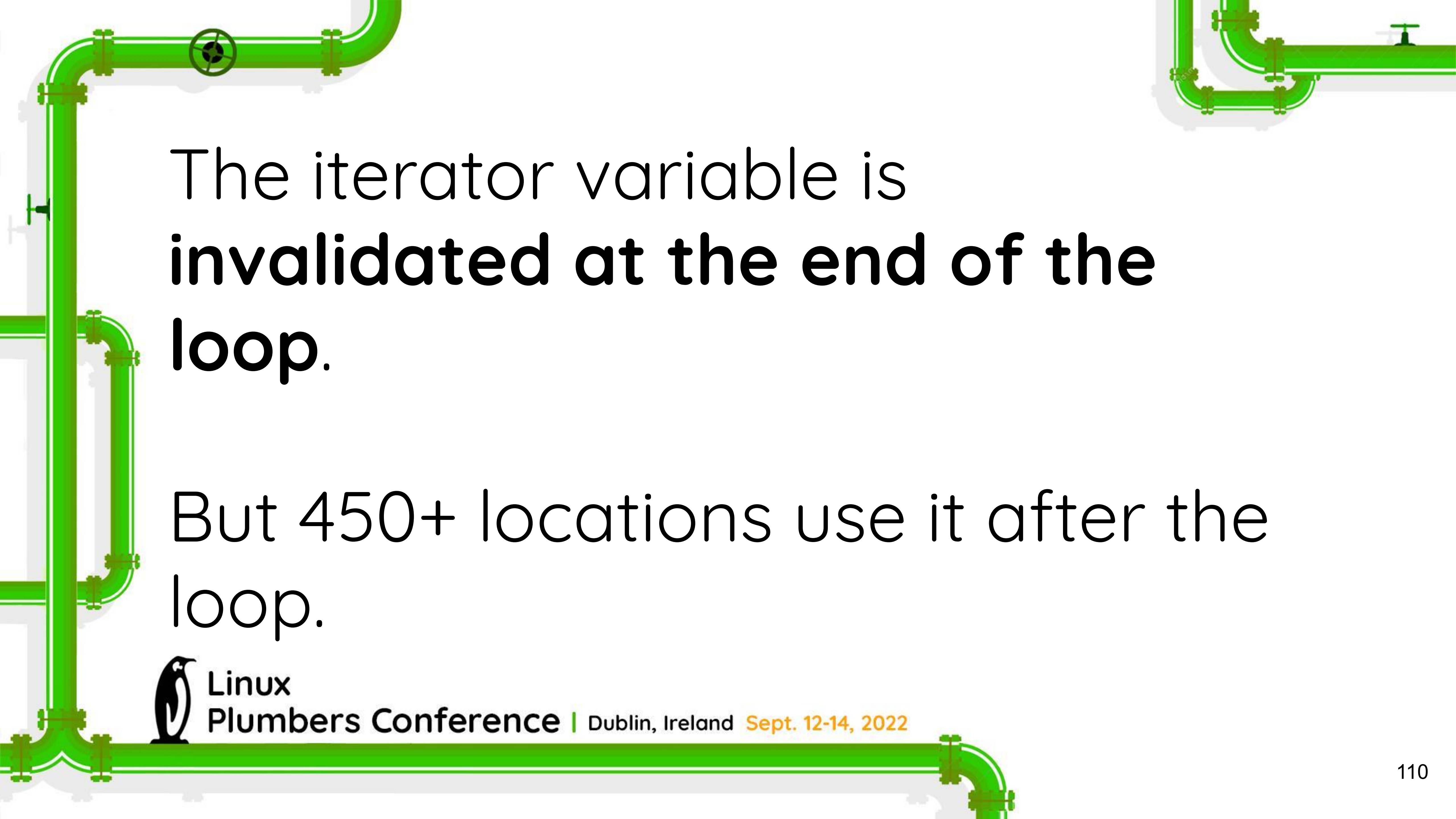
Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# How can we fix this?

```
#define list_for_each_entry(pos, head, member)
    for (pos = list_first_entry(head, typeof(*pos), member);
        ({ bool _cond = !list_entry_is_head(pos, head, member);
          pos = select_nospec(_cond, pos, NULL); _cond; });
        pos = list_next_entry(pos, member))
```





The iterator variable is  
**invalidated at the end of the  
loop.**

But 450+ locations use it after the  
loop.



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of green pipes with valves and elbows, running along the top, left, and bottom edges of the slide.

# But 450+ locations use it after the loop.

```
$ make coccicheck COCCI=scripts/coccinelle/iterators/use_after_iter.cocci  
  
./arch/arm/mach-mmp/sram.c:54:6-10: ERROR: invalid reference to the index variable of the iterator on line 48  
./arch/arm/plat-pxa/ssp.c:54:6-9: ERROR: invalid reference to the index variable of the iterator on line 44  
./arch/arm/plat-pxa/ssp.c:78:6-9: ERROR: invalid reference to the index variable of the iterator on line 68  
./block/blk-mq.c:4481:11-13: ERROR: invalid reference to the index variable of the iterator on line 4472  
./drivers/block/rbd.c:776:16-27: ERROR: invalid reference to the index variable of the iterator on line 766  
./drivers/dma/at_xdmac.c:1583:13-17: ERROR: invalid reference to the index variable of the iterator on line 1571  
...
```



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A stylized green pipe network graphic with various fittings, valves, and elbows, framing the central text. The pipes are bright green with dark green fittings and valves. The background is white.

Turns out some of them are  
**actual bugs!**



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



A decorative graphic of a green pipe network with various fittings, valves, and elbows, set against a white background. The pipes are arranged in a complex, interconnected pattern.

Let's look at **architectural** bugs  
now.

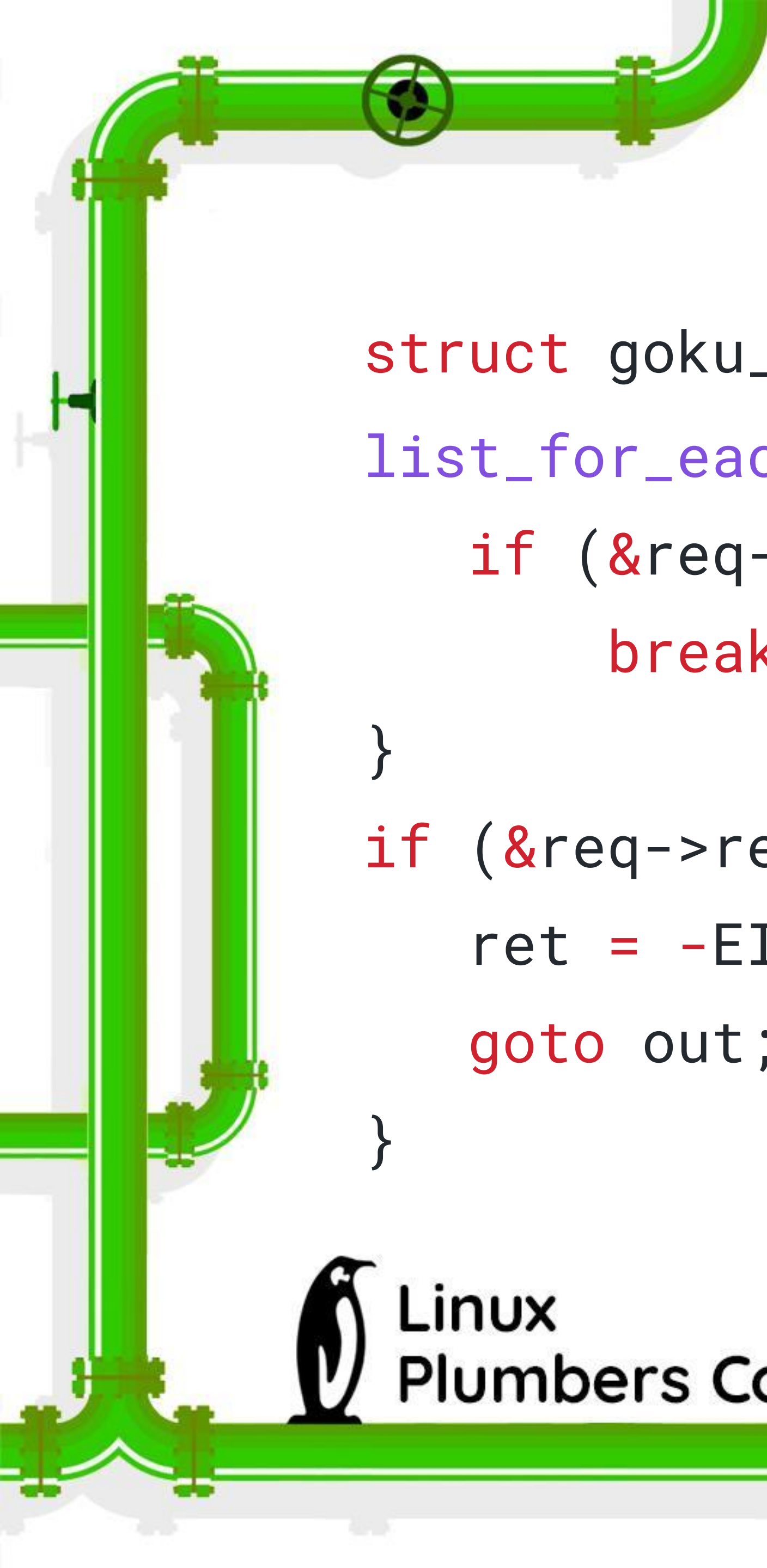
**No speculation** beyond this  
point...

```
asm("lfence");
```



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative green pipe graphic runs vertically on the left side of the slide, with various fittings and valves. It connects to a horizontal pipe at the bottom.

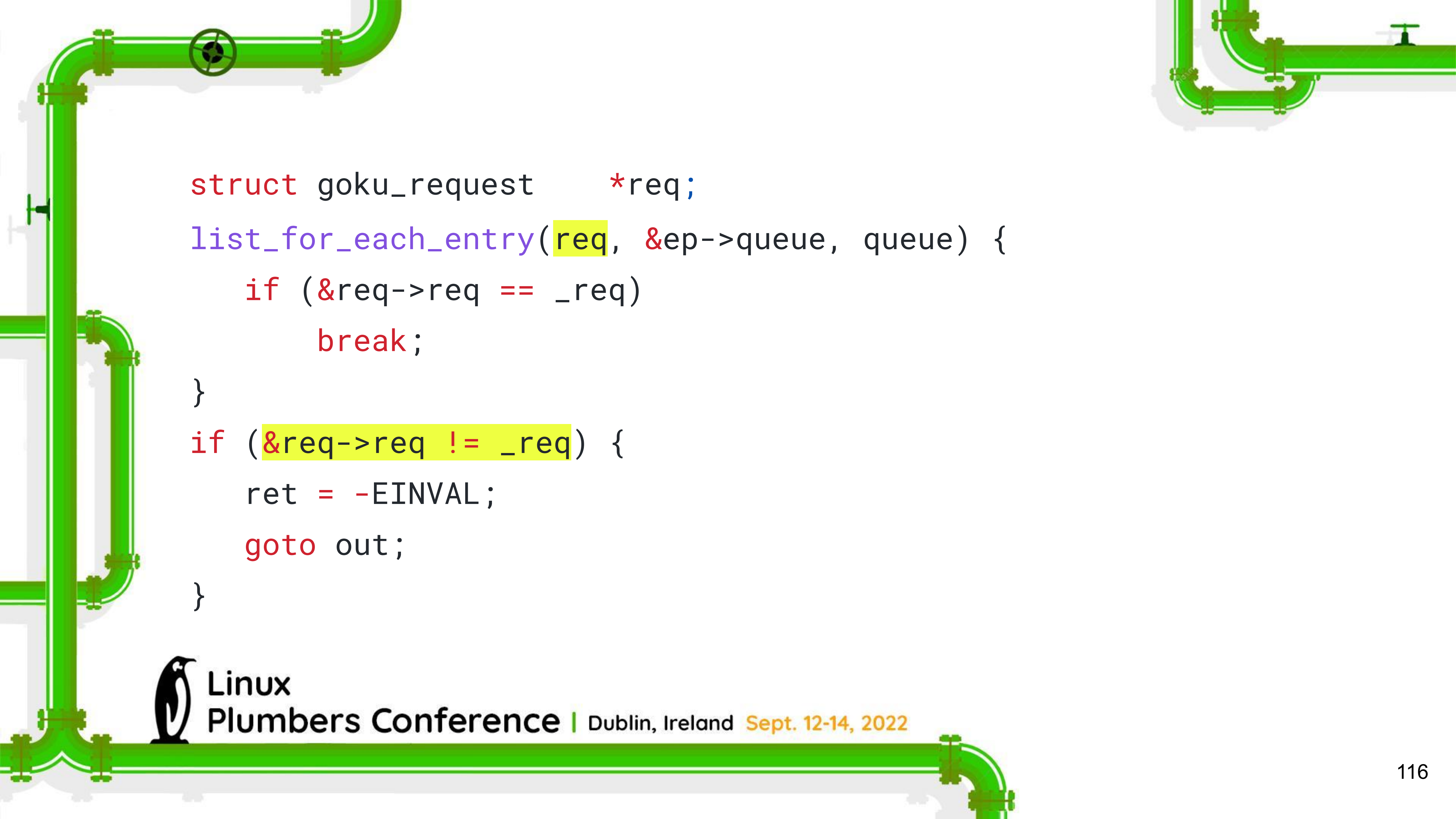
```
struct goku_request *req;
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}
```



# It **looks** safe, right?

```
struct goku_request *req;
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}
```



A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

```
struct goku_request *req;
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}
```

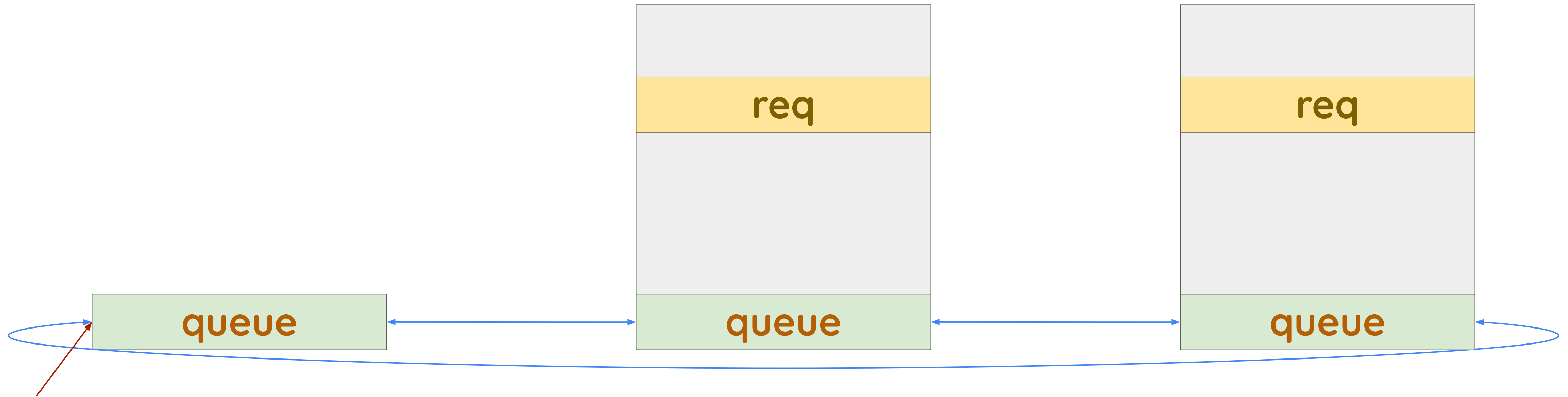


# Does it still look safe?

```
struct goku_request *req;
list_for_each_entry(req, &ep->queue, queue) {
    if (&req->req == _req)
        break;
}
if (&req->req != _req) {
    ret = -EINVAL;
    goto out;
}
```



# Case study

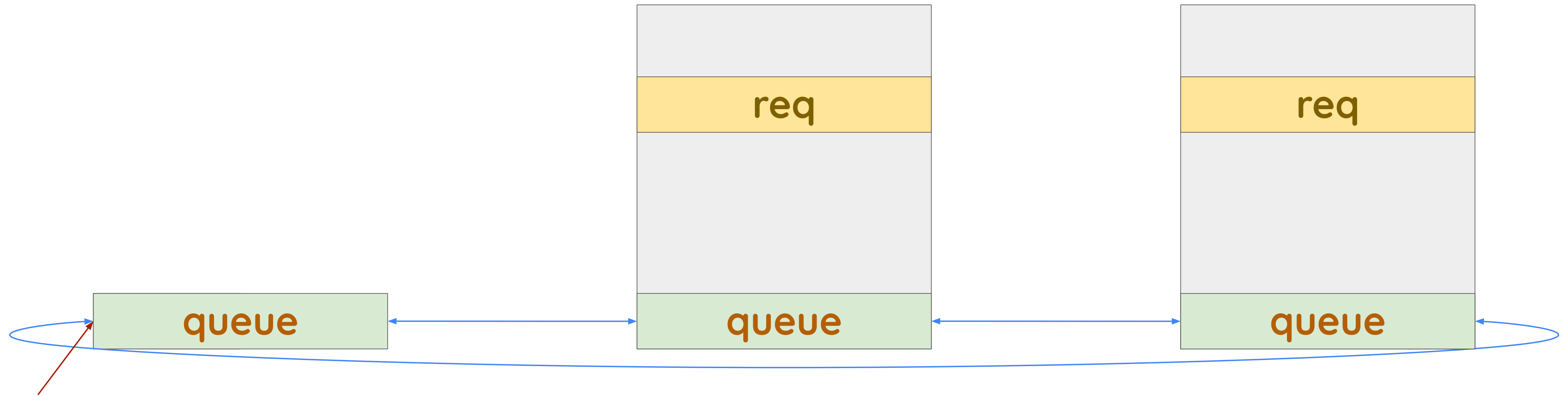


List head

# Case study

Iteration 1

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```

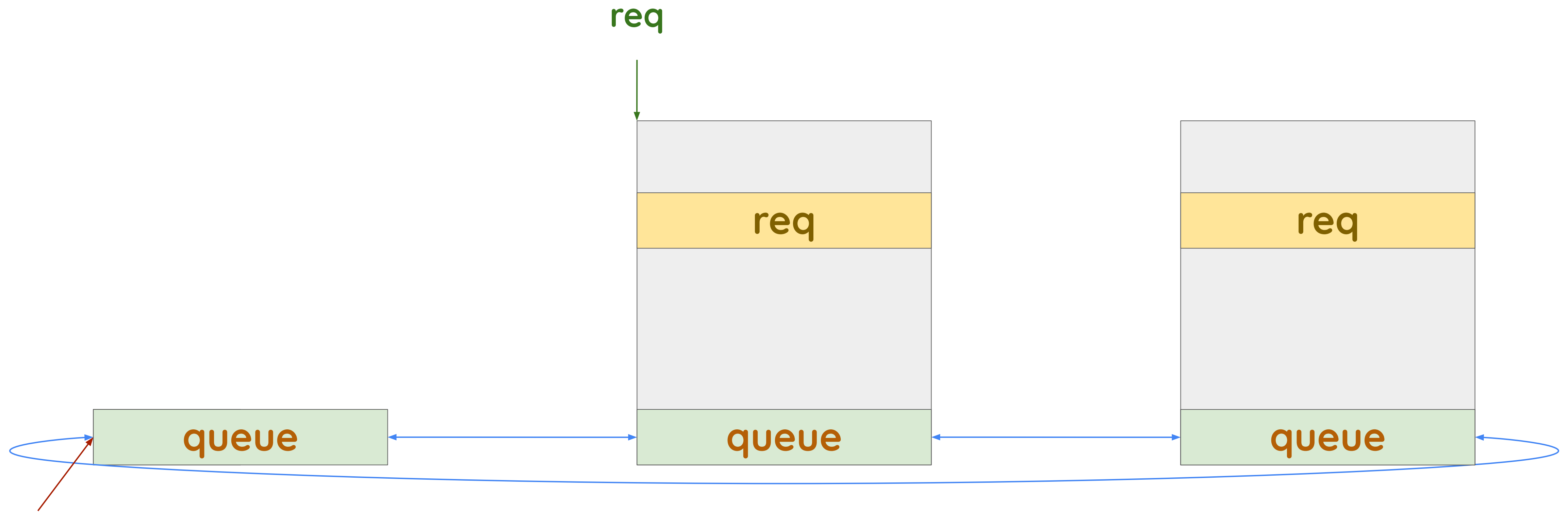


List head

# Case study

Iteration 1

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```



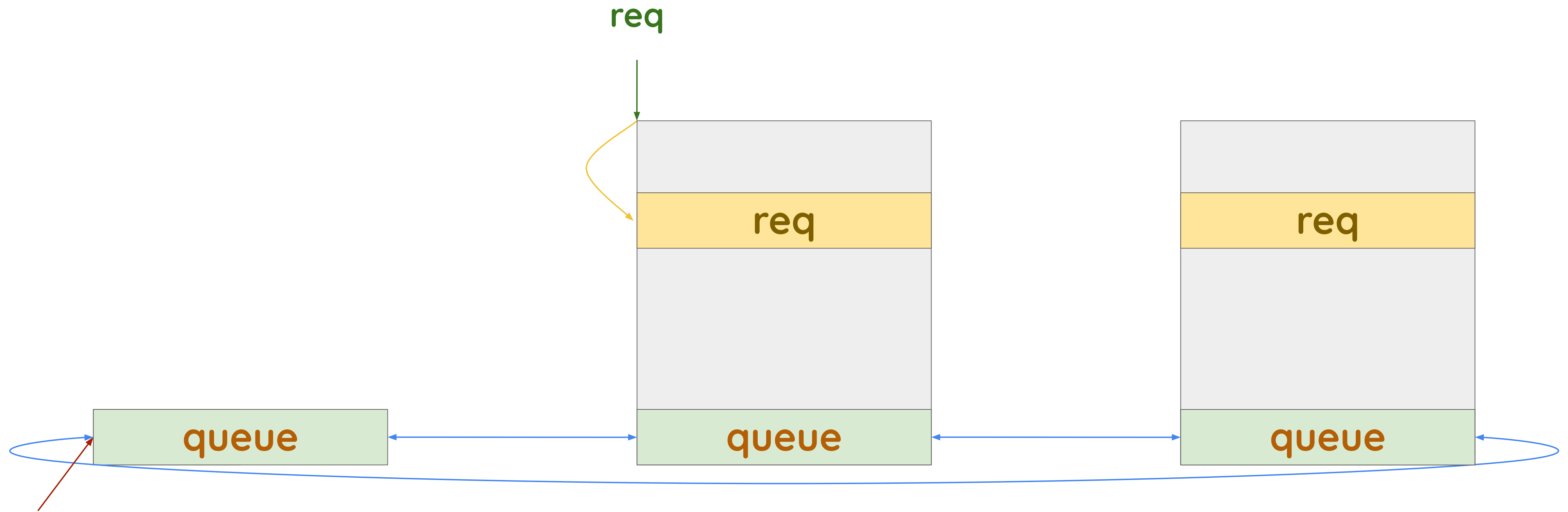
List head



# Case study

Iteration 1

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```

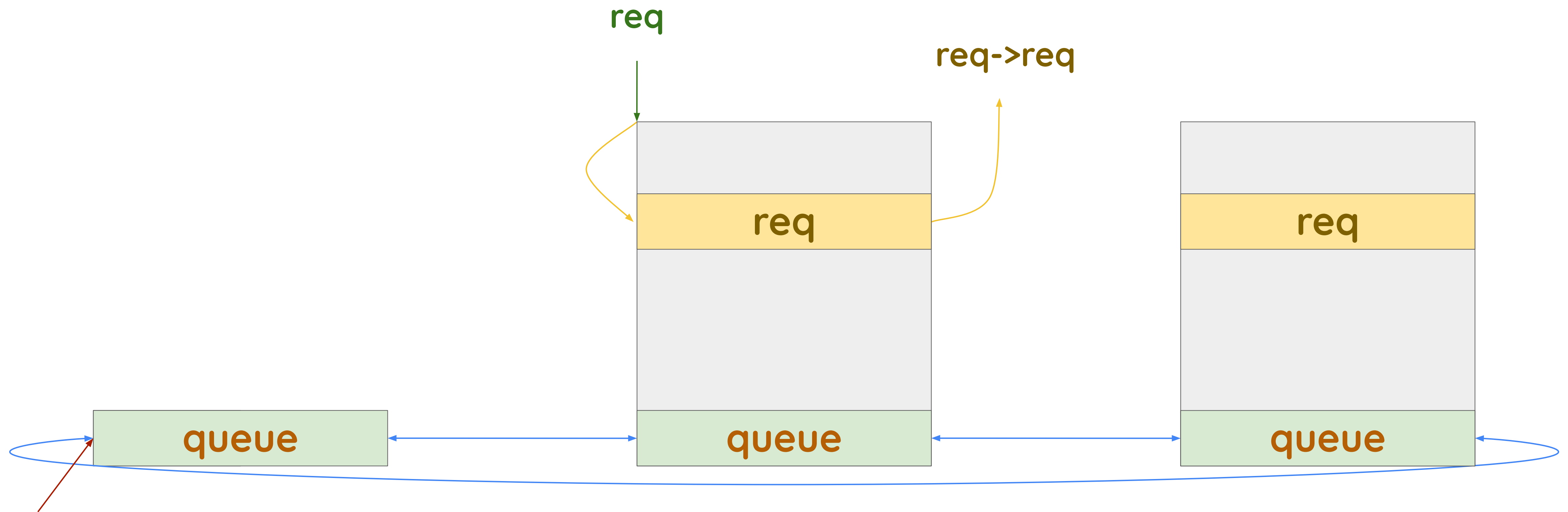


List head

# Case study

Iteration 1

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```

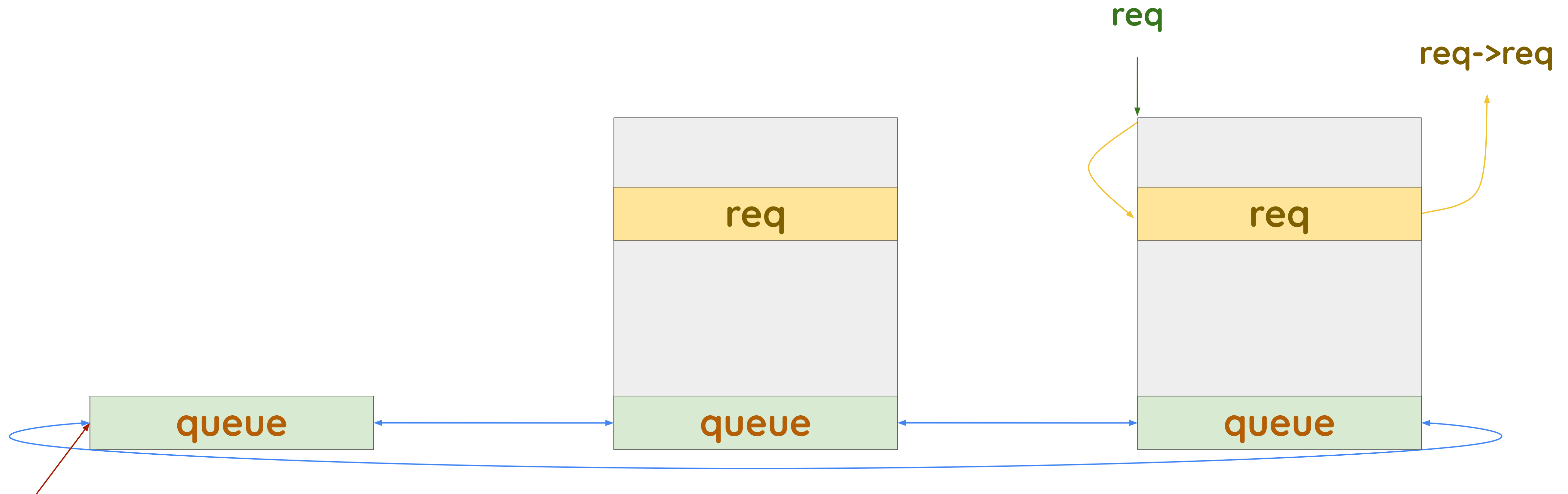


List head

# Case study

Iteration 2

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```

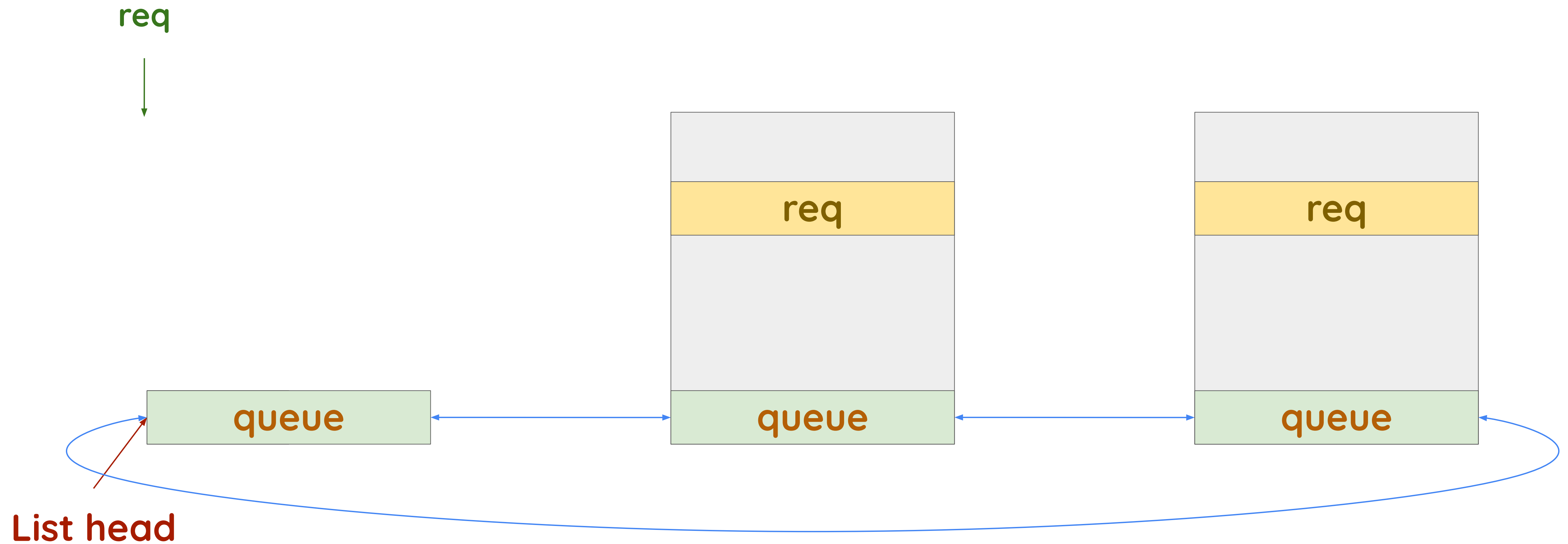


List head

# Case study

After loop

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```



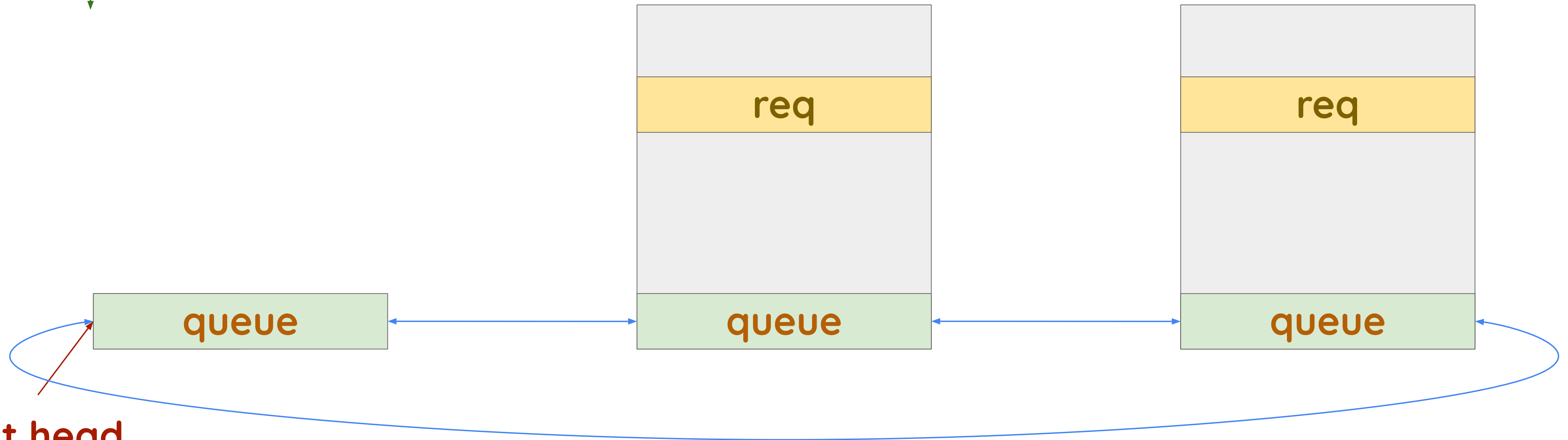
# Case study

After loop



```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```

req

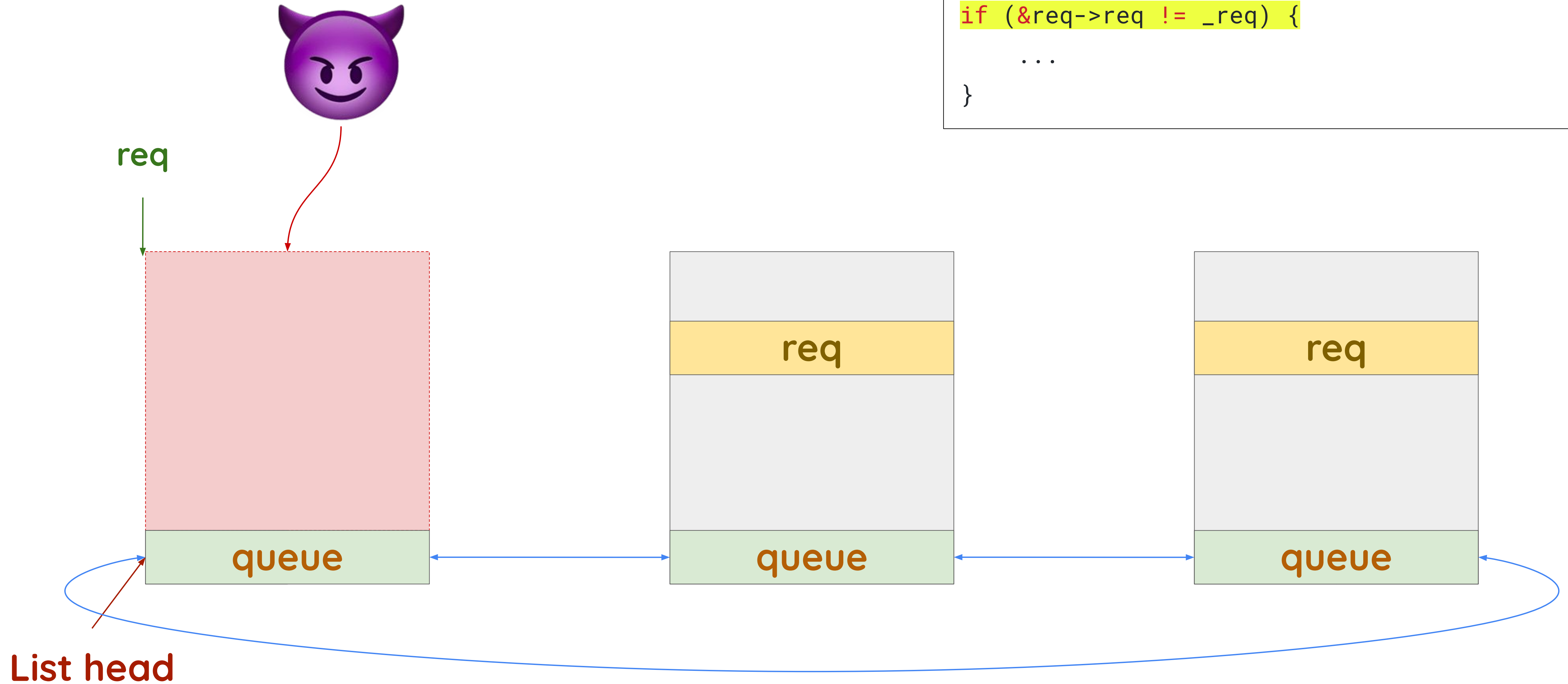


List head

# Case study

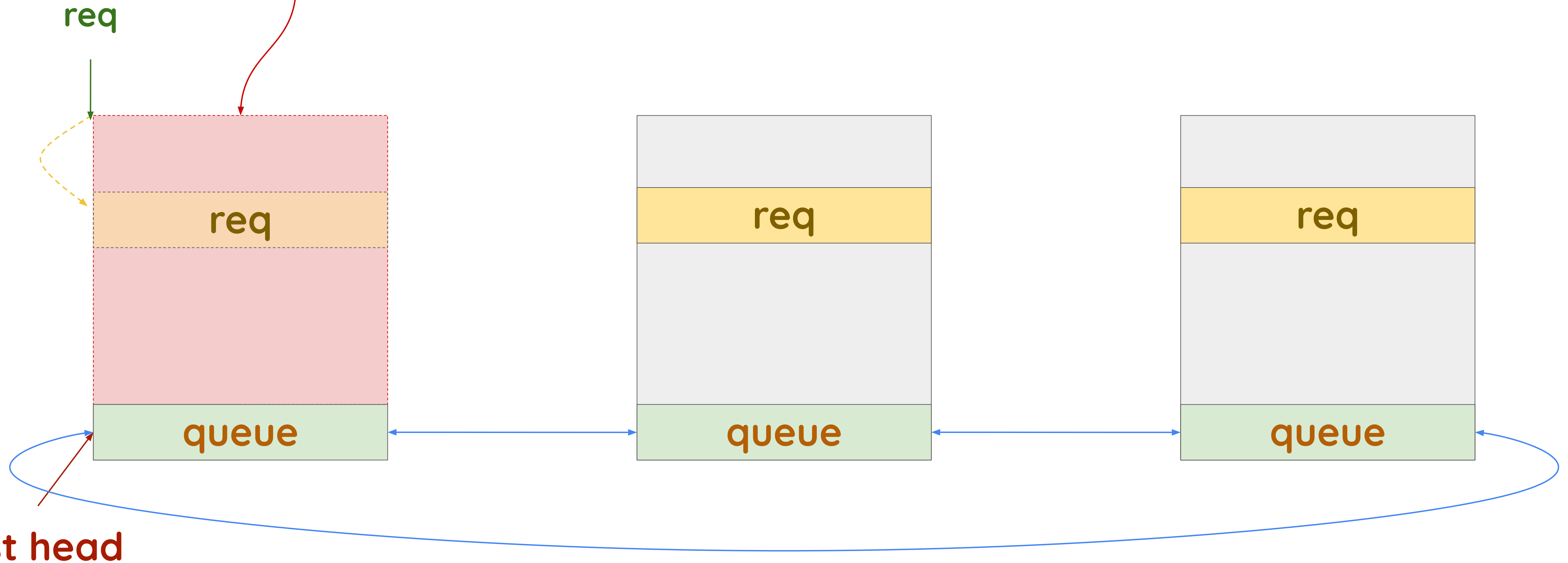
After loop

```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```



# Case study

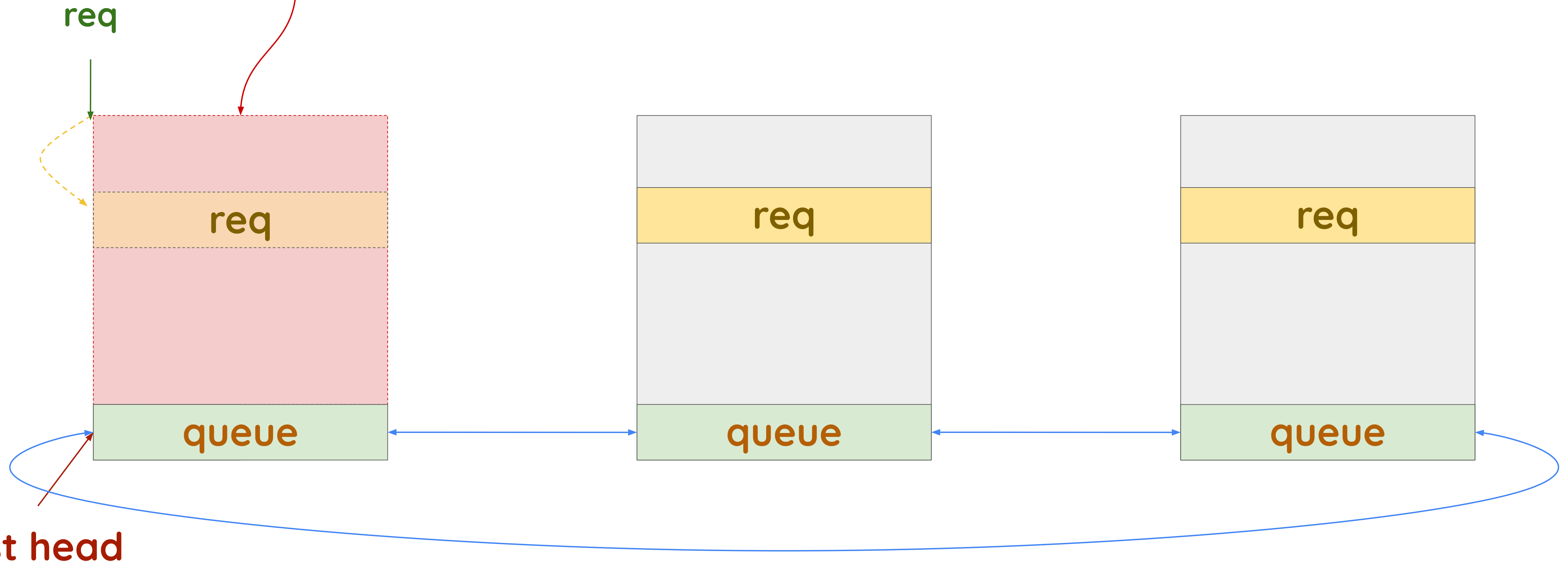
After loop



```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```

# Case study

After loop



```
list_for_each_entry(req, &ep->queue, queue) {  
    if (&req->req == _req)  
        break;  
}  
if (&req->req != _req) {  
    ...  
}
```



# Type Confusion in C



Linux  
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# Type Confusion in C

- `container_of()` is performed on the `list_head` which is **not** contained in a struct



# Type Confusion in C

- `container_of()` is performed on the `list_head` which is **not contained** in a struct
- it resembles an **invalid downcast** in object oriented programming



# Type Confusion in C

- `container_of()` is performed on the `list_head` which is **not contained** in a struct
- it resembles an **invalid downcast** in object oriented programming
- that's why we call it a **type confusion**



# Quotes from Linus



Linux  
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# Quotes from Linus

Make the rule be "you never use the iterator outside the loop".



Linux  
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# Quotes from Linus

Make the rule be "you never use the iterator outside the loop".

The whole reason this [...] bug can happen is that we [...] didn't have C99-style "declare variables in loops".



# Quotes from Linus

Make the rule be "you never use the iterator outside the loop".

The whole reason this [...] bug can happen is that we [...] didn't have C99-style "declare variables in loops".

"we could finally start using variable declarations in for-statements"

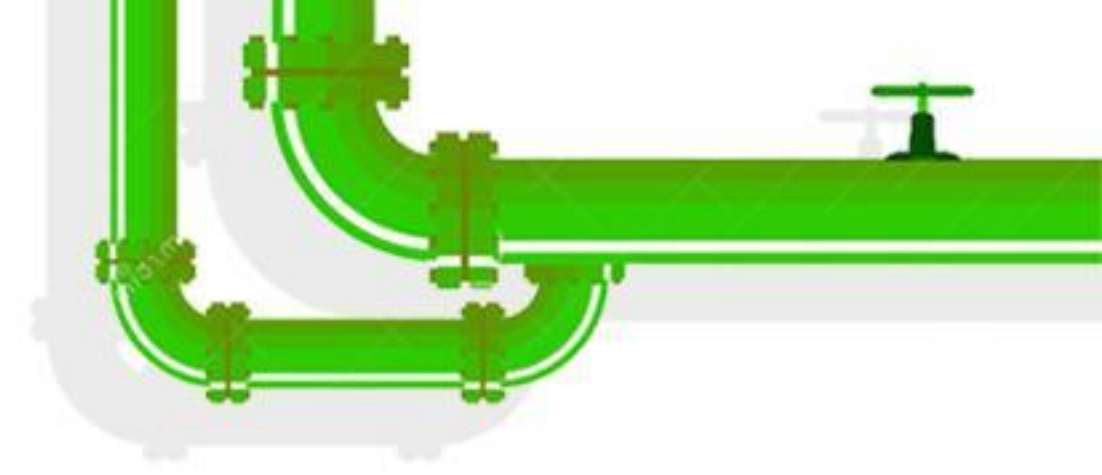
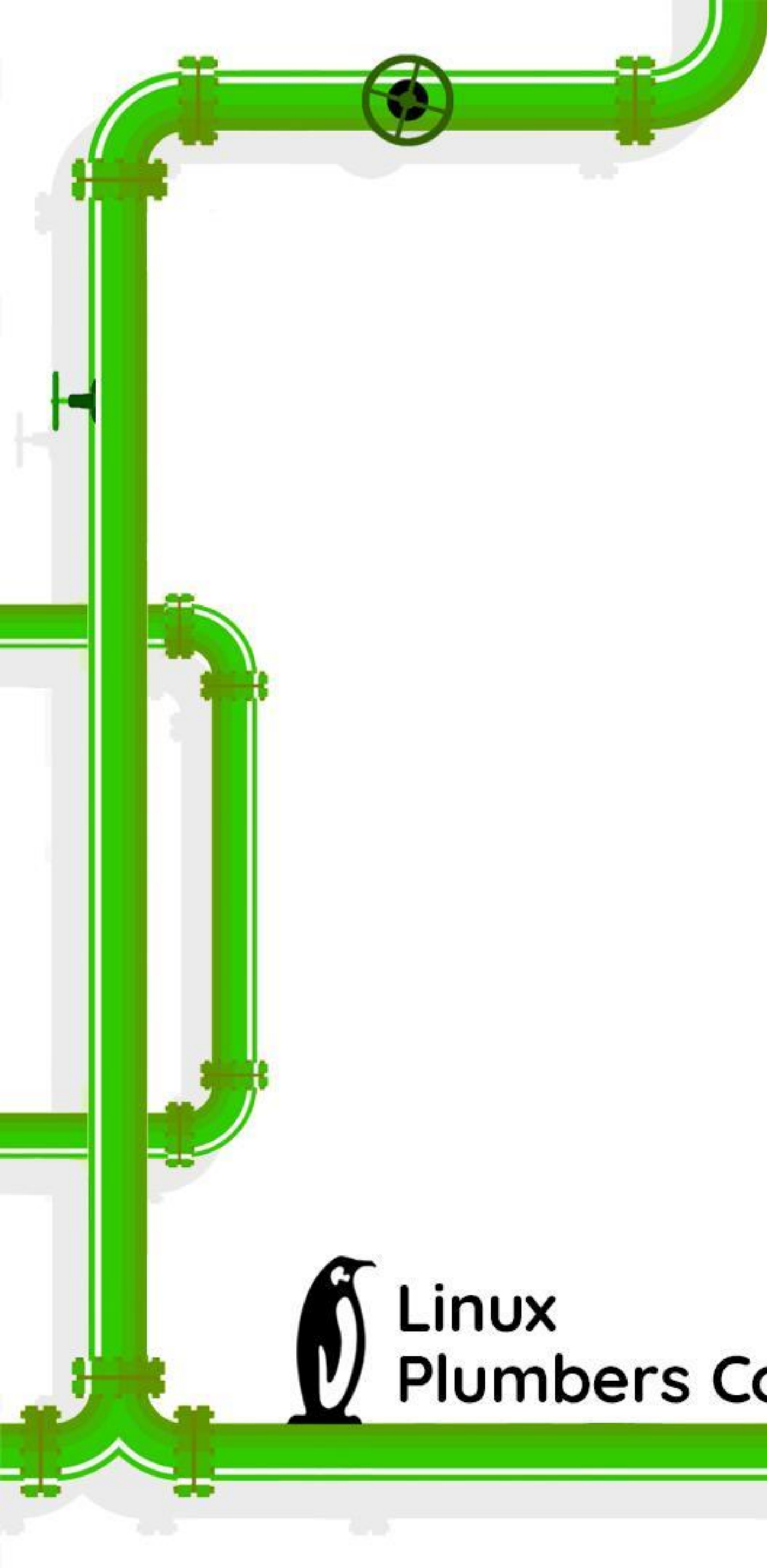


Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



# The correct way



Linux  
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

# The correct way

```
struct goku_request *req = NULL, *iter;  
list_for_each_entry(iter, &ep->queue, queue) {  
    if (&iter->req == _req) {  
        req = iter;  
        break;  
    }  
}  
if (!req) {  
    ret = -EINVAL;  
    goto out;  
}
```





# Moving the kernel to modern C

By **Jonathan Corbet**  
February 24, 2022

Despite its generally fast-moving nature, the kernel project relies on a number of old tools. While critics like to focus on the community's extensive use of email, a possibly more significant anachronism is the use of the 1989 version of the C language standard for kernel code — a standard that was codified before the kernel project even began over 30 years ago. It is looking like that longstanding practice could be coming to an end as soon as the 5.18 kernel, which can be expected in May of this year.

## Linked-list concerns

The discussion started with [this patch series](#) from Jakob Koschel, who is trying to prevent speculative-execution



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

<https://lwn.net/Articles/885941/>

A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the text on the slide.

Submitting patches is **fun but very time intensive.**

Around 80 patches have been merged so far.



Linux

Plumbers Conference | Dublin, Ireland **Sept. 12-14, 2022**

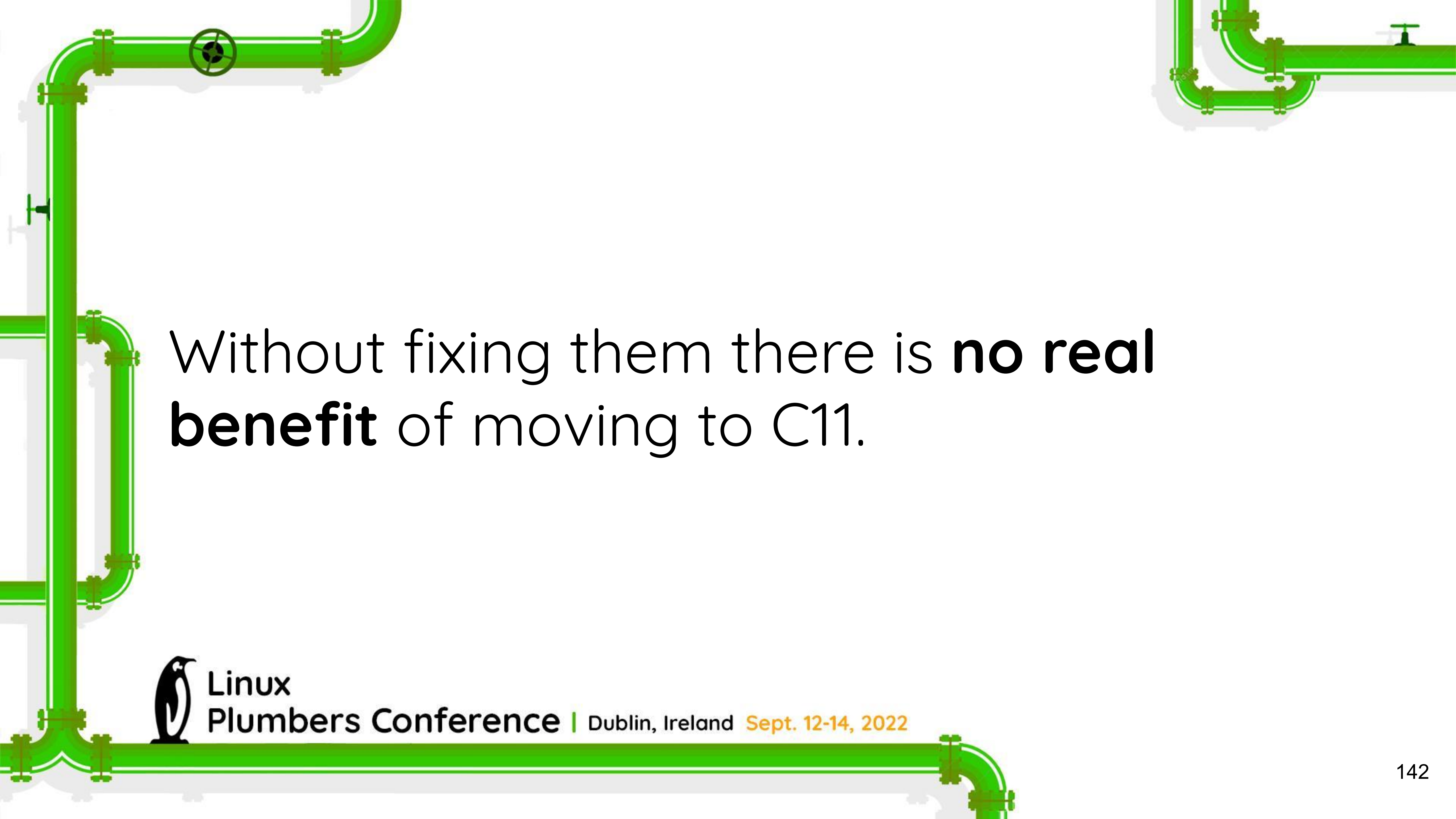


~300 locations still use the list iterator  
**after the loop!**

Patching has to be done **one by one.**



Linux  
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A stylized green pipe network graphic with various fittings, valves, and elbows, framing the central text. The pipes are bright green with dark green fittings and valves. The background is white.

Without fixing them there is **no real benefit** of moving to C11.



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



Treewide changesets is **a tricky entry** to submitting patches.

Same bugs will need **different fixes** depending on the maintainer.



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

Knowing how to split them into pieces is **difficult.**

Different subsystems have **different rules.**



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



A decorative graphic of a green pipe network with various fittings, valves, and elbows, framing the central text.

Big shoutout to Mike Rapoport for his massive help!



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022

A decorative graphic of a green pipe network with various fittings, valves, and elbows, set against a white background. The pipes are arranged in a complex, interconnected pattern, with some sections being thicker than others. The graphic is positioned around the edges of the slide, framing the central text.

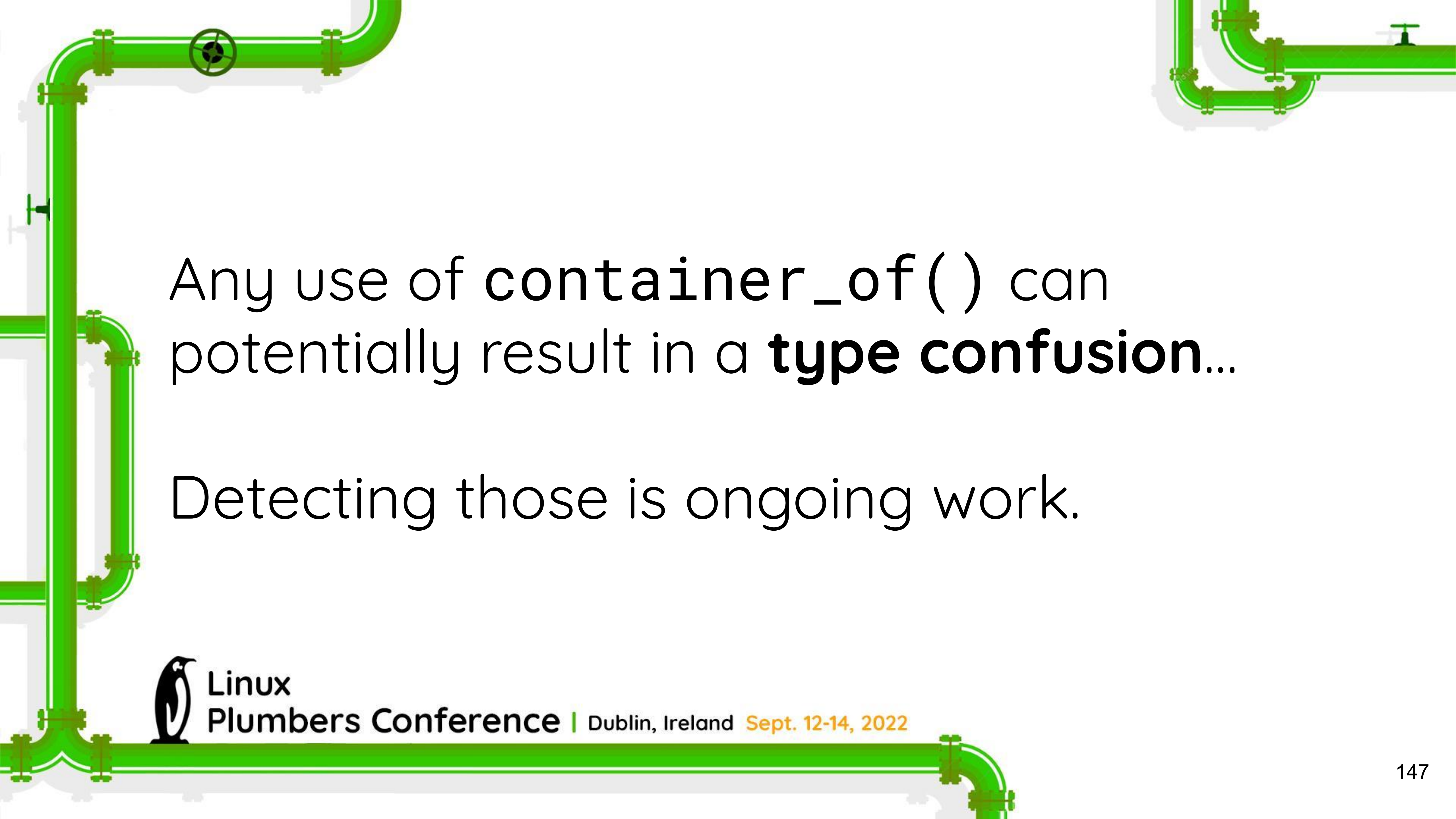
There might be **more type confusions**  
in the kernel.

Maybe it's time for a **new scanner...**



Linux

Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



Any use of `container_of()` can potentially result in a **type confusion**...

Detecting those is ongoing work.



A decorative graphic of a green pipe network with various fittings, elbows, and valves, running along the top and left sides of the slide.

We've started with building a **speculative gadget scanner...**

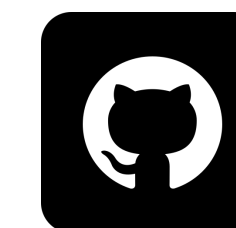
... ended up with **real type confusion bugs**

and caused the kernel to move to a **more modern version of C.**

Thank you!



Linux  
Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022



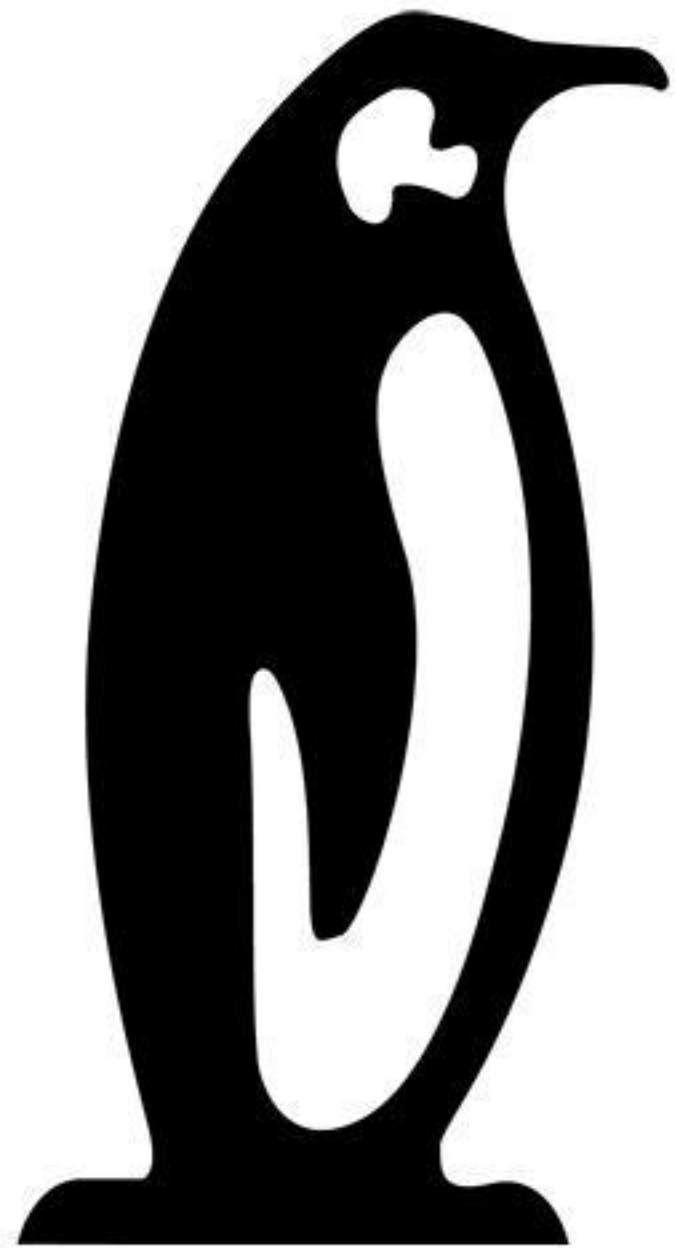
Jakob-Koschel



@JakobKoschel



j.koschel AT vu DOT nl



# Linux Plumbers Conference

Dublin, Ireland **September 12-14, 2022**