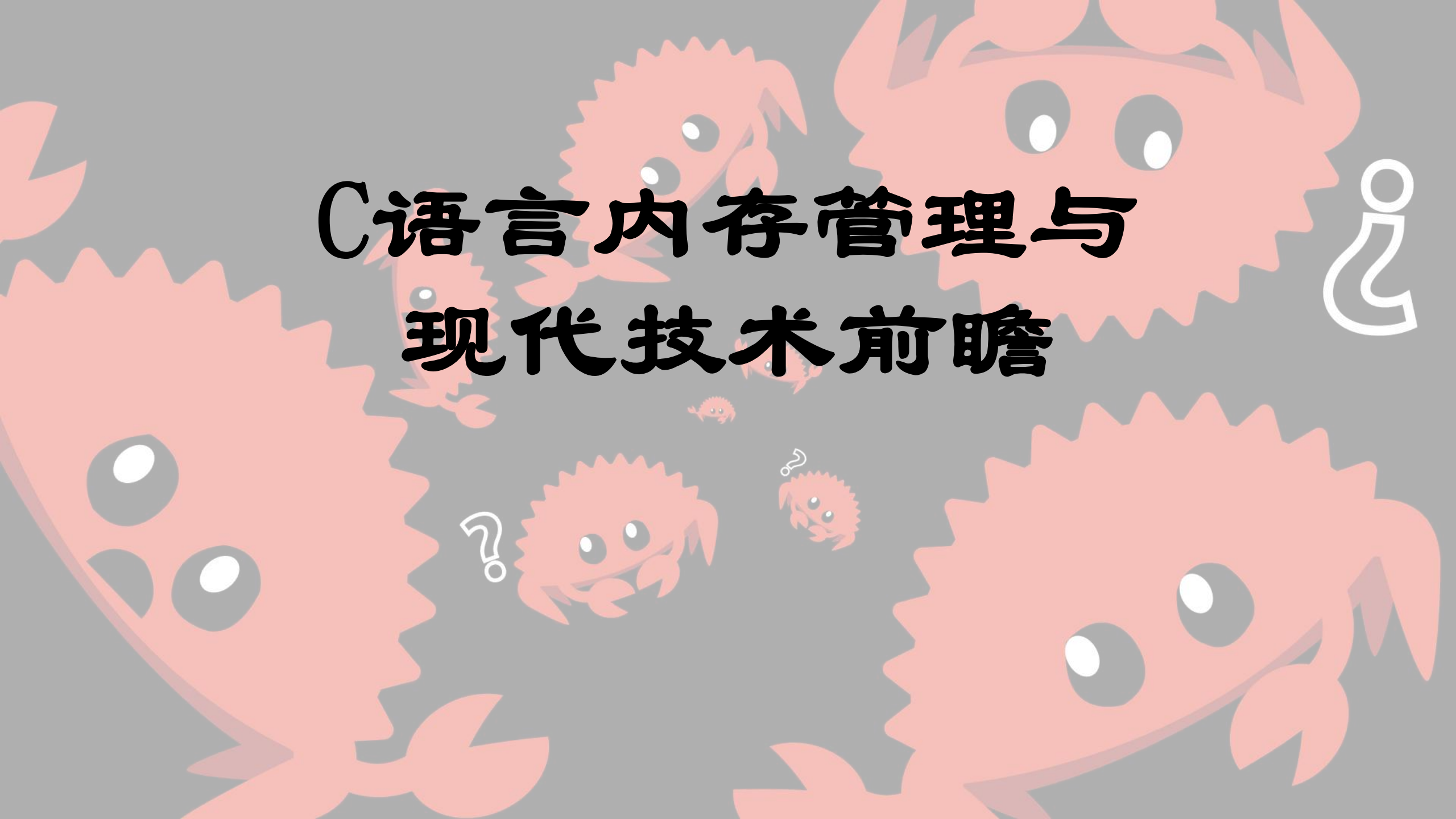
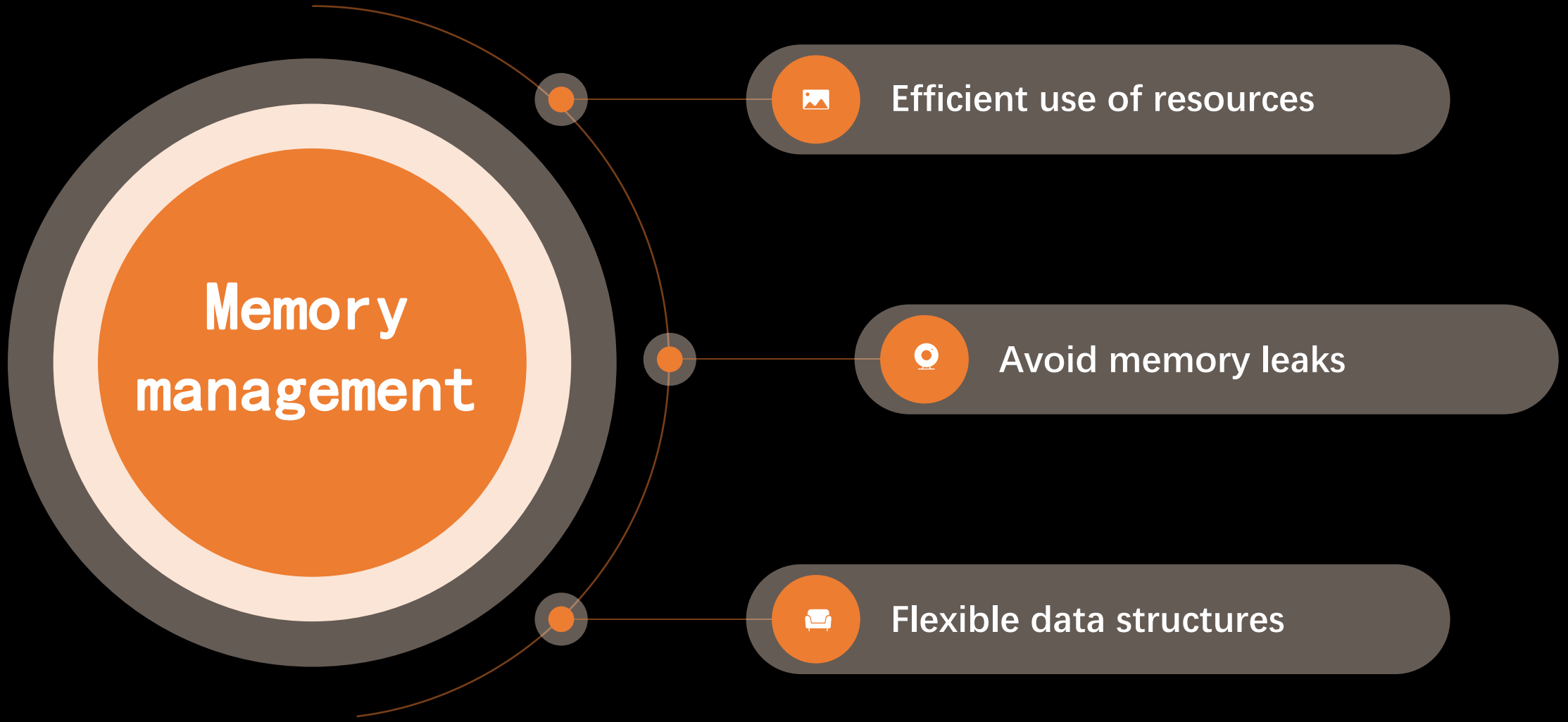
C语言内存管理与现代技术前瞻



Why manage memory for a program?



The necessity of memory management

Efficient use of resources

- 确保程序充分利用系统内存资源，避免内存浪费。通过动态分配和释放内存，程序可以根据实际需要动态调整内存的使用情况。**这对于嵌入式系统等对资源要求极高的应用场景十分重要**

Avoid memory leaks

- 帮助防止内存泄漏问题。内存泄漏是指程序在使用完内存后未正确释放，导致系统中的可用内存逐渐减少。通过及时释放不再需要的内存，可以避免这类问题，提高程序的稳定性和性能。

Flexible data structures

- 动态内存管理使得程序能够在运行时创建和调整数据结构，而不受静态内存分配的限制。这样可以更灵活地处理不同大小和类型的数据，适应不同的运行条件和需求。

Advantages of using C for memory management

1

Manual Memory Management

使用malloc、free、calloc、realloc等函数，可以灵活地进行内存分配和释放，确保资源的高效利用。

2

Customized Data Structures

通过指针方式精确访问内存，从而使程序员可以定制数据结构和访问方式，使程序运行效率提升

3

High Performance

C语言控制粒度较细，可以根据需求灵活管理内存分配，从而给了开发者更多的性能优化空间

4

Low-Level System Programming

C语言控制内存的方式使其更加适合系统级应用对内存的管理

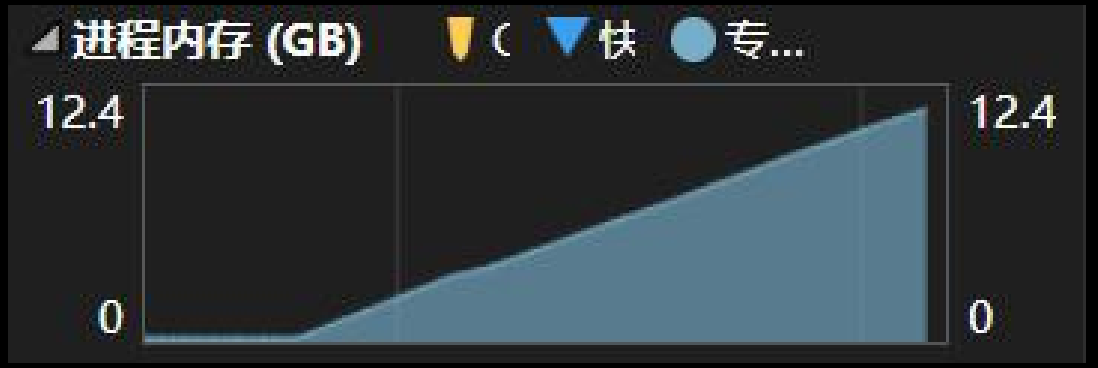
Dangers of improper memory management

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

int main() {
    int *p = NULL;
    int i = 0;

    while (1) {
        p = (int *)malloc(200*sizeof(int));
        if (p == NULL)
            return 0;
        i++;
        printf("malloc success %d\n", i);
    }

    return 0;
}
```

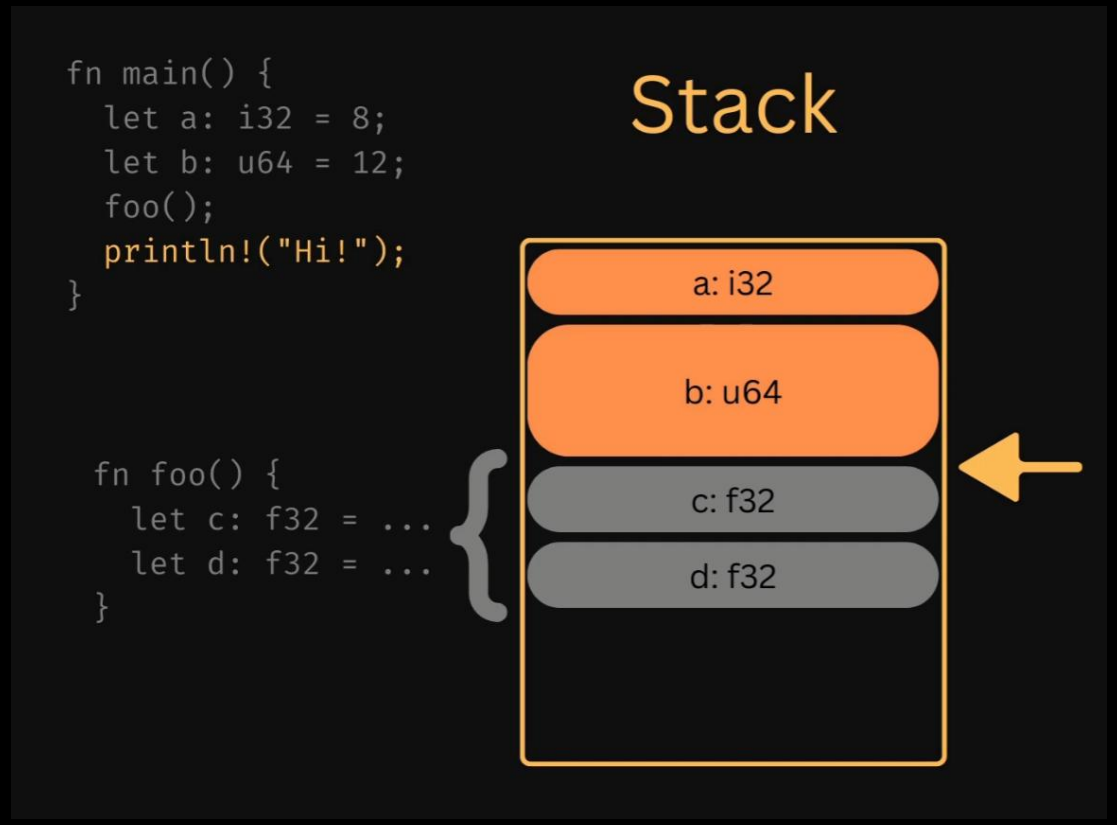


Basic memory organization

Stack

局部变量内存的自动栈分配

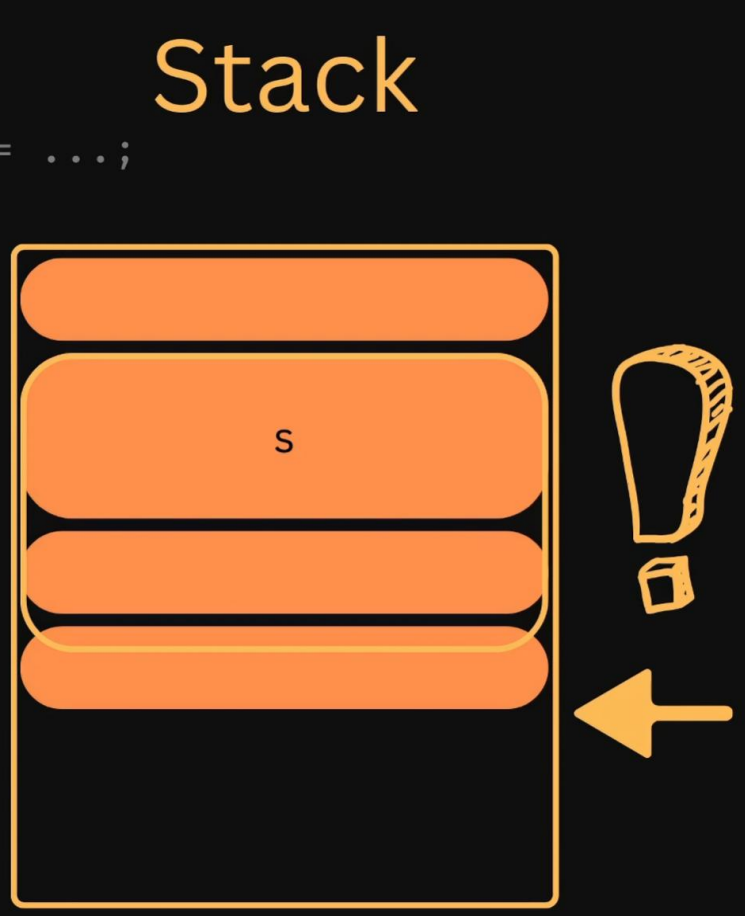
```c
void func(void)
{
    int q;
    int p[100000];
    ``````
    ~~~~~~~~~~~~
}
int main(void)
{
    int n;
    int i[100000];
    func();
    ````
    ~~~~~~~~
    return 0;
}
```




栈帧
```

# Limitations of the Stack

```
fn main() {
  ...
  let s: DynamicStruct = ...;
  ...
  s += ...;
}
```



# Heap

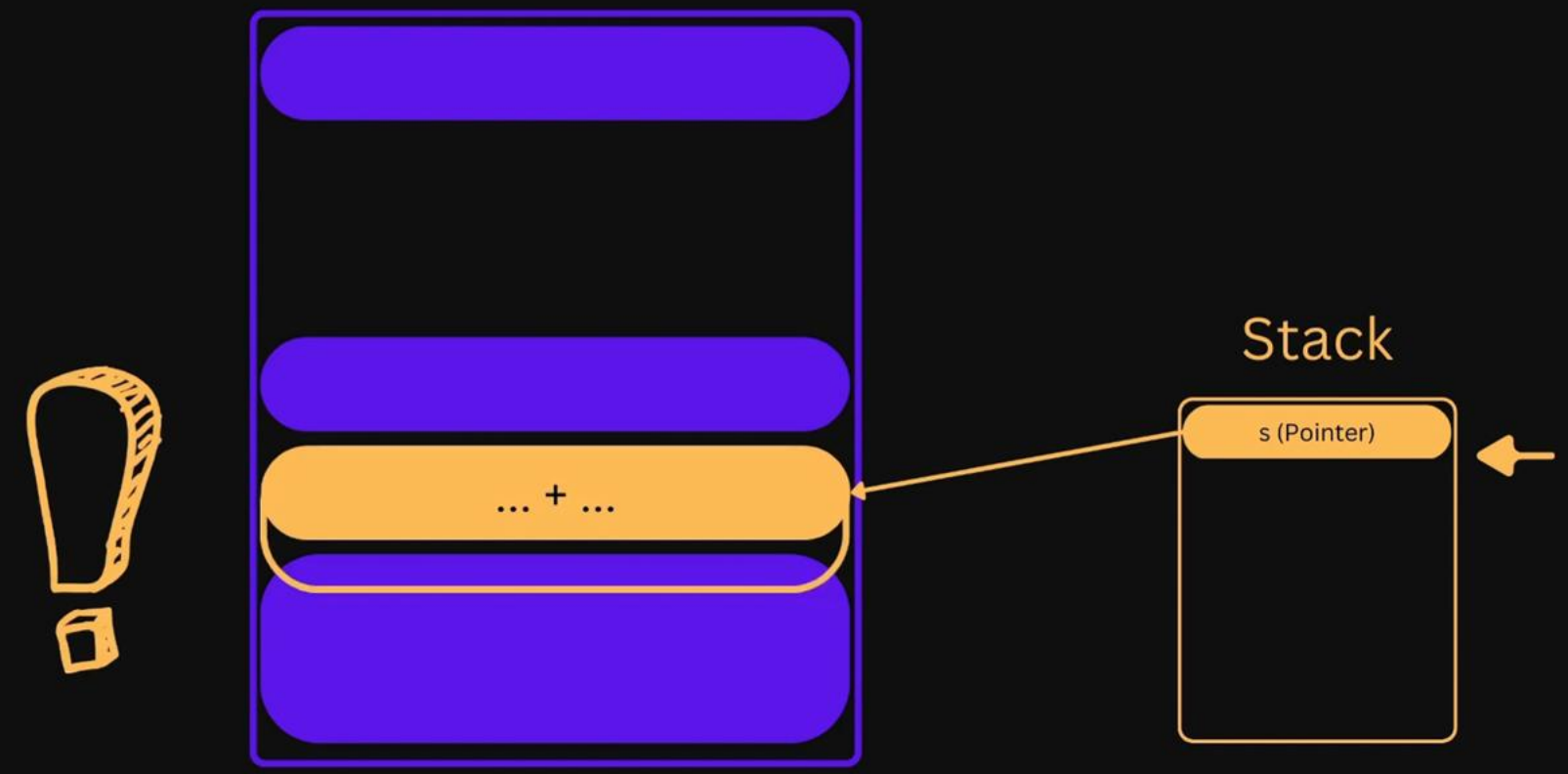
有了动态管理内存需求，就有了堆

在内存中开辟一大块空间来存储数据

堆中的内存可按需释放

# Stack & Heap

```
fn main() {
  let s: DynamicStruct = ...;
  s += ...;
}
```



# Limitations of the Heap

堆中内存碎片化会导致内存浪费和内存读取效率低下。大大浪费计算机资源，与内存管理原则相悖

# Use data structures for memory management

- 链表
- 二叉树
- 图
- 哈希表
- 队列
- 栈

内存管理的难点是如何释放垃圾内存

# Developments in memory management technology

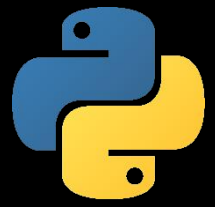
01 Pointer

02 Garbage Collection(GC)

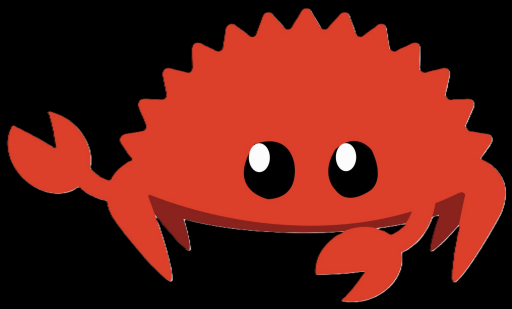
03 Ownership

C/C++

JAVA Python Go C#

Rust



Ferris可爱喵
关注Ferris谢谢喵

# Rust：拥抱所有权和借用的革命性思维方式

# Ownership rules


01 Rust 中的每个值都有一个变量，称为其所有者。

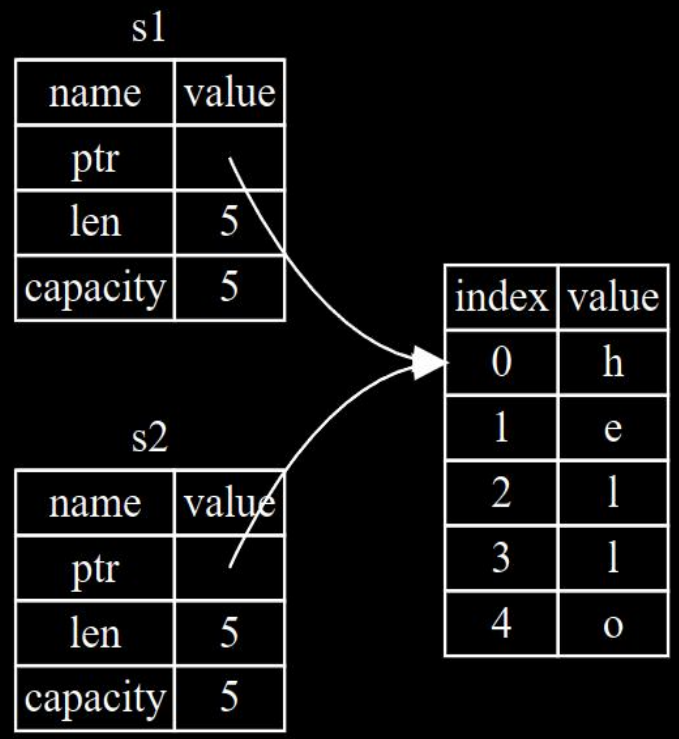
02 一次只能有一个所有者

03 当所有者不在程序运行范围时，该值将被删除。

# Ownership move

\Error!

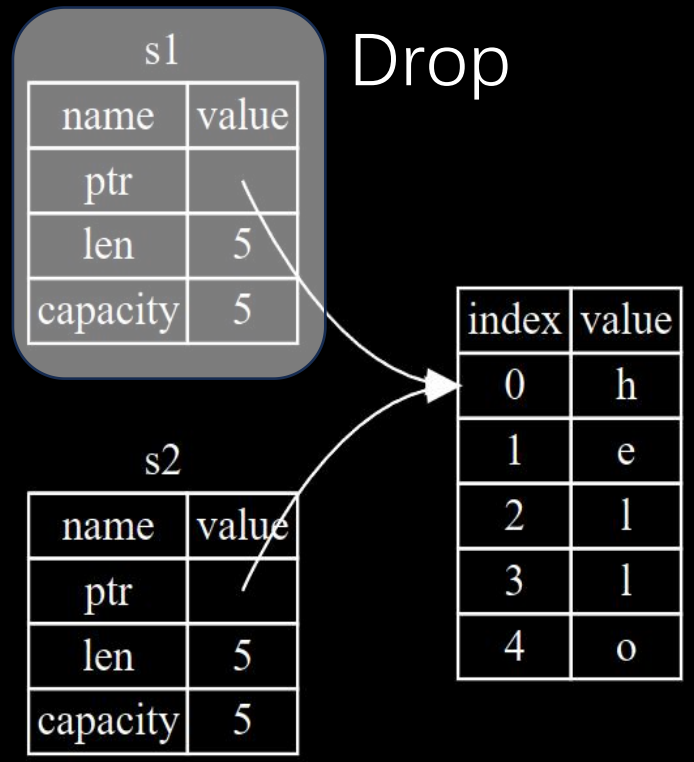
```rust
fn main() {
    let s1 : String = String::from( s: "hello");
    let s2 : String = s1;
    println!("{}, world!", s1);
}
```




```
error[E0382]: borrow of moved value: `s1`
 --> src\main.rs:4:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
```

# Ownership move

Correct!




```rust
fn main() {
    let s1 :String = String::from( s: "hello");// s1 comes into scope
    let s2 :String = s1;// s1 is moved into s2
    println!("{}, world!", s2);// s2 is printed
}
```

```
C:/Users/Plato/.cargo/bin/cargo.exe run --color=always --package untitled2 --bin untitled2
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\untitled2.exe`
hello, world!
```

# Data Copy trait

```rust
fn main() {
    let x: i32 = 5;
    let y: i32 = x;
    println!("x = {}, y = {}", x, y);
}
```

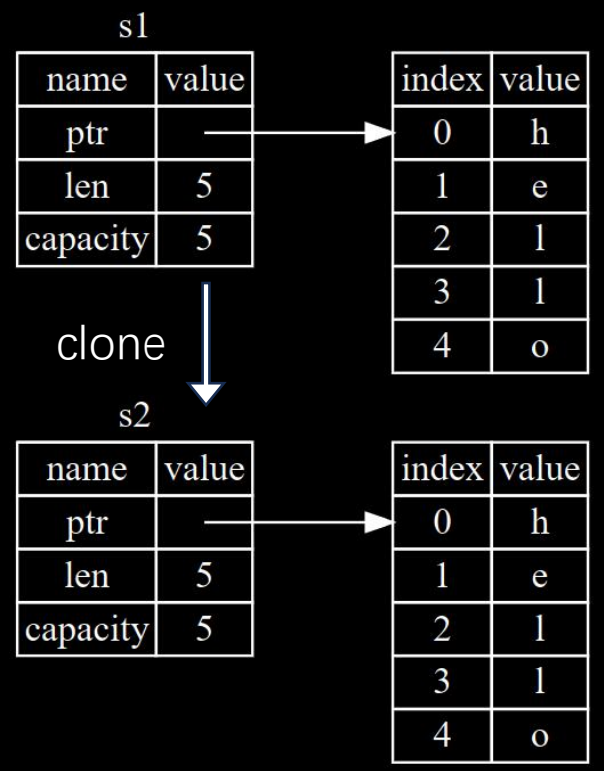
那么哪些类型实现了 `Copy` trait 呢？你可以查看给定类型的文档来确认，不过作为一个通用的规则，任何一组简单标量值的组合都可以实现 `Copy`，任何不需要分配内存或某种形式资源的类型都可以实现 `Copy`。如下是一些 `Copy` 的类型：

- 所有整数类型，比如 `u32`。
- 布尔类型，`bool`，它的值是 `true` 和 `false`。
- 所有浮点数类型，比如 `f64`。
- 字符类型，`char`。
- 元组，当且仅当其包含的类型也都实现 `Copy` 的时候。比如，`(i32, i32)` 实现了 `Copy`，但 `(i32, String)` 就没有。

# Data clone

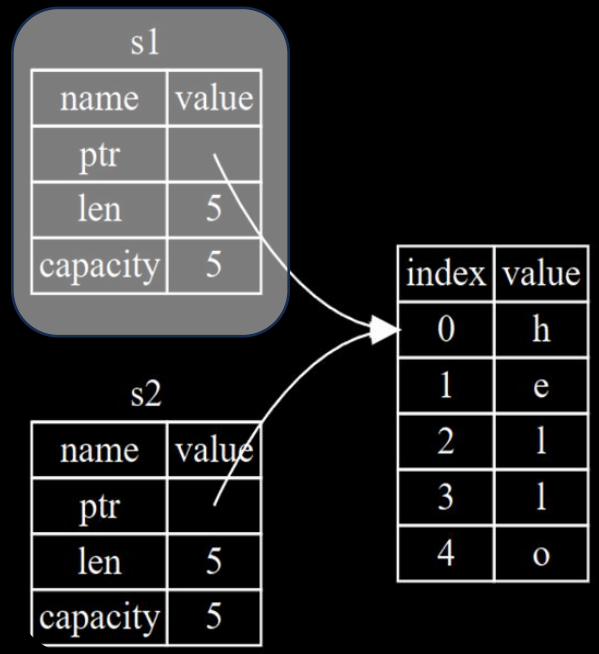
```rust
fn main() {
    let s1 : String = String::from( s: "hello");
    let s2 : String = s1.clone();

    println!("s1 = {}, s2 = {}", s1, s2);
}
```




```
C:/Users/Plato/.cargo/bin/cargo.exe run --color=always --package untitled2 --bin untitled2
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\untitled2.exe`
s1 = hello, s2 = hello
```

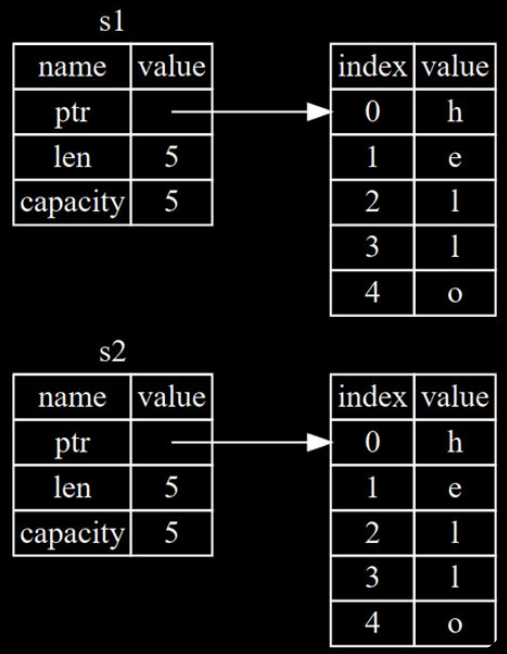
# Data copy format




01

## Shallow Copy

浅拷贝是创建一个新的对象，然后将原始对象中的元素复制到新对象中。但是，如果元素是对象引用（例如列表中的列表），则只复制引用，而不是实际的对象。***Rust中不是标准的浅拷贝。***




02

## Deep Copy

深拷贝创建一个新对象，然后递归地复制原始对象及其所有嵌套的对象，而不仅仅是第一层元素。

# Ownership move in function

```rust
fn main() {
    let s :String = String::from( s: "hello");
    takes_ownership( some_string: s);
    //println!("{}", s);
    let x :i32 = 5;
    makes_copy( some_integer: x);
    println!("{}", x);
}
1 个用法
fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}
1 个用法
fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

## 作用域与Drop方法

```
C:/Users/Plato/.cargo/bin/cargo.exe run --color=always --package untitled2 --bin untitled2
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\untitled2.exe`
hello
5
5

进程已结束，退出代码为 0
```

# Ownership move in function (return value)

```rust
fn main() {
    let s1 :String = gives_ownership();         // gives_ownership 将返回值
    // 移给 s1
    let s2 :String = String::from( s: "hello");     // s2 进入作用域
    let s3 :String = takes_and_gives_back(s2);  // s2 被移动到
    // takes_and_gives_back 中,
    // 它也将返回值移给 s3
} // 这里, s3 移出作用域并被丢弃。s2 也移出作用域, 但已被移走,
// 所以什么也不会发生。s1 移出作用域并被丢弃
fn gives_ownership() -> String {            // gives_ownership 将返回值移动给
    // 调用它的函数
    let some_string :String = String::from( s: "yours"); // some_string 进入作用域
    some_string                                 // 返回 some_string 并移出给调用的函数
}

// takes_and_gives_back 将传入字符串并返回该值
1 个用法
fn takes_and_gives_back(a_string: String) -> String { // a_string 进入作用域
    a_string  // 返回 a_string 并移出给调用的函数
}
```

函数中反复转移所有权是臃肿的

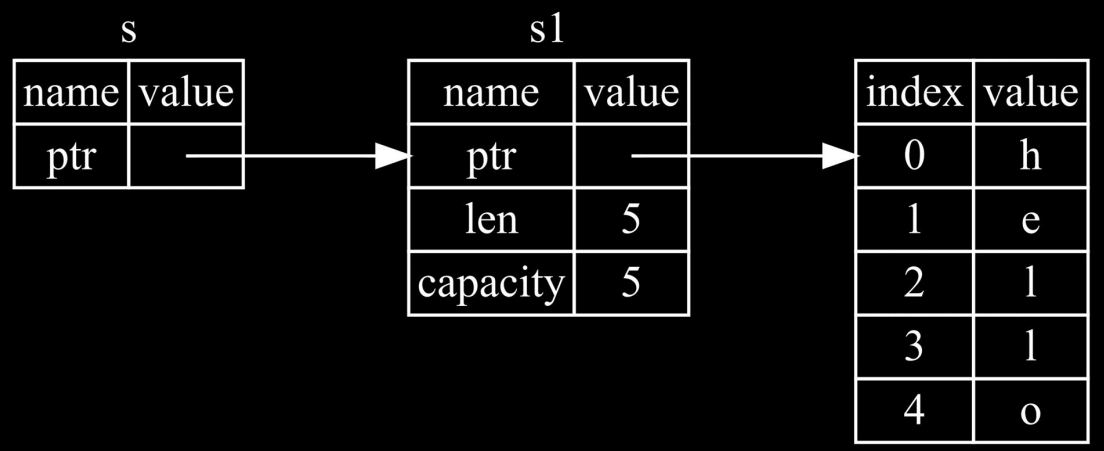
# References and Borrowing

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```




引用允许你使用值但不获取其所有权

# Mutable references

```rust
fn main() {
    let mut s :String = String::from( s: "hello");

    change(&mut s);
    println!("{}", s);
}

1 个用法
fn change(some_string: &mut String) {
    some_string.push_str( string: ", world");
}
```

在同一时间内，只能有一个可变引用指向数据，这确保了在修改数据时不会发生竞争条件。这是Rust借用检查器的一部分，它在编译时防止数据竞争和多线程中的一些常见错误。

# Smart pointers(Deref)

```rust
fn main() {
    let s1 :String  = String::from( s: "hello");
    let s2 :&String  = &s1;
    println!("s1 = {}, s2 = {}", s1, *s2);
}
```

Rust通过显式标识符来确保程序的安全性和高效率
在编译层面确保可执行文件在效率和安全性上具有最好的表现
和C语言一样，Rust强编译弱运行，但更加极端

# Advantages and disadvantages

- 01 High Performance
- 02 High security
- 03 Concurrency

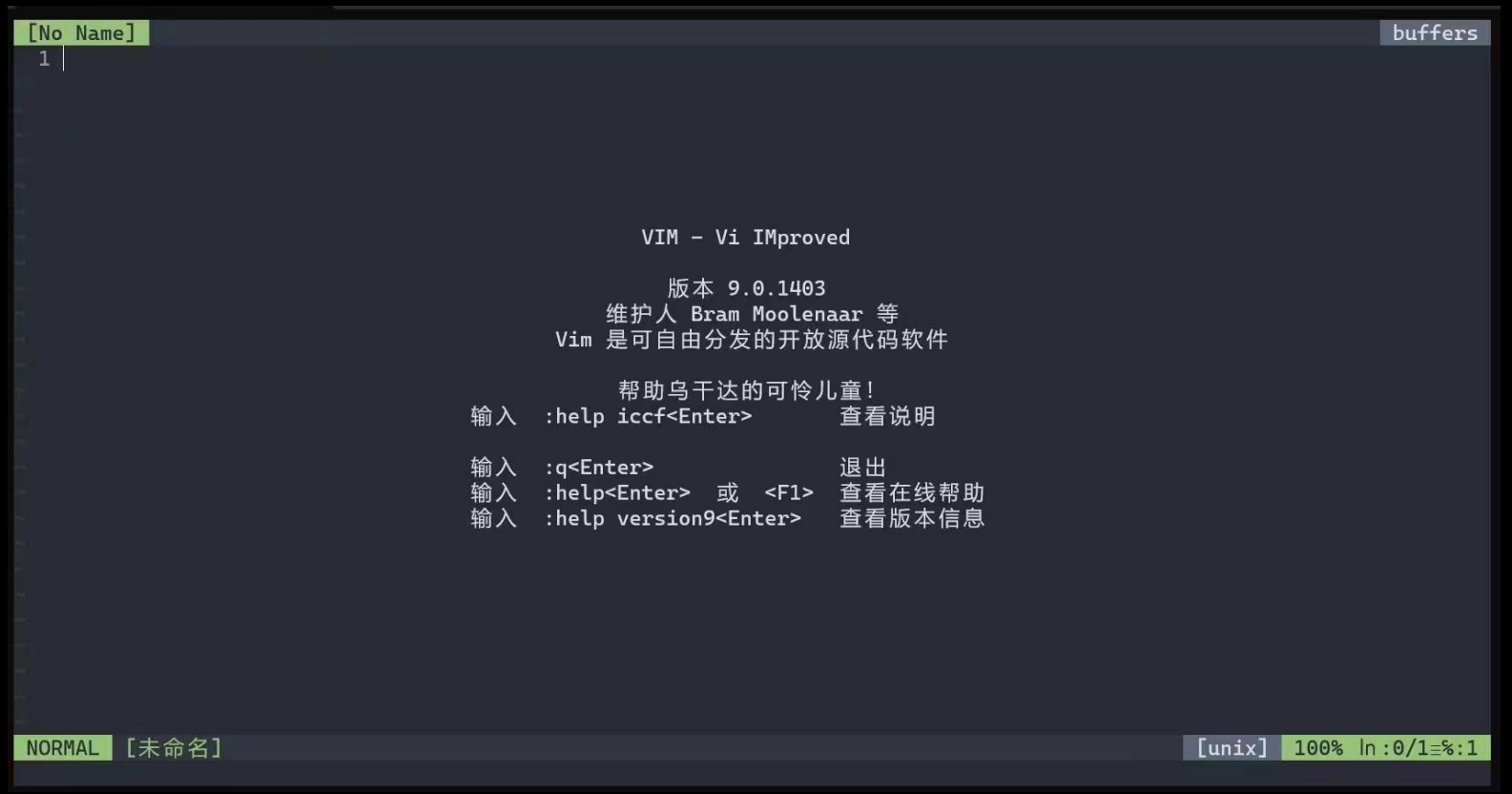
- 01 Learning Curve
- 02 Development Speed
- 03 Compilation Times

Rust是一门开源的编程语言

感谢一代又一代技术人员对开源事业做出的贡献

## Bram Moolenaar

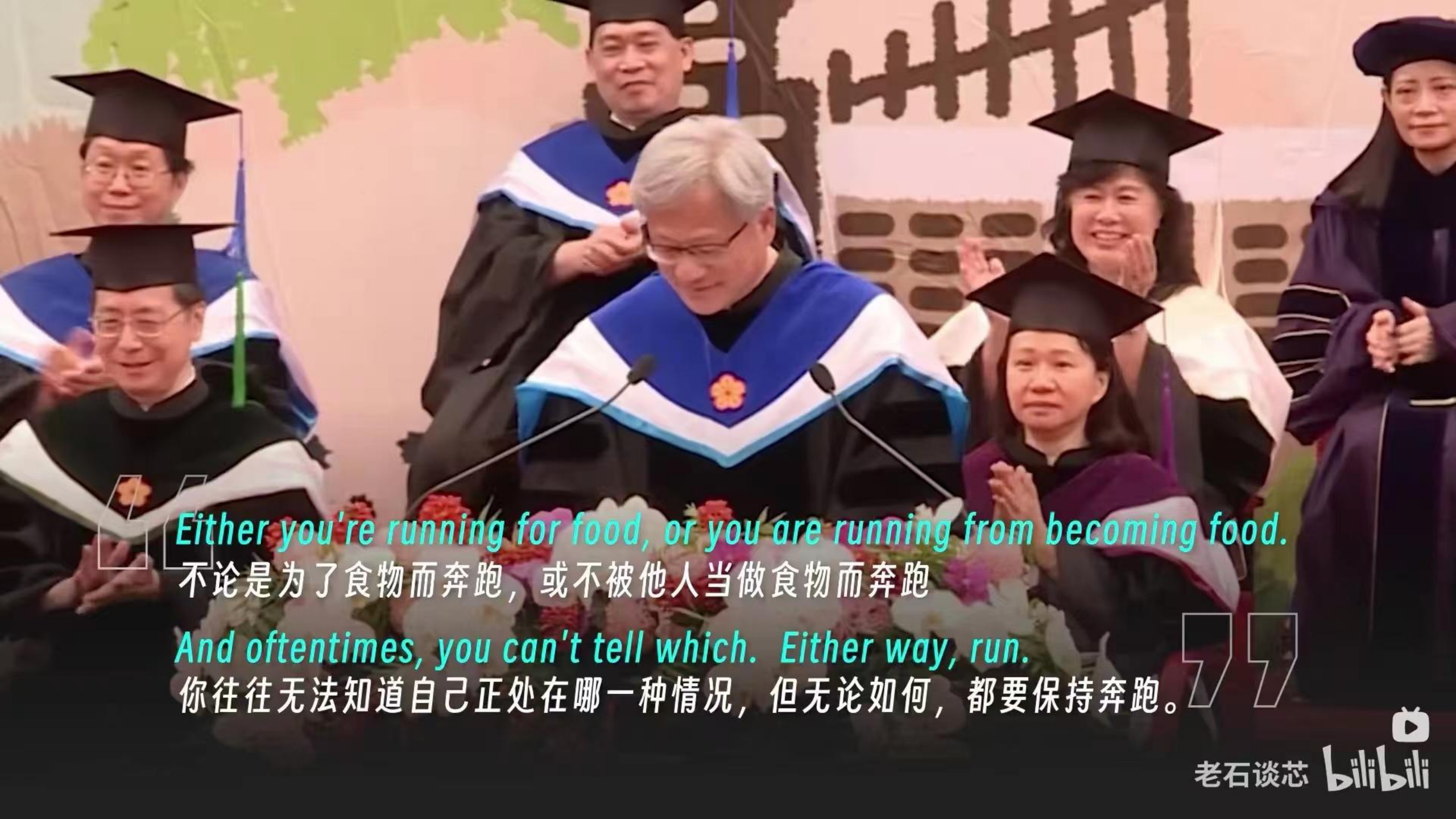

| | |
|---|---|
| **Born** | 1961<br>Lisse |
| **Died** | August 3, 2023 (aged 62) |
| **Known for** | Vim, ICCF Holland |
| **Awards** | NLUUG Awards |
| **Website** | www.moolenaar.net |




著名开源文本编辑器Vim创始人Bram Moolenaar于2023年8月3日去世

世界会记住您对开源事业做出的贡献

# The Last Page


Either you're running for food, or you are running from becoming food.
不论是为了食物而奔跑，或不被他人当做食物而奔跑
And oftentimes, you can't tell which. Either way, run.
你往往无法知道自己正处在哪一种情况，但无论如何，都要保持奔跑。
老石谈芯
bilibili