README.md — Demo: memória (stack, heap) e código com Rust (para Codespaces)

memoria_demo — Stack, Heap e Código (Rust)

Objetivo: demonstrar, de forma prática e visual, como um programa compilado é carregado em memória, onde ficam dados em *stack* e *heap*, e onde está o código (texto/segmento .text).
Formato: exemplo em Rust (simples CLI) que pede nome e data de nascimento e calcula a idade. Foca em: alocação em stack vs heap, ponteiros/endereços e inspeção do binário/assembly.

> Este projeto foi pensado para uso em **GitHub Codespaces** (ou qualquer terminal com Rust toolchain). É educativo — não contem informações sensíveis dos alunos.

Conteúdo

---

1. `Cargo.toml`

```
[package]
name = "memoria_demo"
version = "0.1.0"
edition = "2021"

[dependencies]
chrono = "0.4"
```

2. `src/main.rs`

```
use chrono::prelude::*;
use std::io::{self, Write};

fn main() {
    println!("=== memória_demo (Stack vs Heap) ===\n");
```

```
    // 1) Dados estáticos (literal) -> tipicamente armazenado em .rodata
(segmento de dados do executável)
    let welcome: &str = "Bem-vindo ao demo de memória!";

    // 2) Entrada do usuário: String (heap) e parsing (ex: birth_year)
    println!("{}", welcome);
    let name = read_line("Nome do estudante: ");
    let birth_year: i32 = loop {
        let s = read_line("Ano de nascimento (YYYY): ");
        match s.trim().parse() {
            Ok(y) => break y,
            Err(_) => println!("Ano inválido. Tente novamente."),
        }
    };

    // 3) Exemplos de alocações: stack vs heap
    // stack_value: valor escalar (armazenado na stack como parte do frame
atual)
    let stack_value: i32 = 12345;

    // heap_box: Box aloca no heap (o Box em si (pointer) fica na stack; o
valor apontado fica na heap)
    let heap_box = Box::new(2025i32);

    // name_chars: Vec<char> (estrutura no stack, buffer no heap)
    let name_chars: Vec<char> = name.chars().collect();

    // 4) calcular idade (usa chrono para pegar o ano atual)
    let now = Local::now();
    let current_year = now.year();
    let age = current_year - birth_year;

    println!("\n--- Resultado ---");
    println!("Nome (String)   : {}", name.trim());
    println!("Ano nascimento  : {}", birth_year);
    println!("Ano atual       : {}", current_year);
    println!("Idade aproximada: {} anos\n", age);

    // 5) Mostrar endereços e demonstrar onde cada coisa vive (observacional)
    println!("--- Endereços / Pistas de memória ---");
    println!("&welcome (literal .rodata)      = {:p}", welcome as *const str);
    println!("name (String object on stack)   = {:p}", &name);
    println!("name buffer (heap) as_ptr()     = {:p}", name.as_ptr());
    println!("stack_value (stack)             = {:p}", &stack_value);
    println!("heap_box pointer (on stack)     = {:p}", &heap_box);
    println!("heap_box pointee (heap)         = {:p}", &*heap_box);
    println!("name_chars Vec struct (stack)   = {:p}", &name_chars);
    println!("name_chars buffer (heap)        = {:p}", name_chars.as_ptr());

    // 6) endereço de função (código -> typically in .text)
    println!("example_function (endereço código) = {:p}", example_function as
*const ());
```

```
    // 7) usar uma função separada para mostrar outro frame de stack (para
comparar)
    show_stack_frame(&name, stack_value);

    println!("\n(Dica) Para inspecionar o binário/assembly: veja seção 'Ver
binário / assembly' no README.");
}

/// função auxiliar que existe no segmento de código (.text)
fn example_function() {
    // corpo vazio — usamos apenas o endereço
}

fn read_line(prompt: &str) -> String {
    print!("{}", prompt);
    io::stdout().flush().expect("flush failed");
    let mut s = String::new();
    io::stdin().read_line(&mut s).expect("failed to read");
    s
}

fn show_stack_frame(name: &String, local: i32) {
    // esse frame terá seus próprios locais na stack; imprimimos endereços para
comparar
    println!("\n--- Dentro de outra função (novo frame na stack) ---");
    println!("param name (referência) addr = {:p}", name);
    println!("local (i32) addr              = {:p}", &local);
}
```

---

3. Instruções (GitHub Codespaces)

1. Abra o Codespace / workspace do repositório.
2. No terminal do Codespace, instale Rust se necessário:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
```

3. Crie o projeto (ou copie os arquivos acima):

```
cargo new memoria_demo
cd memoria_demo
# substituir Cargo.toml e src/main.rs pelo conteúdo deste projeto
```

4. Compile e execute:

```
cargo run
```

ou em modo release:

```
cargo build --release
./target/release/memoria_demo
```

---

3. Como inspecionar em tempo de execução (endereços)

O programa imprime endereços observáveis. Anotem:

- endereço do literal (`welcome`) — dá pista de `.rodata` ou seção semelhante do executável;
- endereço do `String` (struct) em pilha (stack);
- endereço do buffer (`as_ptr()`) — tipicamente no *heap*;
- endereço do `Box` e do valor apontado — mostra pointer on stack vs value on heap;
- endereço do `example_function` — mostra que código vive em outro segmento (tipicamente `.text`).

---

4. Ver binário / assembly

Existem várias formas. Duas formas simples:

(A) Gerar assembly com `rustc`

No Codespace você pode compilar o `src/main.rs` diretamente:

```
# compilar gerando assembly
rustc --edition=2021 -C opt-level=3 --emit=asm -o memoria.s src/main.rs
# abrir o arquivo memoria.s
less memoria.s
```

> Observação: em projetos `cargo` pode ser necessário apontar para o arquivo `src/main.rs`. Outra alternativa:

(B) Gerar o binário e usar `objdump` para disassembly

```
cargo build --release
# localizar binário
objdump -d target/release/memoria_demo | less
```

(C) Ferramentas extras (opcional, para demonstração ao vivo)

- `cargo-asm` (instalar via `cargo install cargo-asm`) mostra assembly por função.
- `gdb` / `lldb` para debug e inspeção de segmentos.

> Atenção: comandos podem variar conforme o ambiente. No Codespaces os utilitários `objdump`, `gdb` costumam estar disponíveis, mas confirme.

---

6. Discussão

- **Loader / Processo:** quando você executa o binário, o *loader* do sistema operacional mapeia segmentos do executável para a memória: `.text` (código), `.rodata` (literais constantes), `.data` (variáveis globais), segments para heap e stack (heap é gerenciado pelo alocador/allocator -> `malloc`/`jemalloc`/`mimalloc`/`alloc` do Rust; stack é criado por thread).
- **Stack:** frames crescem/encolhem no `call/return`; locais e parâmetros aparecem como endereços próximos. Funções separadas tem frames separadas (comparamos endereços em `main` vs `show_stack_frame`).
- **Heap:** alocações dinâmicas (`String`, `Vec`, `Box`) têm seus buffers no heap; o objeto wrapper (ex.: `String` struct) vive no stack (ou em outro heap/frame) e contém ponteiros para o buffer.
- **Código:** endereço de função demonstra que o trecho de código está em uma região separada (segmento de código).
- **ASLR & variabilidade:** muitas plataformas usam ASLR (Address Space Layout Randomization) — endereços mudam entre execuções. Discuta isso com a classe: torna ataques mais difíceis; afeta demonstrações (pode ser um ponto de discussão).
- **Otimizador / inlining:** em release / otimizado, o compilador pode otimizar/eliminar variáveis ou inline funções; para demonstrar stack vs heap com mais estabilidade, prefira `cargo build` (debug) para observar estruturas não-otimizadas, ou compile com `-C opt-level=0`.

---

7. Crítica e sugestões de melhoria (reflexões sobre o exercício)

Pontos fortes

- Simples, direto e portável.
- Permite discutir loader/processo/segmentos sem mexer em código perigoso.
- O uso de endereços impressos é excelente para visualização imediata.
- Funciona bem em Codespaces (container), sem necessidade de permissões especiais.

Riscos / limitações

- Em builds otimizados (`--release`) o compilador pode eliminar variáveis não usadas ou mover/inline funções — isso pode confundir a análise dos dados. **Solução:** experimente compilar em *debug* (`cargo build`) e depois em *release* para comparar o efeito das otimizações.
- Endereços variam por execução por causa do ASLR.
- Usar `chrono` adiciona dependência (download no `cargo build`).
- Codespaces é um ambiente containerizado — o processo ainda tem a mesma organização de memória, mas algumas ferramentas de baixo nível podem não estar instaladas por padrão (ex.: `objdump`). Tenha alternativas prontas (mostrar assembly gerado com `rustc --emit=asm`).

Melhorias possíveis

- Adicionar modo *instrumentado* que salva um simple memory map (usando `/proc/self/maps` em Linux) para mostrar como o kernel mapeou segmentos (ótimo pra Linux/Codespaces).
- Mostrar a diferença entre debug vs release automaticamente (rodar os dois e imprimir observações).
- Implementar uma versão que grava somente estatísticas agregadas (evita gravar PII de alunos).
- Incluir um slide com diagrama da memória do processo (stack, heap, bss, data, text, mmap).

---

8. Exercícios em aula

1. **Comparar endereços debug vs release:** compile com `cargo build` e `cargo build --release`, rode ambos e compare endereços impressos — discuta otimizações e inlining.
2. **Identificar heap vs stack:** peça que identifiquem quais endereços apontam para heap (buffers) e quais para stack (structs locais).
3. **Desativar ASLR (opcional, uso em laboratório controlado):** em Linux: `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` (apenas se permitido). Explique implicações de segurança. *Nota: execute apenas em VMs de laboratório controladas.*
4. **Mapas de memória:** (Linux) ler `/proc/self/maps` no programa para mostrar mapeamentos do processo.
5. **Modificar o programa:** forçar alocações grandes no heap (e.g., `Vec::with_capacity(1_000_000)`) e observar crescimento/uso de heap com `top` ou `htop`.
6. **Implementar contador sem armazenar nomes:** alterar para gravar apenas contagens por primeiro caractere do nome (prática de minimização de dados).

---

9. Observações finais e Ética

- **Evite gravar** dados sensíveis (senhas, números etc.).
- Não utilize este código em produção. Ele é apenas para demonstração educacional.

---

10. Recursos & comandos resumidos

- Build debug: `cargo build`
- Run debug: `cargo run`
- Build release: `cargo build --release`
- Generate asm: `rustc --edition=2021 -C opt-level=3 --emit=asm -o memoria.s src/main.rs`
- Disassemble binary: `objdump -d target/release/memoria_demo | less`
- (Optional) cargo-asm: `cargo install cargo-asm` e depois `cargo asm memoria_demo::example_function`