

Depuração, Gerenciamento de Memória e Análise de Desempenho

Boas Práticas de Programação - 2025.2

Prof. Fernando Figueira

14 de Novembro de 2025

Agenda da Aula

A Arte da Depuração (Debugging)

Técnicas e ferramentas para encontrar bugs.

Gerenciamento de Memória

Entendendo memory leaks e uso eficiente de recursos.

Análise de Desempenho e Gargalos

Profiling e otimização: de $O(n^2)$ para $O(1)$.

Exercícios Práticos (Passo a Passo)

ASan (C/C++), cProfile (Python) e Chrome DevTools (JS).

Integração com o Projeto (Requisitos U3)

Como aplicar a aula na entrega final.

Parte 1: A Arte da Depuração

“ Se depurar é o processo de remover bugs, então programar deve ser o processo de colocá-los. ”

— Edsger Dijkstra

Técnicas Essenciais de Depuração



Debuggers (Step-by-Step)

Use breakpoints para pausar a execução. Inspecione variáveis (watch) e execute linha por linha (step over, step in). É a forma mais controlada de entender o fluxo.

Ferramentas: VS Code Debugger, GDB, PDB (Python), Debuggers de IDEs (IntelliJ, Eclipse).



Logging Estratégico

Adicione logs em pontos-chave (início de função, condicionais, loops) para rastrear o fluxo de execução e o estado das variáveis sem interromper o programa.

Níveis: Use níveis de log (DEBUG, INFO, ERROR) para filtrar ruído em produção.



Git Bisect

Quando um bug "apareceu do nada", use o controle de versão. O `git bisect` automatiza uma busca binária no histórico de commits para encontrar exatamente qual commit introduziu o bug.

Comando: `git bisect start`

...

Zoom-in: O que é Git Bisect?



1. Iniciar a "Caçada"

Você informa ao Git um commit "ruim" (`bad`) onde o bug existe (ex: `HEAD`) e um commit "bom" (`good`) onde ele não existia (ex: `v1.0`).

Comando: `git bisect start`



2. Testar (Busca Binária)

O Git faz checkout do commit no *meio* do intervalo. Você testa.

Se o bug sumiu, digite `git bisect good`. Se o bug apareceu, `git bisect bad`. O Git corta o intervalo pela metade.



3. Encontrar o Culpado

O processo repete (para 1000 commits, ~10 testes). O Git isola e aponta o *primeiro commit* que introduziu o bug.

Comando final: `git bisect reset`

Gerenciamento de Memória

Onde estão seus bytes?

Problemas Comuns de Memória

C Linguagens Manuais (C/C++)

- **Memory Leaks (Vazamentos):** Alocar memória (`malloc`, `new`) e nunca liberar (`free`, `delete`). O programa "incha" ao longo do tempo.
- **Dangling Pointers (Ponteiros Soltos):** Tentar usar um ponteiro para uma área de memória que já foi liberada. Causa comportamento indefinido.
- **Buffer Overflows:** Escrever dados além dos limites de um array ou buffer, corrompendo memória adjacente.

≠ Linguagens com GC (Python/Java/JS)

- **Referências "Esquecidas":** O Garbage Collector (GC) só libera o que não tem referência. Manter objetos em listas globais ou caches que nunca são limpos é o "novo" memory leak.
- **Estruturas Ineficientes:** Carregar um arquivo de 5GB em uma lista na memória, em vez de processá-lo linha por linha (com iteradores/generators).

Análise de Desempenho

Por que meu código está lento?

Profiling e Detecção de Gargalos



Profiling (Medição)

Não adivinhe, meça! Profilers são ferramentas que executam seu código e mostram exatamente quanto tempo (CPU) ou memória cada função gasta.

O resultado, muitas vezes, é surpreendente e revela o verdadeiro gargalo.



Complexidade (Big O)

O gargalo é algorítmico? Um loop dentro de outro ($O(n^2)$) para processar 10.000 itens é insustentável.

Trocar a estrutura de dados (de lista para hash map) pode mudar uma busca de $O(n)$ para $O(1)$ e ser a otimização mais impactante.



Gargalos de I/O

Seu código pode estar rápido, mas estar **esperando**. O gargalo é o código ou a espera?

Verifique chamadas de rede lentas, leituras/escritas de disco, ou queries ineficientes no banco de dados (ex: "N+1 queries").

Zoom-in: O Gargalo "N+1 Queries" (I/O)

🔴 O Problema (N+1)

Um anti-padrão comum que causa lentidão extrema de I/O.

1. **Query 1:** Buscar 100 posts de um blog. (1 query)
2. **Loop (N=100):** Para cada post, fazer uma nova query para buscar o nome do autor. (100 queries)

Total: 101 queries para uma única página. A maior parte do tempo é gasta esperando a rede/DB.

✓ A Solução (Eager Loading)

Consiste em carregar os dados relacionados de forma antecipada.

1. **Query 1:** Buscar 100 posts de um blog. (1 query)
2. **Query 2:** Buscar os Autores *apenas* dos 100 posts encontrados (usando `JOIN` ou `WHERE author_id IN [...]`). (1 query)

Total: 2 queries. A otimização (comum em ORMs) reduz drasticamente o gargalo de I/O.

Exercícios Práticos

Passo a passo com as Ferramentas

Visão Geral das Ferramentas por OS

| Tarefa | Linux (WSL) | macOS | Windows (Nativo) |
|---|--|--|--|
| Depuração de Memória (C/C++) Buffer overflow, leaks, etc. | <code>gcc -fsanitize=address</code> (ASan) | <code>clang -fsanitize=address</code> (ASan) | MSVC (VS 2019+): <code>/fsanitize=address</code> Alternativa: Dr. Memory |
| Profiling de CPU (Python) Funções lentas (gargalos) | <code>cProfile + snakeviz</code> (Cross-platform) | <code>cProfile + snakeviz</code> (Cross-platform) | <code>cProfile + snakeviz</code> (Cross-platform) |
| Profiling de UI/JS Renderização, Layout Thrashing | Chrome / Firefox DevTools (Cross-platform) | Chrome / Firefox DevTools (Cross-platform) | Chrome / Firefox DevTools (Cross-platform) |

Exercício 1: Depuração de Memória (C/C++)

Objetivo

Encontrar um bug de "Buffer Overflow" (escrita fora da memória alocada) usando o **AddressSanitizer (ASan)**.



Ferramenta

O compilador C (`gcc` , `clang` ou `cl.exe`) com a flag `-fsanitize=address` .

Exercício 1 (Passos 1-2): Código e Compilação

Passo 1: O Código com Bug

Salve este código como `bug.c`. O bug é sutil: o array `arr` tem 5 posições (0-4), mas o loop `for` tenta acessar a posição 5.

```
#include <stdio.h>

int main() {
    int arr[5];
    int i;

    // Loop vai de 0 a 5 (6 iterações)
    for (i = 0; i <= 5; i++) {
        arr[i] = i * 10; // BUG na última iteração!
    }

    printf("Valor: %d\n", arr[0]);
    return 0;
}
```

Passo 2: Compilar com ASan (por OS)

Compile com a flag de "sanitize" e debug (`-g`). O comando varia um pouco:

```
# Linux (WSL) com GCC:
gcc -g -fsanitize=address bug.c -o bug_asan

# macOS com Clang:
clang -g -fsanitize=address bug.c -o bug_asan

# Windows (Terminal do Desenvolvedor do VS):
cl.exe /DEBUG /fsanitize=address bug.c
```

Isso cria um executável (`bug_asan` ou `bug.exe`) que é "instrumentado" e vai monitorar a si mesmo.

Exercício 1 (Passos 3-4): Rodar e Analisar

Passo 3: Rodar o Programa

Agora, simplesmente execute o programa:

```
# Em Linux/macOS  
./bug_asan  
  
# Em Windows  
.\\bug.exe
```

O ASan vai interceptar o erro e imprimir um relatório detalhado.

Analizando a Saída do ASan

O relatório será grande, mas claro (em vermelho):

```
==12345==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x...  
WRITE of size 4 at 0x... thread T0  
    #0 0x10... in main bug.c:8:14  
    #1 0x7... in start (libdyld.dylib:...)  
  
Address 0x... is located in stack of thread T0 at offset 52  
This frame has 1 object(s):  
[48, 68) 'arr' (int[5])
```

Tradução: "Erro: Stack Buffer Overflow... na função `main`, arquivo `bug.c`, **linha 8, coluna 14**... ao lado do array '`arr`'."

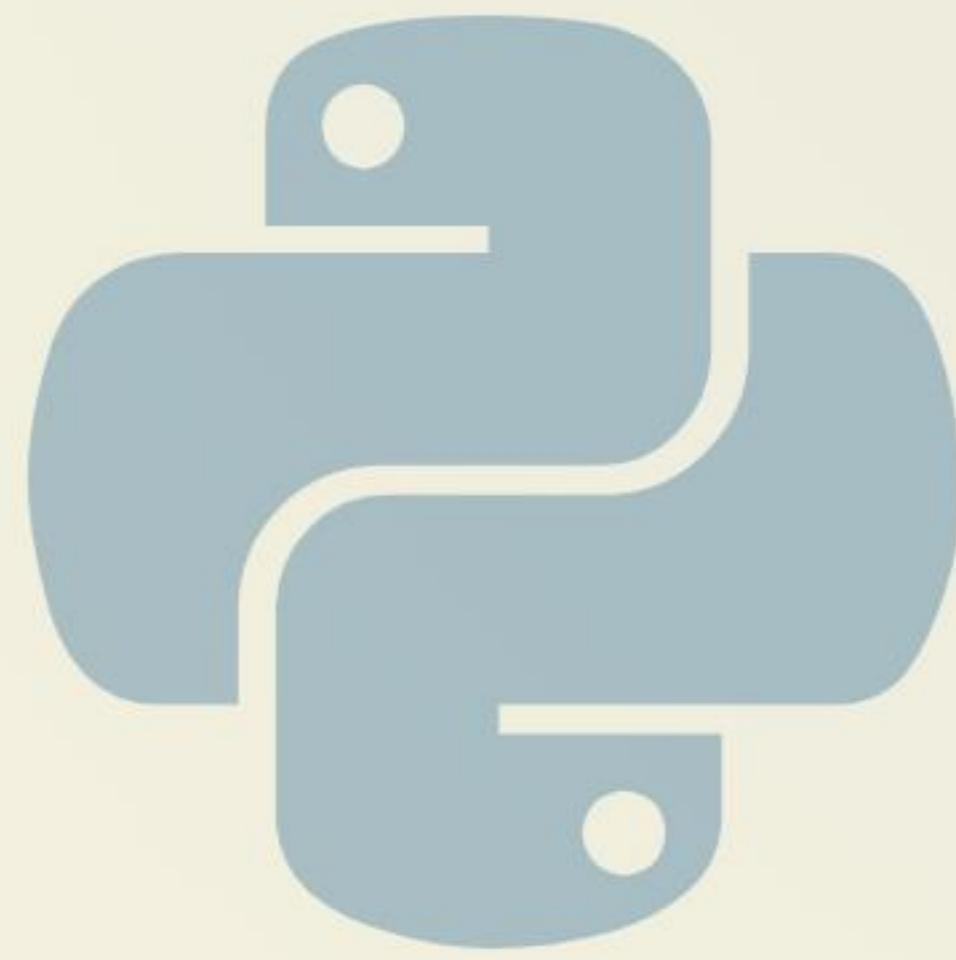
Passo 4: A Correção

O relatório aponta a **linha 8**. Corrija o loop para `i < 5`. Recompile e rode novamente. O erro desaparecerá.

Exercício 2: Profiling de CPU (Python)

Objetivo

Encontrar o gargalo algorítmico ($O(n^2)$) em um script Python. Esta tarefa é **cross-platform**.



Ferramentas

`cProfile` (nativo do Python), `pip` (para instalar) e `snakeviz` (visualizador).

Setup (em qualquer OS):

```
pip install snakeviz
```

Exercício 2 (Passos 1-2): Código Lento e cProfile

Passo 1: O Código Lento

Salve como `profile_me.py`. A função `find_matches` é $O(n^2)$ (loop dentro de loop). `process_data` gasta 99% do tempo nela.

```
def find_matches(list1, list2):
    matches = []
    for item1 in list1:
        for item2 in list2: # O(n*m)
            if item1 == item2:
                matches.append(item1)
    return matches

def process_data():
    big_list_1 = list(range(10000))
    big_list_2 = list(range(5000, 15000))
    print("Processando...")
    matches = find_matches(big_list_1, big_list_2)
    print(f"Encontrados: {len(matches)}")

if __name__ == "__main__":
    process_data()
```

Passo 2: Rodar o cProfile

No terminal, rodamos o módulo `cProfile`. A flag `-o` salva as estatísticas em um arquivo `profile.stats`.

```
# Comando idêntico para Windows, macOS e Linux
python -m cProfile -o profile.stats profile_me.py
```

O script vai rodar (e demorar alguns segundos). Ao final, você terá o arquivo `profile.stats`.

Exercício 2 (Passos 3-4): snakeviz e Otimização

Passo 3: Visualizar com snakeviz

Agora, usamos o `snakeviz` para ler o arquivo de estatísticas. Ele abrirá o resultado no seu navegador.

```
# Inicia um servidor web e abre o relatório  
snakeviz profile.stats
```

No navegador, você verá um "Flame Graph" ou "Sunburst" mostrando que 99% do tempo (`tottime`) é gasto na função `find_matches`.

Passo 4: A Otimização ($O(n)$)

O gargalo é o $O(n^2)$. A solução é usar um `set` (tabela hash) para busca $O(1)$.

```
def find_matches_fast(list1, list2):  
    # Converte a lista maior para um set (busca O(1))  
    set2 = set(list2)  
    matches = []  
  
    # Apenas 1 loop (O(n))  
    for item1 in list1:  
        if item1 in set2: # Busca O(1)  
            matches.append(item1)  
    return matches
```

Ao trocar a função e rodar o profiler de novo, o tempo total cairá de segundos para milissegundos.

Exercício 3: Profiling de UI (JavaScript)

Objetivo

Identificar um gargalo de renderização (UI) no navegador. Esta tarefa é **cross-platform**.

Ferramenta

Chrome DevTools (Aperte F12 ou Cmd+Opt+I) -> Aba **Performance**.



Exercício 3 (Passos 1-2): O Cenário Lento

Passo 1: O Código Lento (JS)

Em um `index.html`, adicione um `Rodar` e este script. O bug: ele mexe no DOM 10.000 vezes *dentro* do loop.

```
function addItem() {
  for (let i = 0; i < 10000; i++) {
    const el = document.createElement('div');
    el.innerText = `Item ${i}`;

    // GARGALO: Cada .appendChild() força o
    // navegador a recalcular o layout.
    document.body.appendChild(el);
  }
}

document.getElementById('btn').onclick = addItem;
```

Passo 2: Gravar a Performance

Abra o `index.html` no Chrome.

1. Abra o DevTools (F12).
2. Vá para a aba "**Performance**".
3. Clique no botão "**Record**" (gravar) .
4. Clique no botão "Rodar" na sua página.
5. Espere a página "descongelar".
6. Clique em "**Stop**" no DevTools.

Exercício 3 (Passos 3-4): Analisar e Corrigir

Passo 3: Analisar o "Flame Graph"

Você verá um gráfico com muito tempo gasto em "Scripting" e "Rendering".

O gráfico da função `addItem` terá muitos "triângulos" roxos  (Layout) e verdes  (Paint) repetidos.

Isso é o **Layout Thrashing**: o navegador é forçado a recalcular o layout milhares de vezes.

Passo 4: A Otimização (Fragment)

A solução é fazer ***todas*** as mudanças fora do DOM (em um "fragment") e anexar tudo de ***uma só vez***.

```
function addItemFast() {  
    // 1. Cria um "fragment" em memória  
    const fragment = document.createDocumentFragment();  
  
    for (let i = 0; i < 10000; i++) {  
        const el = document.createElement('div');  
        el.innerText = `Item ${i}`;  
  
        // 2. Adiciona ao fragmento (operação rápida)  
        fragment.appendChild(el);  
    }  
  
    // 3. Adiciona o fragmento ao DOM (1 ÚNICA OPERAÇÃO)  
    document.body.appendChild(fragment);  
}
```

Se você gravar a performance de novo, o tempo de "Layout" será mínimo.

Aplicação no Projeto (Requisitos U3)

- Depuração (15%):** Crie o `debugging-log.md`. Documente 3+ bugs, a **técnica** de depuração usada (debugger, logging, git bisect) e o **código antes/depois**.
- Análise de Performance (20%):** Identifique 2+ gargalos. Use um **profiler** (cProfile, DevTools, JProfiler) para **medir**. Documente métricas "antes vs. depois" e a análise de **complexidade (Big O)**.
- Gerenciamento de Memória (15%):**
Use a ferramenta apropriada para seu OS (veja slide 12). **(C/C++):** Use **ASan** (`-fsanitize=address` ou `/fsanitize=address`), **Dr. Memory**, ou o **Visual Studio Profiler** e documente a correção de leaks/erros.

Dúvidas?

Próximos Passos:

1. Rodar os exercícios práticos hoje (usando a ferramenta do seu OS).
2. Aplicar o profiler (cProfile, DevTools, etc.) no projeto U3.
3. Começar a escrever o `debugging-log.md`.
4. Entrega U3 (05/12).

Repositório: github.com/fmarquesfilho/bpp-2025-2