Chapter 1

Gerenciamento de Memória e Ambientes de Runtime

1.1 Objetivos

⋆ title=Objetivos da Aula

Ao final deste capítulo, o estudante deverá ser capaz de:

- Compreender como o **compilador e o sistema operacional** colaboram para gerenciar a memória de um programa em execução.
- Detalhar o layout de memória de um processo: **Código, Dados Estáticos, Heap e Stack**.
- Entender o papel da pilha (**Stack**) na implementação de sub-rotinas, suporte à recursão e alocação de variáveis locais através dos **Registros de Ativação**.
- **Simular passo a passo** o empilhamento e desempilhamento de frames durante uma chamada recursiva, identificando os valores de SP, FP e dos campos de cada frame.
- Analisar como linguagens com funções aninhadas resolvem o acesso a variáveis não locais por meio de **Links Estáticos**.
- Compreender os desafios do **Heap** (como a fragmentação) e o funcionamento dos algoritmos clássicos de **Coleta de Lixo** (Garbage Collection).

1.2 Introdução ao Ambiente de Runtime

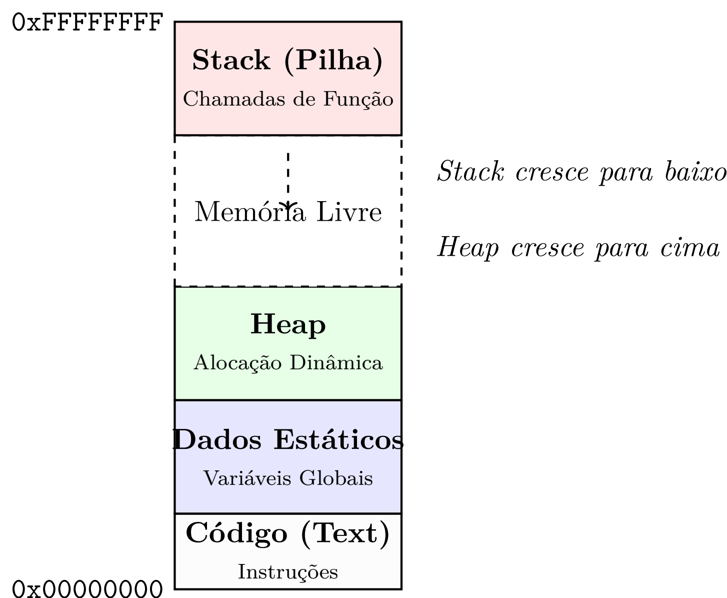
Durante as fases de compilação, o compilador traduz nomes lógicos (variáveis, funções) em endereços de memória, mapeando abstrações de alto nível para estruturas concretas da máquina alvo. No entanto, o compilador **não executa** o programa. Quem administra e gerencia o programa durante sua execução é o **Ambiente de Runtime** (Ambiente de Tempo de Execução), atuando em estreita colaboração com o Sistema Operacional.

O Papel do Compilador e do Runtime

O compilador deve gerar código de máquina fazendo suposições precisas sobre como o sistema operacional organizará a memória do processo. Ele deve inserir instruções extras (conhecidas como *prologue* e *epilogue* de funções) para alocar e liberar memória na pilha, e delegar requisições de memória dinâmica para bibliotecas do runtime (como `malloc` da libc ou o *Garbage Collector* de linguagens gerenciadas).

1.3 O Layout Típico de um Processo

Embora diferentes sistemas operacionais tenham convenções específicas, o layout da memória virtual de um processo em execução segue um padrão universal, dividido em quatro segmentos lógicos:



- **Código (Text):** Contém as instruções em linguagem de máquina do programa compilado. É uma área de tamanho fixo e geralmente marcada pelo SO como apenas leitura (*read-only*) para evitar que o programa sobrescreva suas próprias instruções acidentalmente.
- **Dados Estáticos:** Armazena constantes literais e variáveis globais cujo tamanho é conhecido estaticamente durante a compilação.

- **Heap:** Memória alocada **dinamicamente** sob demanda (por exemplo, via `new` em C++ ou `malloc` em C). Seu tamanho é flexível e ele costuma crescer rumo aos endereços mais altos.

- **Stack (Pilha):** Usada de forma automática pelo compilador para manter informações sobre chamadas de função, variáveis locais e parâmetros. Na maioria das arquiteturas modernas (como x86), a pilha cresce de endereços altos para endereços mais baixos, aproximando-se do Heap.

1.4 Estratégias de Alocação de Memória

Os compiladores tradicionalmente mapeiam variáveis usando três métodos de alocação:

- **Alocação Estática:** As variáveis globais e estáticas são mapeadas para endereços fixos absolutos no segmento de Dados. *Problema:* Não suporta alocação sob demanda e proíbe a recursão (pois não há espaço para múltiplas cópias independentes da mesma variável local de uma função, como acontecia no FORTRAN antigo).

- **Alocação Dinâmica na Pilha (Stack):** As variáveis locais são empilhadas quando uma função é chamada e desempilhadas quando ela retorna. Como a estrutura é *Last-In, First-Out* (LIFO), casa perfeitamente com a natureza aninhada das chamadas e retornos de sub-rotinas, além de habilitar a **recursão**.

- **Alocação Dinâmica no Heap:** Usada para objetos cujo tempo de vida deve se estender além do escopo da função que o criou, ou para estruturas de tamanho variável (como listas encadeadas ou strings dinâmicas).

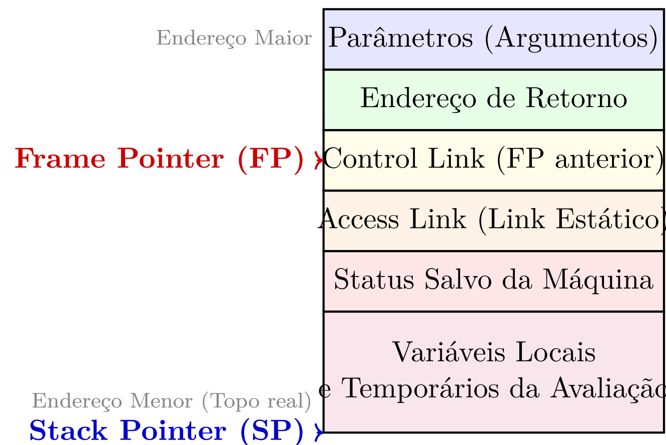
1.5 A Pilha (Stack) e os Registros de Ativação

Quando uma função A chama uma função B, a execução de A é paralisada temporariamente. O compilador deve garantir que, quando B terminar, A consiga retomar a execução exatamente de onde parou, mantendo todos os seus valores de variáveis locais intactos.

Para solucionar isso, a cada chamada de função o sistema empilha um bloco de memória chamado **Registro de Ativação** (*Activation Record* ou *Stack Frame*).

1.5.1 Estrutura do Registro de Ativação

Embora a composição exata de um frame dependa fortemente da **Convenção de Chamadas** (ABI - *Application Binary Interface*) adotada pelo SO/Processador, um frame típico possui os seguintes campos genéricos:



- **Parâmetros:** Valores passados pelo chamador (*caller*) para o chamado (*callee*).

- **Endereço de Retorno:** Para onde o *Program Counter* (PC) deve pular quando a função terminar.

- **Control Link (Link Dinâmico):** Armazena o *Frame Pointer* da função chamadora, permitindo restaurá-lo no fim.

- **Access Link (Link Estático):** Ponteiro para funções em escopos léxicos aninhados (detalhado na próxima seção).

- **Status Salvo:** Cópias dos registradores de uso geral (como `R1`, `R2`) que a função vai sobrescrever, para que sejam devolvidos intactos.

- **Variáveis Locais:** O espaço alocado pelo compilador para matrizes e escalares declarados dentro da função.

1.5.2 Stack Pointer (SP) vs Frame Pointer (FP)

O hardware mantém dois registradores cruciais para o manuseio da pilha:

- O **Stack Pointer (SP):** Aponta para o topo físico da pilha (menor endereço na memória). O SP sofre alterações constantes durante a execução da função, já que operações como empilhar argumentos para chamar outras funções causam seu incremento ou decremento.

- O **Frame Pointer (FP) / Base Pointer (BP):** Fica estático durante **toda** a execução da função. Ele é alinhado normalmente no momento que o Control Link é salvo.

◇ Informação

Por que o FP é mágico? Como o FP nunca muda, o compilador traduz as variáveis locais usando *offsets estáticos e fixos* relativos ao FP. Por exemplo:

- Um parâmetro de função pode estar em `FP + 8`.
- Uma variável local declarada pode estar em `FP - 4`.

Se usássemos o SP para calcular o offset, a compilação seria dificílima, pois a distância entre a variável e o topo da pilha mudaria o tempo todo conforme pushs e pops ocorressem.

1.5.3 Sequência de Chamada (Calling Sequence)

A sequência de códigos que prepara um Registro de Ativação (Prologue) e o desmonta (Epilogue) é inserida no Assembly gerado pelo compilador.

Exemplo de Prologue e Epilogue (estilo C/x86)

Prologue (Ao entrar na função):

```
push FP          // Salva o FP do caller
FP = SP          // FP torna-se a base atual
SP = SP - N      // Desce o SP para alocar N
                 // bytes para variaveis locais
```

Epilogue (Ao sair da função):

```
SP = FP          // Desaloca locais bruscamente
FP = pop         // Restaura o FP do caller
return           // Desempilha Endereco Retorno
```

Note como as instâncias de variáveis das chamadas recursivas não se misturam, pois `SP = SP - N` é executado separadamente, criando um Frame **independente** para cada invocação da função.

1.6 Simulação da Pilha: Recursão com `fat(3)`

Para consolidar o funcionamento dos Registros de Ativação, vamos simular **passo a passo** o que acontece na memória durante a execução do seguinte código:

Código da Simulação

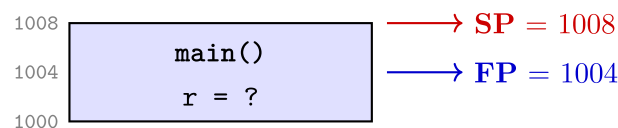
```
int fat(int n) {
  if (n <= 1)
    return 1;
  return n * fat(n-1);
}
int main() {
  int r = fat(3);
  return 0;
}
```

Convenção adotada nesta simulação:

- A pilha **cresce para cima** (endereços maiores indicam topo mais recente).
- Cada célula de memória ocupa 4 bytes.
- **SP** = Stack Pointer (topo atual); **FP** = Frame Pointer (base do frame atual).
- Os endereços são simplificados (base 1000) para facilitar a leitura.

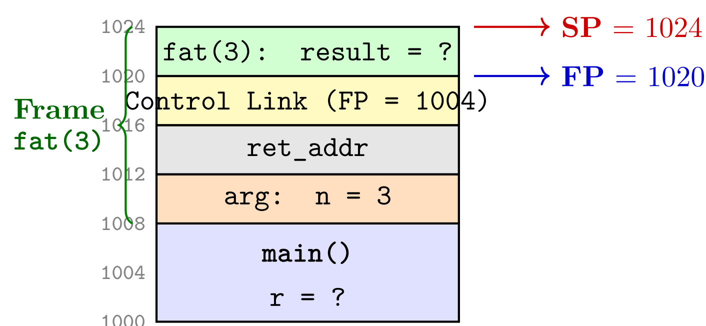
1.6.1 Estado Inicial: apenas `main()` na pilha

Antes de qualquer chamada a `fat`, a pilha contém apenas o frame de `main`:



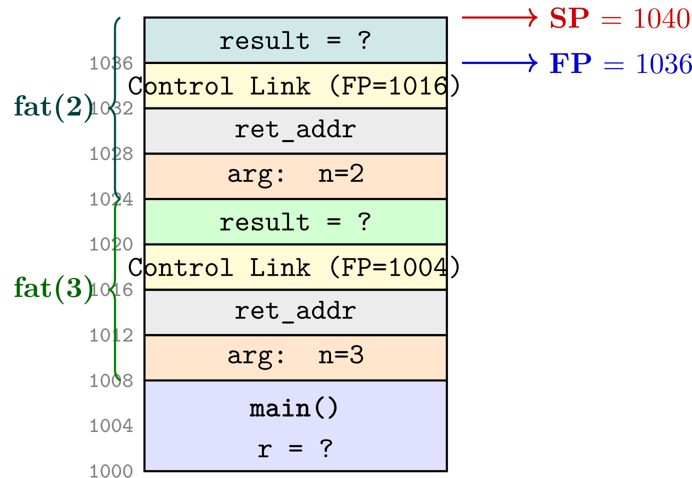
1.6.2 Passo 1: `main` chama `fat(3)`

O *caller* (`main`) empilha o argumento `n = 3`, o endereço de retorno e o Control Link. Em seguida, o *prologue* de `fat` é executado: salva o FP antigo, ajusta `FP = SP` e reserva espaço local.



1.6.3 Passo 2: `fat(3)` chama `fat(2)`

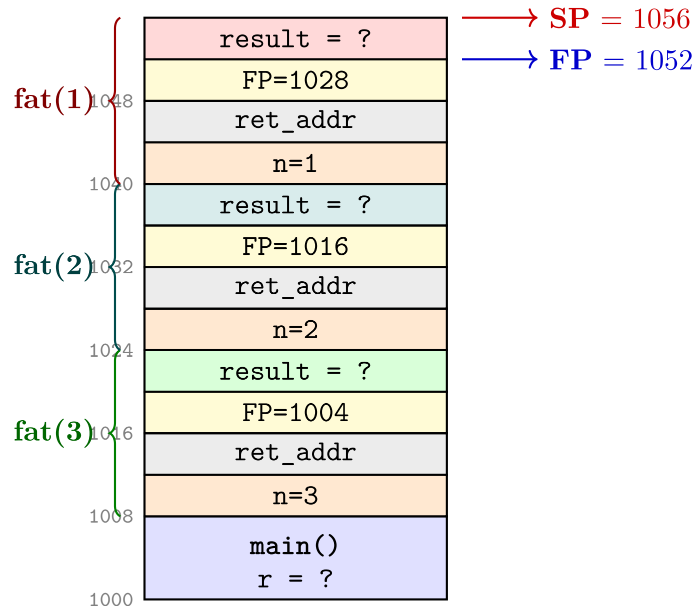
Como `n = 3 > 1`, `fat(3)` chama `fat(2)`, gerando um segundo frame empilhado:



Observe que a pilha já guarda **duas cópias independentes** de `n`: a variável de `fat(3)` (com valor 3) e a de `fat(2)` (com valor 2), em células de memória distintas.

1.6.4 Passo 3: Pico da Pilha — `fat(1)` empilhado

`fat(2)` chama `fat(1)`. Neste momento, há **3 frames simultâneos** ativos na pilha. `fat(1)` detecta `n <= 1` e vai retornar sem fazer nova chamada recursiva.



1.6.5 Passos 4–6: Retornos em Cascata (Desempilhamento)

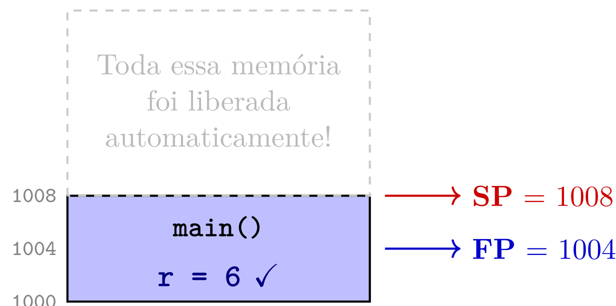
Agora a pilha se desfaz na ordem inversa. Em cada retorno, o epilogue executa `SP = FP` (desaloca o frame) e `FP = pop` (restaura o FP do chamador):

Passo 4 — `fat(1)` retorna 1: Seu frame é desempilhado. `fat(2)` recebe o valor 1 e calcula `result = 2 * 1 = 2`.

Passo 5 — `fat(2)` retorna 2: Seu frame é desempilhado. `fat(3)` recebe 2 e calcula `result = 3 * 2 = 6`.

Passo 6 — `fat(3)` retorna 6: Frame desempilhado. `main` grava `r = 6`.

O estado final da pilha é idêntico ao estado inicial: apenas o frame de `main`, com `r = 6`:



◇ Informação

Custo de memória da recursão: Para `fat(n)`, o pico da pilha contém $n + 1$ frames simultâneos. Para n muito grande, isso pode causar um **Stack Overflow** — o SP ultrapassa a fronteira da região de pilha alocada pelo SO. Linguagens funcionais resolvem isso com *Tail Call Optimization* (TCO), que reutiliza o mesmo frame quando a chamada recursiva é a última operação da função.

1.7 Escopo Léxico e Links Estáticos

Linguagens de programação da família do C não permitem declarar funções dentro de funções. O escopo local é fechado à própria função, e o escopo global acessa variáveis absolutas. Contudo, linguagens como Pascal, ML, Python e JavaScript suportam **Funções Aninhadas** (*Nested Functions*) e capturas de closures lexicais.

O Problema do Escopo em Pascal

```
procedure Outer();
var x: integer;
    procedure Inner();
    var y: integer;
    begin
        y := x + 1; // Acesso a variavel x do escopo de Outer
    end;
begin
    x := 10;
    Inner();
end;
```

Quando `Inner()` for executada em tempo de execução, ela precisará ler `x`. Onde `x` está na pilha? A resposta não é trivial: `Inner` poderia ter sido chamada diretamente por `Outer`, mas também poderia ter sido passada como ponteiro para outra função recursiva profunda, de forma que o Frame Pointer atual de `Inner` estaria **muito distante** do Frame de `Outer` na pilha. E a distância exata é invisível em tempo de compilação (*Dynamic Scoping*).

A Solução: Link Estático (Access Link)

O compilador resolve esse problema inserindo no Registro de Ativação um ponteiro chamado **Link Estático**.

- **Link Dinâmico (FP anterior):** Aponta para o Frame da função que chamou a atual. (Útil para saber para onde retornar o controle de CPU).
- **Link Estático:** Aponta para o Frame da **instância mais recente** da função que *engloba textualmente/lexicamente* a função atual.

Assim, se `Inner` quiser ler `x` de `Outer`, ela segue seu próprio *Link Estático* (que é inserido no frame pela função chamadora), alcança o Frame de `Outer`, e lá aplica o *offset* fixo conhecido de `x`.

1.8 O Heap e a Alocação Dinâmica

O **Heap** (ou Monte) complementa as restrições da Pilha. Objetos que devem transcender a morte do Frame da função que os gerou (como criar um nó de uma árvore e retorná-lo) devem ser alocados no Heap.

1.8.1 O Gerenciador de Memória e a Fragmentação

A administração do Heap é feita por um **Gerenciador de Memória** residente na biblioteca padrão (como as rotinas internas do `malloc` em C). O maior inimigo do gerenciador é a **Fragmentação Externa**.

Com o contínuo processo de criação e deleção de objetos de tamanhos variados, o Heap torna-se esburacado (intercalando blocos em uso e blocos livres). É possível que existam 100 MB de blocos livres somados, mas espalhados em fragmentos de 10 KB. Se um programa solicitar um único array contíguo de 1 MB, a **alocação falhará**, evidenciando o colapso pela fragmentação externa.

1.8.2 Algoritmos Clássicos de Alocação (Free-lists)

O *Runtime* rastreia os "buracos" através de listas encadeadas (*free-lists*). Quando um `malloc(size)` chega, ele deve decidir de onde arrancar o bloco de memória:

- **First-Fit (Primeiro Encaixe):** Escaneia a lista desde o início e toma o primeiro buraco livre que seja maior ou igual ao tamanho solicitado. É rápido, mas tende a acumular pequenos fragmentos no topo da memória com o passar do tempo.

- **Best-Fit (Melhor Encaixe):** Percorre a lista inteira e encontra o buraco cujo tamanho seja **o mais próximo possível** do tamanho requisitado. Tenta preservar blocos grandes, mas deixa "lascas" microscópicas tão finas que jamais serão usadas novamente por nenhuma estrutura real.
- **Buddy System:** Exige que a memória seja parcelada sempre em blocos de potências de 2 ($2, 4, 8, 16 \ldots$). Se for pedido 5 Bytes, devolve-se um bloco de 8. É fácil encontrar o "irmão gêmeo" livre de um bloco (seu *buddy*); se ambos estiverem livres, o runtime os une instantaneamente em um buraco de 16, unindo esse a outro de 16 para gerar um de 32, desfragmentando o sistema a baixo custo.

1.9 Coleta de Lixo (Garbage Collection - GC)

Em C e C++, o programador deve comandar a desalocação no Heap manualmente (`free` ou `delete`). Essa gestão provou ser a maior fonte de falhas catastróficas de software por meio de:

- **Memory Leaks:** Esquecer de liberar (causando crashes de Out Of Memory).
- **Dangling Pointers:** Liberar um objeto, mas manter uma cópia do ponteiro, gerando leitura/escrita acidental em memória de outro objeto mais tarde.
- **Double Free:** Liberar o ponteiro duas vezes, corrompendo a estrutura interna do Gerenciador de Memória.

Para modernizar o desenvolvimento, linguagens como Lisp (1959), Java, C#, Python e JavaScript utilizam a **Coleta de Lixo**. O *Runtime* descobre automaticamente quais objetos não servem mais e os recicla.

1.9.1 Conceito de Alcançabilidade

Como o GC sabe o que é Lixo? Ele aplica o conceito de **Alcançabilidade** (*Reachability*).

Um objeto só está **vivo** (útil) se um código futuramente conseguir ler esse objeto. O GC define as "sementes de referência" chamadas de **Root Set** (Conjunto Raiz):

- Toda variável ponteiro declarada em um **Registro de Ativação (Pilha)** atualmente em execução pertence ao Root Set.
- Todo ponteiro mantido no segmento de **Dados Estáticos** (variáveis Globais) pertence ao Root Set.
- **Fecho Transitivo:** Se um objeto A está vivo, e possui uma propriedade que aponta para um objeto B, então B está **vivo**.

Qualquer objeto no Heap que não for encontrado partindo do Root Set é matematicamente **inacessível**. O programa jamais poderá encostar nele novamente. Logo, **é Lixo e pode ser obliterado**.

1.9.2 Estratégia 1: Reference Counting (Contagem de Referências)

Adiciona um contador inteiro ("meta-dado") ao cabeçalho de cada objeto. Sempre que uma nova variável passa a apontar para o objeto, o contador incrementa (+1). Se a variável sair do escopo ou o ponteiro for sobrescrito, o contador decrementa (-1). Se o contador chegar a exatamente zero, o objeto entra em colapso e deleta a si mesmo *imediatamente*.

> △ Importante
>
> **O Câncer da Contagem: Ciclos de Referência.** Imagine que o *Objeto A* possua um ponteiro para o *Objeto B*, e *B* aponte de volta para *A* (como em um grafo fechado ou lista duplamente encadeada). Mesmo se ambos os objetos perderem suas ligações com o Root Set (a Pilha), a ref-count de A e B será 1 indefinidamente, gerando um **Vazamento de Memória** não rastreável. Linguagens com Reference Counting puro (como Swift) exigem declarações complexas de "Weak Pointers" para lidar com isso.

1.9.3 Estratégia 2: Mark-and-Sweep (Marcação e Varredura)

Lida perfeitamente com os ciclos de referência operando em fases (frequentemente acionando um congelamento do programa principal chamado *Stop the World*):

1. **Fase Mark (Marcação):** Começando pelo *Root Set*, o algoritmo roda uma travessia DFS ou BFS no grafo de objetos. Ao tocar um objeto, acende um bit na sua memória (*Mark Bit = 1*).

2. **Fase Sweep (Varredura):** O algoritmo varre a totalidade do espaço físico do Heap. Todo objeto que tiver o *Mark Bit* igual a 0 é desalocado e devolvido à Free-List. Os que sobreviveram têm seu bit resetado para a rodada seguinte.

Vantagem: Resolve o problema de ciclos perfeitamente.
Desvantagem: O programa "congela" durante a coleta. Causa pausas imprevisíveis de latência e a fragmentação da memória ainda persiste.

1.9.4 Estratégia 3: Stop-and-Copy (Copiador Compactador)

O problema do Mark-and-Sweep é que ele apenas recicla os buracos, não resolvendo a fragmentação externa. O Copiador resolve a fragmentação dividindo fisicamente a memória de Heap em duas metades: *From-Space* e *To-Space*.

- Durante a execução normal, objetos nascem colados uns aos outros no *From-Space*. Isso faz com que a **alocação seja extremamente rápida** (apenas incrementar um ponteiro, como se fosse um Stack).

- Quando o *From-Space* enche, o GC congela o programa e começa o *tracing*. Diferentemente da marcação, cada objeto vivo descoberto é **copiado imediatamente** para o *To-Space*, empacotados lado-a-lado, removendo todos os buracos de fragmentação!

- Ao final, o que sobrou no *From-Space* (o lixo) é inteiramente descartado sem necessidade de ser visitado. Os papéis do From e To Space se invertem e o programa continua. O custo é perder **metade da RAM** total da máquina.

1.9.5 Estratégia 4: Generational GC (GC Geracional)

É o algoritmo sofisticado usado hoje em engines como o **V8** (Chrome/Node.js) e pela **JVM** (Java). Ele é estruturado em cima da premissa empírica conhecida como **Hipótese Geracional**:

> *"A esmagadora maioria dos objetos alocados morre muito jovem. Os poucos objetos que sobrevivem a uma coleta provavelmente sobreviverão pelo resto da vida do programa."*

O Heap é particionado em espaços baseados na "idade" do objeto:

- Os objetos novos sempre nascem na **Geração Jovem (Young/Nursery)**. Esta área é mantida com um tamanho pequeno e gerida por um *Stop-and-Copy* muito agressivo, que roda com altíssima frequência. A pausa é na casa de centenas de microssegundos porque a área é minúscula.

- Um objeto que não é "lixo" e consegue sobreviver a N passagens do Copiador Jovem é classificado como "maduro" e é **Promovido** para a **Geração Velha (Old/Tenured Generation)**.

- A Geração Velha ocupa a maior parte da RAM. Por serem objetos persistentes, o GC ignora essa área no dia-a-dia para economizar processamento. Se a Geração Velha encher, roda-se um demorado *Mark-and-Sweep* global (o temido *Major GC*).

1.10 Resumo

✓ title=Pontos-Chave

- Um programa em execução distribui suas necessidades em quatro pilares de layout: **Código, Dados Globais Estáticos, Pilha Dinâmica (Stack) e Memória Livre Global (Heap)**.
- A abstração de sub-rotinas procedural é gerenciada ativamente por meio de **Registros de Ativação (Stack Frames)**, coordenadas pelo ponteiro dinâmico do topo (**SP**) e pela base ancorada em tempo de chamamento (**FP**).
- A simulação de `fat(3)` demonstra concretamente como cada chamada recursiva cria um frame **independente** na pilha, e como o desempilhamento ocorre em ordem inversa (LIFO) restituindo os valores calculados.
- A resolução de variáveis não locais em linguagens baseadas em escopo aninhado léxico ocorre por meio do acompanhamento em runtime do **Access Link** (Link Estático).
- O uso arbitrário do **Heap** propicia a catástrofe da **Fragmentação Externa**. A limpeza preditiva deste estado gerou a abstração do Runtime de **Garbage Collection (GC)**.
- Motores complexos como a V8 e JVM modernizaram a Coleta com o paradigma **Geracional**, usando *Stop-and-Copy* ágil para lidar com os objetos em sua infância, guardando o lento e custoso *Mark-and-Sweep* para objetos veteranos do processo.

1.11 Referências

- **AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D.** *Compiladores: Princípios, Técnicas e Ferramentas.* 2ª ed. Pearson, 2008. Cap. 7 (Ambientes de Tempo de Execução).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler.* 2ª ed. Morgan Kaufmann, 2011. Cap. 6 (The Procedure Abstraction).
- **APPEL, A. W.** *Modern Compiler Implementation in Java.* 2ª ed. Cambridge University Press, 2002. Cap. 13 (Garbage Collection).
- **WILSON, P. R.** "Uniprocessor Garbage Collection Techniques." *Proceedings of the International Workshop on Memory Management (IWMM)*, 1992.