a memória se faz presente na programação na forma de variáveis, arrays, estruturas de dados. programador e ambiente de execução organizam a memória, e isso pode ser percebido especialmente na questão da alocação. as linguagens modernas ganharam espaço, entre outros motivos, pela facilidade que proporcionam ao fornecerem gerenciamento automático de memória, como a coleta de lixo.

1. Variáveis globais e a alocação estática

No loteamento da memória de um programa, nem tudo pode ser definido de antemão. As demandas de um programa muitas vezes se revelam somente durante a execução.

Há alguns elementos de um programa que fogem a essa caracterização. É o caso das variáveis globais. Elas são completamente conhecidas e definidas mesmo antes do programa ser executado. O seu conteúdo pode mudar ao longo da execução do programa, mas o seu tamanho, a sua localização, o seu tipo, etc. permanecem fixos. Qualquer propriedade de uma linguagem de programação que pode ser definida de antemão a partir do exame do código é dita *estática*. Nesse caso, fala-se em alocação estática de memória. Variáveis globais são alocadas estaticamente.

Não somente as variáveis globais são alocadas estaticamente. Há também os casos das variáveis locais marcadas com a palavra *static* (ver Questão 3.1), as variáveis de classe na orientação a objetos, etc.

Se a memória fosse um estacionamento, as variáveis alocadas estaticamente seriam as vagas fixas previamente reservadas para idosos, portadores de necessidades, diretores, etc.

2. Variáveis locais, parâmetros de funções e a alocação dinâmica em pilha

Nas primeiras versões da linguagem Fortran, as variáveis locais de uma sub-rotina eram alocadas estaticamente. Em outras palavras, a variável `a` da sub-rotina `f` já tinha uma posição fixa na memória antes mesmo de o programa começar. Todas as variáveis locais do programa eram assim previamente alocadas.

Como consequência, não era possivel invocar recursivamente uma sub-rotina em Fortran. A recursão implica que várias instâncias de uma mesma função coexistam na memória ao mesmo tempo. Cada uma requer a alocação de seu próprio conjunto de variáveis locais. Embora essas variáveis tenham os mesmos nomes, podem conter valores diferentes. O Fortran já havia limitado a quantidade de cada variável dessas a uma só instância.

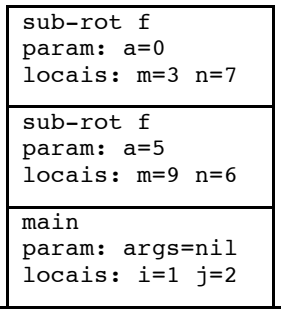
Não é possível prever quantas chamadas recursivas de uma função coexistirão na memória de um programa. Isso talvez seja definido pelos dados de entrada fornecidos durante a execução. Qualquer propriedade de uma linguagem de programação que só pode ser definida durante a execução de um programa é dita *dinâmica*. Variáveis locais de sub-rotinas são alocadas dinamicamente, isto é, enquanto o programa está em execução.

Os parâmetros de uma função também precisam ser alocados. O processo de alocação que serve para variáveis locais também serve para os parâmetros das funções.

A alocação dinâmica de memória para variáveis locais e parâmetros de funções é feita mediante o recurso de uma pilha. Cada posição da pilha corresponde à invocação individual de uma função, com todos os dados referentes a ela, especialmente variáveis e parâmetros. Essa estrutura de dados é chamada de registro de ativação (ou *stack frame*).

topo da pilha ➔

cada bloco representa um registro de ativação



(3) invocação recursiva de `f`. Aqui o parâmetro `a` e as variáveis locais `m` e `n` têm valores próprios.

(2) primeira invocação de `f`.

(1) o programa começa na rotina `main`, que não possui parâmetros, mas duas variáveis locais, `i` e `j`. A rotina fica suspensa quando a sub-rotina `f` é invocada.

3. **Roteiro** de atividades

(3.1) A palavra *static* na linguagem C. Compile e execute o programa `prog1415.c` do livro **Linguagem C**, de Luís Damas, 10.ed , Rio : LTC, 2007, que exemplifica o emprego da palavra *static*. Explique o comportamento do programa.

(3.2) Retornando um ponteiro para uma variável local. O que acontece se uma função retornar um ponteiro para uma variável local? Execute o programa abaixo e observe o resultado. Explique. Use ilustrações das variáveis na memória.

```
#include <stdio.h>
int* f() {
       int i = 10;
       int* p = &i;
       return p;
}
int g() {
       int j = 23;
       return -1;
}
main() {
       int* a = f();
       printf("%d\n", (*a) );
       int b = g();
       printf("%d\n", (*a) );
}
```

4. Alocação dinâmica em *heap*

tempo de vida

Há variáveis criadas durante a execução do programa cujo ciclo de vida não pode se restringir à duração de uma sub-rotina. É o caso, por exemplo, de objetos em Java (alocados mediante a instrução `new`) ou de nós de uma lista encadeada em C (alocados com `malloc`).

não determinística

Esse tipo de variável é alocado numa região da memória reservada para criação e destruição aleatórias de dados, conhecida como *heap* (monte, monturo, e sem relação com o *heapsort*). O *heap* (ou a *heap*?) possui a dinâmica de um estacionamento de shopping center, as regiões são ocupadas e desocupadas sem uma ordem definida. Uma fragmentação se manifesta ali, análogo ao que acontece com o disco rígido de um computador com respeito a seus arquivos.

Assim, a memória de um programa pode ser dividida em três faixas: estática, pilha e *heap*.

Em geral, o uso do *heap* envolve a requisição de alocação (`malloc` em C) e a requisição de liberação da variável anteriormente alocada (`free` em C).

A sequência de ilustrações abaixo representa uma situação típica de funcionamento do *heap*.

HEAP

f	**f**	**f**	

f ← alocar(3)

HEAP

f	**f**	**f**	**g**
g	**g**	**g**	**g**
g	**g**	**g**	

g ← alocar(8)

HEAP

f	**f**	**f**	**g**
g	**g**	**g**	**g**
g	**g**	**g**	**h**
h	**h**	**h**	**h**
h	**h**		

h ← alocar(7)

HEAP

			g
g	**g**	**g**	**g**
g	**g**	**g**	**h**
h	**h**	**h**	**h**
h	**h**		

liberar(f)

HEAP

i	i		g
g	g	g	g
g	g	g	h
h	h	h	h
h	h		

`i ← alocar(2)`

HEAP

i	i		g
g	g	g	g
g	g	g	h
h	h	h	h
h	h	j	j
j	j	j	j
j			

`j ← alocar(7)`

Há 7 espaços no *heap*, mas não são contíguos (fragmentação). Para atender à solicitação, ocorre uma expansão do *heap*.

HEAP

i	i		
			h
h	h	h	h
h	h	j	j
j	j	j	j
j			

`liberar(g)`

HEAP

i	i	k	k
k	k	k	k
k	k	k	h
h	h	h	h
h	h	j	j
j	j	j	j
j			

`k ← alocar(9)`

5. Gerenciamento automático de memória (especialmente coleta de lixo)

Em típicas linguagens de programação com alocação dinâmica em *heap*, o programador tem a responsabilidade de desalocar variáveis que não sejam mais usadas. Muitos problemas de programação têm origem aí. Se o programador desadvertidamente se esquecer de liberar variáveis não usadas, o espaço ocupado por elas no *heap* torna-se inútil. Esse desperdício de memória pode comprometer o funcionamento do programa. Em inglês, esse fenômeno é chamado de *memory leak* (“fuga” de memória, “perda” de memória). Defeitos dessa natureza são difíceis de detectar e depurar, pois não são determinísticos. Mesmo bons programadores estão sujeitos a cometer erros de desalocação do *heap*.

Uma solução para problemas desse tipo é liberar o programador da incumbência de desalocar memória, deixando a tarefa a cargo do *runtime* da linguagem de programação. Assim, o programador não mais desaloca variáveis explicitamente, no que é chamado *gerenciamento automático de memória*. Os ganhos de produtividade e confiabilidade obtidos com o gerenciamento automático de memória são tão significativos, que é uma tendência em linguagens de programação modernas adotar essa estratégia.

A técnica de gerenciamento automático de memória que mais se destaca é a *coleta de lixo*. Consiste em reconhecer objetos no *heap* que não são mais referenciados por nenhuma parte ativa do código, desalocando-os em seguida. Além da coleta de lixo, pode-se mencionar a técnica da *contagem automática de referência* (popularizada pela Apple no Objective C), entre outras.

Há diversos algoritmos de coleta de lixo.

O algoritmo de coleta de lixo mais representativo é o *mark-and-sweep* (marcar e limpar). Ele funciona em duas fases. Na primeira fase, percorre-se a pilha em busca de referências para o *heap*. Todos os objetos do *heap* apontados a partir da pilha são marcados. Recursivamente, todos os objetos do *heap* que esses objetos marcados apontam, também são marcados, até que todos os objetos que possam assim ser “alcançados” tenham sido marcados. Na segunda fase, os objetos que não foram marcados são liberados da memória.

Os algoritmos de coleta de lixo baseado em gerações fundamentam-se no princípio de que objetos alocados há muito tempo no *heap* tendem a permanecer ativos, enquanto que objetos muito recentes têm maior probabilidade de tornarem-se inativos e aptos à desalocação.

O sistema de coleta de lixo do *runtime* da linguagem de programação, chamado de *coletor de lixo*, tipicamente precisa que o programa interrompa seu funcionamento para que a coleta de lixo seja realizada, o que é conhecido como "pare o mundo!". Essa ação, obviamente, tem influência negativa no desempenho do programa. Historicamente, havia uma apreciação de que programas feitos em Java apresentavam uma lentidão (hoje em dia essa percepção não é tão nítida devido à melhora de desempenho do hardware), o que era erroneamente atribuído ao caráter interpretado da máquina virtual. Ora, tendo adotado a estratégia de compilação JIT desde praticamente os primeiros anos da linguagem, não poderia ser essa a causa do desempenho insatisfatório. Na verdade, era a coleta de lixo realizada simultaneamente com o programa a fonte dos retardos.

Diversas técnicas de coleta de lixo têm sido desenvolvidas para contornar problemas de desempenho. Por exemplo, uma ideia é realizar a coleta de lixo incrementalmente, sem precisar percorrer todo o *heap* em cada interrupção. Outra ideia é executar a coleta de lixo em paralelo (possível em processadores multicore). Algoritmos que compactam os dados no *heap* diminuem a fragmentação e tendem a apresentar melhor desempenho.

Muitos programadores Java se sentem tentados a desligar o coletor de lixo, mas isso é impossível. A mera possibilidade de desligamento do coletor de lixo implicaria a perda de todas as garantias de confiabilidade da linguagem com respeito ao gerenciamento de memória. Em Java, o máximo que se faz programaticamente com respeito ao coletor de lixo, é antecipar a sua ação, invocando-o através do método `System.gc()`. Não há garantias que o coletor será invocado imediatamente em seguida.

6. Roteiro sobre o coletor de lixo da JVM

A máquina virtual Java implementada pela Oracle e algumas das suas versões alternativas apresentam controles para otimização de desempenho, especialmente sobre o coletor de lixo. As chaves abaixo devem ser digitadas na linha de comando de invocação da máquina virtual:

- `-server` : a compilação JIT é realizada completamente no início. Melhor desempenho.
- `-Xms`n : tamanho inicial (s de *start*) do *heap*. Exemplos: `Xms512m` (512 Mbytes), `Xms2g` (2 Gbytes).
- `-Xmx`n : tamanho máximo do *heap*.
- `-XX:NewRatio=`n : proporção da área para gerações jovens (n=1 ou n=2 ou n=3).
- `-verbose:gc` : eventos do *heap* aparecem no terminal.
- `-Xloggc:` file : eventos do *heap* são gravados em arquivo com *timestamp*.
- `-XX:+UseSerialGC` : seleciona o coletor de lixo serial.
- `-XX:+UseParallelGC` : seleciona o coletor de lixo paralelo.
- `-XX:+UseConcMarkSweepGC` : seleciona o coletor de lixo *concurrent mark and sweep* (CMS).
- `-XX:ParallelCMSThreads=`n : número de *threads* do CMS paralelo (n=2 ou n=4).
- `-XX:+UseG1GC` : seleciona o coletor de lixo GarbageFirst da Oracle, para grandes *heaps*.
- `-Xincgc` : habilita o coletor de lixo incremental.

A Oracle publicou na web um excelente tutorial sobre a coleta de lixo na JVM. Acesse-o em:
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

(6.1) Preparação. O tutorial incentiva a visualização gráfica do *heap* e da coleta de lixo obtida com a Java VisualVM (`jvisualvm` disponível no diretório `bin` do JDK da Oracle). Instale o plugin da Java VisualVM para coleta de lixo (VisualGC). Baixe o demonstrativo Java 2D (`Java2Demo.jar`). Execute a Java VisualVM e, em seguida, o Java2Demo. Repita o procedimento com Alice, Netbeans ou outras aplicações Java conhecidas.

(6.2) O tutorial. Siga o tutorial. Os valores para o tamanho do *heap* apresentados são muito pequenos e tendem a não funcionar bem. Adote valores maiores como 64m, 256m, etc.

(6.3) Sobrecarga. Execute o benchmark FSTGCMark disponível em
http://java-is-the-new-c.blogspot.com.br/2013/07/tuning-and-benchmarking-java-7s-garbage.html
Ele tende a travar as máquinas. Use tamanhos diversificados para o *heap* que vão de 2g a 12g. Observe como os tempos de interrupção da coleta de lixo se tornam consideráveis com grandes *heaps*. Isso se deve ao fato de a fase de *sweep* ser proporcional ao tamanho do *heap*.

Exemplo de execução baseado no *post* do blog:

```
java -verbose:gc -Xmx12g -Xms12g -XX:+UseG1GC -XX:-UseAdaptiveSizePolicy -
XX:SurvivorRatio=1 -XX:NewRatio=1 -XX:MaxTenuringThreshold=15  FSTGCMark
```

7. Exercícios

(7.1) Imagine uma linguagem de programação em que só alocação estática é usada. Já se sabe que não se pode criar funções recursivas nela. O que dizer de estruturas de dados como listas encadeadas? Há uma solução para isso? É garantido que não há perda de memória nessa linguagem?

(7.2) Discuta o impacto da coleta de lixo no desempenho de um programa, e os melhoramentos que as linguagens modernas têm oferecido nesse quesito.