capítulo 2

Mapa de memória de um processo

> *Existem apenas dois tipos de planos de batalha, os bons e os maus. Os bons falham, quase sempre, devido a circunstâncias imprevistas que fazem, muitas vezes, que os maus sejam bem-sucedidos.*
>
> Napoleão Bonaparte

Conhecer o inimigo e o campo de batalha, geralmente, são pré-requisitos da vitória.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer e identificar as principais áreas da memória;
- conhecer as implicações do gerenciamento eficiente da memória e seu impacto no desempenho do sistema;
- saber em que parte da memória as informações e, consequentemente, as estruturas de dados são armazenadas;
- alocar e liberar blocos de dados alocados dinamicamente na memória.

Para começar

Numa batalha, a movimentação dos soldados e seus armamentos pode ser fator decisivo para a vitória.

Conhecer o campo onde a batalha ocorrerá, facilita muito essa movimentação e deslocamento dos soldados e de seus armamentos.

No caso dos sistemas computacionais, a programação e a manipulação de estruturas de dados podem ser vistas como uma batalha entre o sistema operacional e a grande quantidade de programas que estão em simultânea execução, incluindo aquele que você acabou de desenvolver. Todos batalham para obter o máximo de memória possível e terminar suas tarefas no mais curto espaço de tempo. De outro lado, o sistema operacional tenta colocar uma ordem em tudo isso, limitando alguns acessos e alocações.

Conhecer esse "terreno", isto é, saber onde podem ou não ser alocadas determinadas informações pode melhorar muito o desempenho de seu programa, possibilitando a otimização de todo o sistema.

Neste capítulo, aprenderemos sobre o processo de alocação da memória, como ela é utilizada, quais são os espaços reservados para ela e a forma correta de a utilizarmos. Vamos lá!

Atenção

Conhecer os processos de alocação dinâmica e linear que ocorrem na memória possibilita a construção de programas que utilizam de forma mais inteligente as áreas da memória, consequentemente, sua execução ocorre de maneira mais otimizada, melhorando o desempenho global do programa.

Não sei se isso ocorre com você, mas, quando inicio meu trabalho, a minha mesa está limpinha. No fim do dia, eu não consigo encontrar com facilidade os documentos ou objetos que procuro. Em geral a mesa fica bem bagunçada.

Em sua opinião, qual(is) procedimento(s) poderia(m) ser realizados ao longo do dia para que essa situação não chegasse ao extremo?

Dica

Para responder a essa questão, pense que a mesa representa um espaço limitado de trabalho. No início, está vazia. Depois de um dia de trabalho intenso, muitos objetos, como *notebook*, grampeador, uma porção de folhas de papel, livros, entre muitas outras coisas, ocupam esse espaço.

E então? Em quais soluções você pensou?

Talvez tenha pensado em dividir o espaço da mesa em áreas, e, em cada uma dessas áreas, dispor certos objetos, como da seguinte forma: área das folhas de papel, área do *notebook*, área dos livros, área das ferramentas (grampeador, extrator, etc.).

Essa é uma boa solução.

Também poderíamos colocar sobre a mesa apenas as coisas necessárias para realizar determinada tarefa. Quando essa tarefa fosse concluída, os objetos seriam, então, retirados da mesa, abrindo espaço para a tarefa seguinte. Isso otimizaria o processo e, no fim do dia, a mesa não estaria uma bagunça!

Transferindo isso para os sistemas computacionais, a mesa representaria a memória do computador, que é limitada e é o local onde colocamos os objetos necessários para a realização da maioria de nossas tarefas. A organização é fundamental para a otimização de sua utilização.

Mas como isso pode ser feito na memória? É o que vamos ver agora. Preparado? Então, vamos em frente!

Conhecendo a teoria para programar

A maioria das linguagens de programação utiliza um esquema de segmentação de memória em pelo menos quatro partes ou regiões logicamente distintas: *instruções ou o código do programa*, *variáveis globais*, *pilha* e *heap*. Cada uma dessas regiões tem funções específicas, como mostra o esquema representado na Figura 2.1. Como você já deve ter deduzido, esse esquema

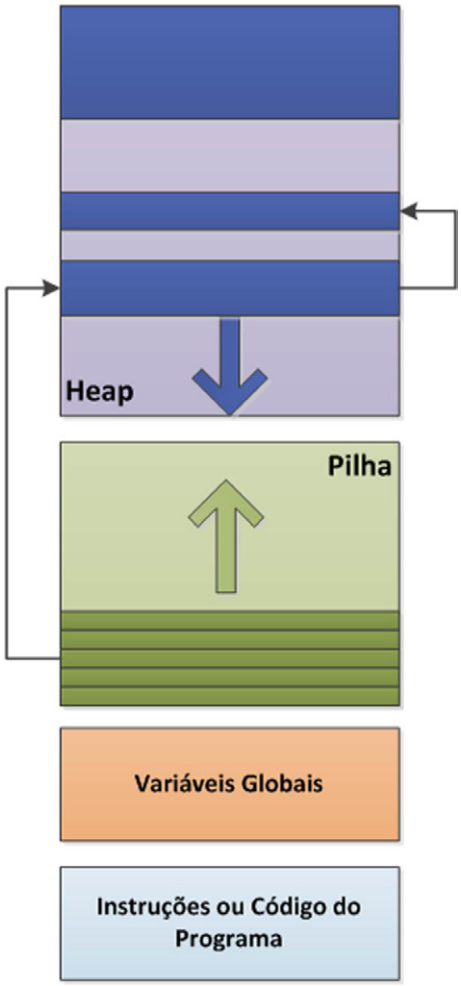

FIGURA 2.1: Esquema lógico de organização da memória nas principais linguagens de programação que utilizam alocação dinâmica.

é apenas um modelo genérico, não representando qualquer implementação real/física.

A alocação estática de memória utiliza, fundamentalmente, a região das variáveis globais. Nesse tipo de alocação, é preciso prever toda a memória de que aquele tipo de dados possa necessitar ao longo da execução do programa. O tamanho máximo de alocação possível nesse tipo é ditado pelo *hardware* (tamanho da memória "endereçável").

O esquema representado na Figura 2.1 permite que se tenha uma ideia dos diferentes papéis assumidos pela memória em linguagens que utilizam alocação dinâmica e linear.

O *heap* é utilizado para a alocação dinâmica dos objetos/dados; já a pilha controla a alocação linear, principalmente para procedimentos sequenciais.

Em linguagem Java, esse processo é feito automaticamente, ao passo que em C/C + + essa alocação é feita manualmente.

Pode-se observar, na Figura 2.1, que o *heap* abriga alguns blocos de informações, que indicam os dados alocados dinamicamente. Olhando com atenção, percebe-se que entre um bloco e outro existe um espaço de memória não ocupado. Esse espaço é chamado de *fragmentação*, e é um dos problemas mais sérios que podem ocorrer quando utilizamos alocação dinâmica de memória, pois não pode ser utilizado, isto é, torna-se um espaço "perdido" na memória.

Na pilha, os blocos podem representar sequências de instruções. Também podem existir indicações da pilha para o *heap*, e de blocos do *heap* para outros blocos do *heap*. Isso acontece quando trabalhamos com ponteiros (listas encadeadas e árvores).

Nas linguagens de programação, essas duas regiões são locais imaginários da memória. Não interessa ao programador saber exatamente onde elas estão, nem como podem ser manipuladas. Mas até mesmo no caso das linguagens Java e Python, que contam com um processo de limpeza automática dessas áreas alocadas e não mais utilizadas (o *garbage collector*, ou coletor de lixo), é interessante conhecê-las, para "forçar" a limpeza nos casos que possam comprometer o desempenho.

Já em linguagens em que a alocação e a liberação de espaço são feitas manualmente, é fundamental ter esse conhecimento para adquirir um controle mais efetivo do processo.

Atenção

A disposição desses elementos na memória, mesmo quando utilizamos C ou C + +, pode variar de compilador para compilador, e de sistema operacional para sistema operacional.

A partir de agora, vamos nos concentrar na alocação dinâmica. Não abordaremos a alocação estática por suas características e simplicidade de utilização.

Em linguagem C, existem quatro funções principais para trabalhar com a alocação dinâmica de memória:

- *Malloc* – permite que seja feita a alocação de uma nova área de memória para uma estrutura. Para tanto, o número de bytes que se deseja

alocar deve ser informado. Se existir esse espaço na memória, a função retorna um endereço de memória, que deve ser colocado em uma variável do tipo ponteiro.

Caso não seja possível alocar o espaço na memória, a função retorna NULL.

Dica

A função *malloc* retorna um ponteiro para o tipo *void*. Portanto, deve-se utilizar o *typecast* para transformar esse endereço no tipo de ponteiro desejado.

```
void *malloc(unsigned int num);
```

- *Calloc* –tem a mesma funcionalidade de *malloc*, entretanto, nesse caso devem ser fornecidos dois parâmetros: o tamanho da área (em bytes) e a quantidade de elementos a serem alocados. A diferença entre as duas funções (*malloc* e *calloc*) é que *calloc* aloca o espaço desejado e o inicializa com zeros.

```
void *calloc(unsigned int num, unsigned int size);
```

- *Realoc* – permite que uma área previamente alocada seja reutilizada, com sua ampliação ou redução. Isso acontece porque, muitas vezes, uma área precisa ser ampliada. Para tanto, deve-se indicar, como parâmetro para essa função, o ponteiro alocado por *malloc* e a indicação do novo tamanho.

```
void *realloc(void *ptr, unsigned int num);
```

- *Free* – permite que uma área previamente alocada seja liberada. Para tanto, deve-se indicar, como parâmetro a essa função, o ponteiro retornado quando fizemos a alocação.

```
void free(void *p);
```

Um exemplo de alocação de memória poderia ser:

Código 2.1

```
#include <stdlib.h>   //Inclusão das duas bibliotecas básicas em C
#include <stdio.h>

char *s;   // Declaração de duas variáveis globais estáticas do tipo ponteiro
int *r;

main()
{
  s = (char *) malloc(2000); //Alocação de 2000 bytes na memória e associação desse espaço com o ponteiro s
  r = (int *) malloc(40*sizeof(int)); //Alocação de 40 espaços do tipo inteiro e associação desse espaço ao ponteiro r.
}
```

Recuperando o esquema lógico mostrado na Figura 2.1, teríamos o seguinte resultado na memória após a execução do segundo comando *malloc* (Figura 2.2):

Note que, depois da alocação, os espaços que antes estavam livres na memória ficaram reservados e são acessados pelos ponteiros *s* e *r*.

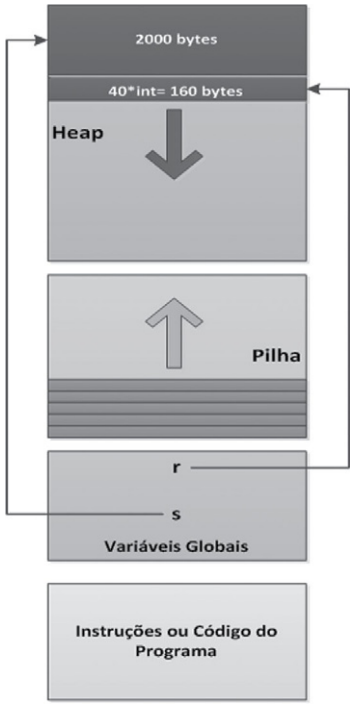

FIGURA 2.2: Resultado da execução do programa no esquema lógico de memória.

Dica

É muito importante que ao fazer a alocação de espaços na memória você cheque se a operação foi realizada. Para tanto, depois de um comando de alocação ou realocação de memória, verifique o ponteiro retornado. Caso ele seja NULL, isso significa que o espaço não foi alocado!

A verificação da realização bem-sucedida de uma operação de alocação pode ser feita da seguinte forma:

Código 2.2

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
  s = (char *) malloc(2000);
  if (s==NULL) {                          // Verifica se s é igual a NULL. Se essa condição for atendida,
    printf("\nErro de alocação de memória"); // significa que o ponteiro s não recebeu o valor do endereço de
    exit(1);                               // memória da posição inicial do espaço alocado. Portanto, não foi
  }                                        // feita a alocação.
  r = (int *) malloc(40*sizeof(int));
  if (r==NULL) {                          // idem para o ponteiro r.
    printf("\nErro de alocação de memória");
    exit(1);
  }
}
```

Após a utilização dos espaços reservados na memória (geralmente no final do programa), é muito recomendável que eles sejam liberados. Isso é feito com a função *free*. O trecho de programa a seguir libera o espaço no fim de sua execução.

Código 2.3

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
  s = (char *) malloc(2000);
  if (s==NULL) {
    printf("\nErro de alocação de memória");
    exit(1);
  }
  r = (int *) malloc(40*sizeof(int));
  if (r==NULL) {
    printf("\nErro de alocação de memória");
    exit(1);
  }

  ...
  {outros commandos/instruções}
  ...

  free(r);   // Libera o espaço alocado na memória e associado ao ponteiro r (160 bytes)


  free(s); // Libera o espaço alocado na memória e associado ao ponteiro s (2000 bytes)

}
```

A utilização da função *calloc* é semelhante à da função *malloc*. Pensando em nosso exemplo, quando fizemos a alocação de 40 elementos do tipo inteiro utilizando *malloc*, também poderíamos ter utilizado a função *calloc* da seguinte forma:

Código 2.4

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
  ...
  r = (int *) calloc(40, sizeof(int));   // Essa seria uma outra forma de fazer a alocação de 40 elementos do tipo
                                         // inteiro, utilizando a função calloc.

  if (r==NULL) {
    printf("\nErro de alocação de memória");
    exit(1);
  }

  ...
}
```

Quando precisarmos de mais espaço, poderemos utilizar a função *realloc*, que permite a alocação de mais espaço. Entretanto, na maioria das vezes, nós o fazemos para expandir o espaço necessário. Assim, em muitos casos, o lugar físico da memória terá de ser alterado. No nosso exemplo, vamos expandir a quantidade de memória associada ao ponteiro ***s*** para 3.000 bytes. Ficaria assim:

Código 2.5

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
  s = (char *) malloc(2000);
  if (s==NULL) {
    printf("\nErro de alocação de memória");
    exit(1);
  }
  r = (int *) malloc(40*sizeof(int));
  if (r==NULL) {
    printf("\nErro de alocação de memória");
    exit(1);
  }

  ...
  {outros comandos/instruções}
  ...

  // necessidade de expansão...

  s = (char *) realloc(3000);   // Expande o espaço reservado inicialmente de 2000 para 3000 bytes.


  free(r);
  free(s);
}
```

Note que, logo depois da primeira alocação de 2.000 bytes, foi alocada outra área. Assim, não será possível expandir linearmente para 3.000 bytes no mesmo lugar. O que acontece é que um primeiro é liberado (automaticamente) e um segundo, com 3.000 bytes, é alocado.

Para melhor visualização depois do comando *realloc*, o que acontece no esquema lógico de memória apresentado na Figura 2.2 é o seguinte (Figura 2.3).

A alocação dinâmica de memória e suas implicações são conceitos importantíssimos que serão tratados com mais profundidade no Capítulo 6.

Para finalizar esses conceitos básicos de alocação, falta verificar como a pilha é utilizada.

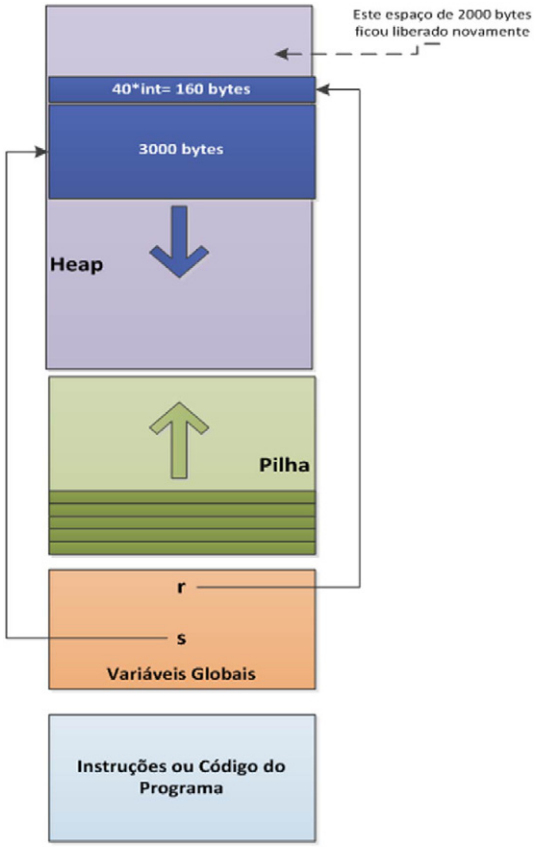

FIGURA 2.3: Resultado da execução do comando *realloc* no esquema lógico de memória.

Basicamente, a pilha organiza a ordem de chamada de funções e suas variáveis locais, estabelecendo, para cada instante da execução, a que cada identificador se refere.

Tomemos como exemplo o seguinte código em linguagem C:

Código 2.6

```
#include <stdio.h>
char *a, *b;

int funcao_X() {
  int localA, localB;
  …
}

main() {
  a = "Global A";
  b = "Global B";
  funcao_X();
}
```

Depois da execução da segunda linha do código anterior o esquema lógico de memória poderia ser definido, conforme apresentado na Figura 2.4.

Após a execução da declaração das variáveis *a* e *b*, o programa é deslocado para a função *main()*.

As duas primeiras linhas da função *main()* associam um valor constante às variáveis. Em seguida, é chamada uma nova função: *função_X()*. Ao chamá-la, as informações de chamada e as variáveis locais são empilhadas. A Figura 2.5 apresenta o esquema lógico de memória após a chamada da *função_X()*.

Note que é identificado o local de onde a chamada provém. Isso serve para o controle retornar à linha subsequente à identificada — no caso, o controle retornará à linha #4 da função *main()* – quando terminar a execução da função chamada.

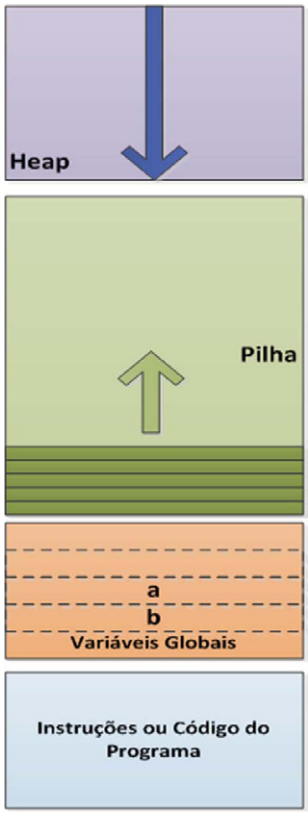

FIGURA 2.4: Resultado no esquema lógico de memória (parcial).

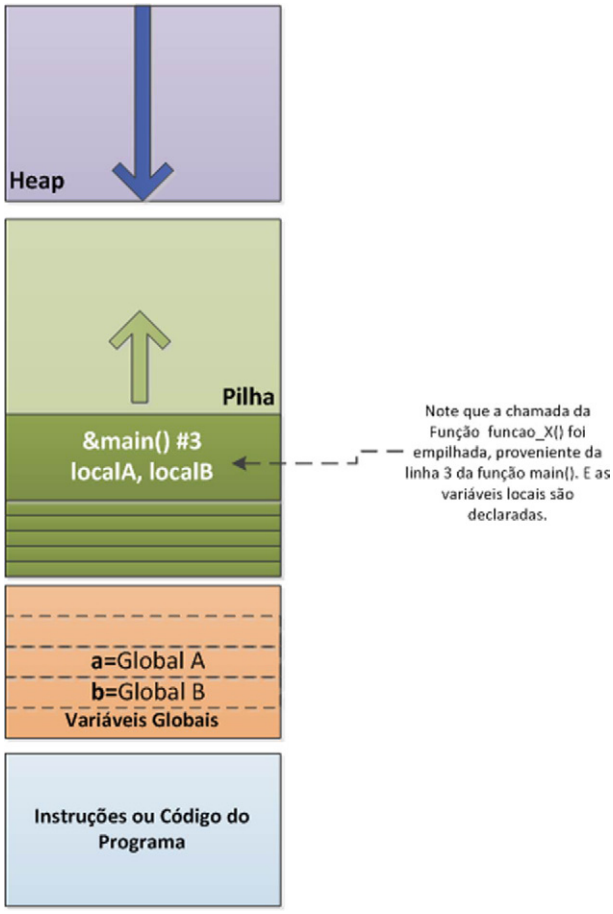

FIGURA 2.5: Resultado no esquema lógico de memória após a chamada da *função_X()*.

Vamos programar

Java e Python

Diferentemente de C ou C + +, programadores Java e Python não têm como gerenciar explicitamente a memória do sistema. Nessas duas linguagens, o programador desenvolve aplicações sem se preocupar com questões de alocação e liberação de memória, pois elas são realizadas automaticamente pela máquina virtual, usando para isso algoritmos específicos.

Portanto, os exemplos simples de alocação e liberação de memória apresentados neste capítulo não apresentam implementações semelhantes nas linguagens Java e Python.

Para fixar

Tomando como base o esquema lógico da memória da Figura 2.6, responda às quatro questões propostas.

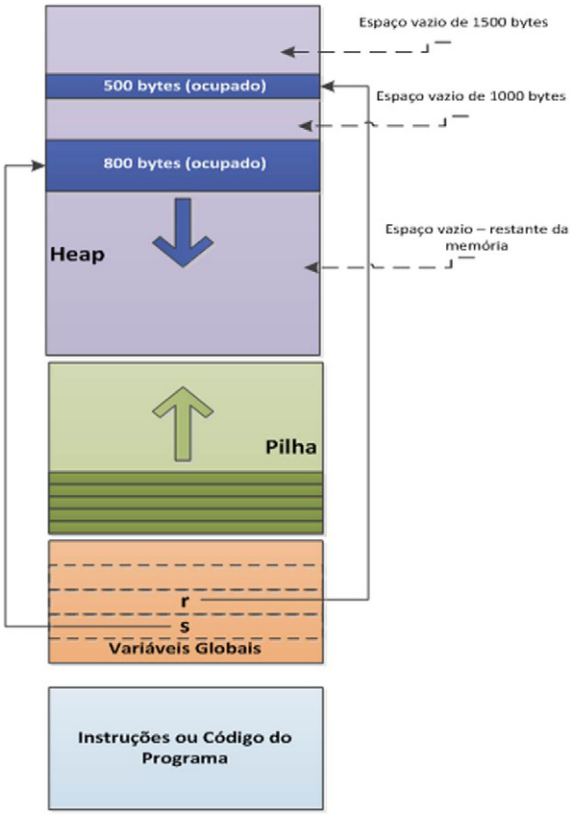

FIGURA 2.6: Estado inicial do esquema lógico da memória.

1. Tomando como base o atual estado da memória, onde seria alocado o espaço de memória alocado pelo seguinte comando:

```
t = (char *) malloc(800);
```

2. Depois que ocorreu a alocação proposta no primeiro exercício, onde seria alocada essa nova área?

```
z = (int *) calloc(100, sizeof(int));
```

3. Como ficaria o esquema lógico da memória com esse terceiro comando?

```
t = (char *) realloc(1000);
```

4. Como ficaria se o comando do exercício três fosse o representado a seguir?

```
t = (char *) malloc(2000);
```

Vamos lá, você consegue!

Papo técnico:

Existem linguagens que, mesmo possuindo o *garbage collector*, permitem ao programador interferir no gerenciamento da memória. É o caso das linguagens C++ e Java.

Para saber mais

Você poderá encontrar mais informações sobre o processo de gerenciamento da memória em livros como *Fundamental Algorithms* (KNUTH) (veja a seção *Dynamic Storage Allocation*).

Existe também um interessante documento, produzido pelo Departamento de Ciência da Computação da Universidade do Texas, em Austin, chamado *Dynamic Storage Allocation: A Survey and Critical Review*. Você pode acessá-lo no site:

http://www.cs.northwestern.edu/~pdinda/icsclass/doc/dsa.pdf

Lembre-se: a questão de alocação dinâmica de memória será aprofundada no Capítulo 6. No momento, você precisa apenas entender como funcionam os mecanismos de gerenciamento da memória para obter melhores resultados ao longo do curso. Bons estudos!

Navegar é preciso

Existe um site bem antigo que trata do gerenciamento de memória. Nele existe um glossário com algumas centenas de palavras, além de bons artigos e uma vasta bibliografia que você poderá consultar. Para isso, acesse o site: http://www.memorymanagement.org/

No site http://www.scirp.org/ journal/PaperDownload.aspx?paperID = 23532, você poderá ter acesso a um artigo que apresenta um objeto de aprendizagem específico para gerenciamento de memória. É uma boa ferramenta para entender o que acontece na memória.

Exercícios

1. Quais são as quatro principais regiões lógicas da memória? Explique cada uma delas.
2. O que é o *garbage collector*? Para que serve e onde pode ser encontrado?
3. Faça um programa que declare duas variáveis, *a* e *b*, do tipo ponteiro. Em seguida, aloque um espaço de 50 elementos do tipo inteiro para *a* e 3.000 bytes para *b*, do tipo char.
4. Tomando como base o esquema lógico da memória da Figura 2.7, como ficariam as alocações propostas pelo conjunto de comandos a seguir no esquema lógico?

```
t = (char *) malloc(1800);
z = (float *) calloc(50, sizeof(float));
```

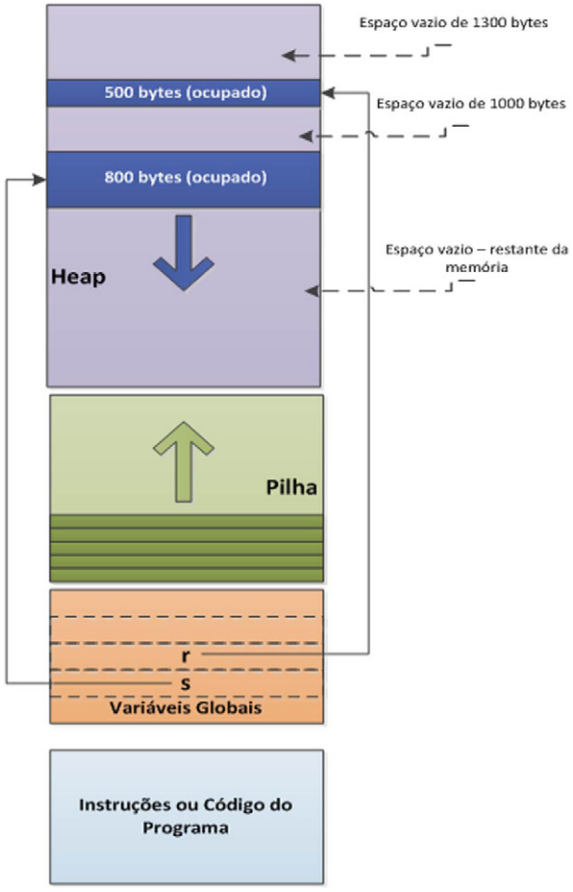

FIGURA 2.7: Estado inicial do esquema lógico da memória.

Glossário

Garbage collector: forma de gerenciamento automático da memória. Trata-se de um coletor que tenta recuperar os espaços ocupados por "lixo", ou seja, por objetos que não são mais utilizados pelo programa. É encontrado em várias linguagens, como Java, C#, Phyton, entre outras.

Heap: designa uma área reservada para alocação dinâmica de objetos na memória. Existem várias formas de organização dessa área; a mais usual é uma lista encadeada de blocos livres. Essa forma é encontrada nas linguagens que permitem o gerenciamento manual de tais áreas. O processo de fragmentação dessa área pode ser um grande problema.

Referências bibliográficas

BATES, B.; SIERRA, K. *Use a cabeça: Java*. Rio de Janeiro: Alta Books, 2007.

CORMEN, T. H. et al. *Introduction to Algorithms: A Creative Approach*. Boston: Addison Wesley, 1989.

SCHILDT, H. *C avançado: guia do usuário*. São Paulo: McGraw-Hill, 1989.

SCHILDT, H. *C completo e total*. 3. e. São Paulo: Makron Books, 1996.

SOBELMAN, G. E.; KREKELBERG, D. E. *C avançado: técnicas e aplicações*. Rio de Janeiro: Campus, 1989.

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.

O QUE VEM DEPOIS...

Agora que você já sabe como funciona o processo de alocação dinâmica na memória e como os dados estão distribuídos, chegou a hora de aprofundar alguns conceitos que possibilitarão a otimização de seus algoritmos. O primeiro deles é a *recursão*. Preparado? Então, bons estudos!