capítulo 6

Alocação dinâmica

> *A mente que se abre para uma nova ideia jamais volta ao seu tamanho original.*

Albert Einstein

É fundamental ser capaz de armazenar novas informações, que, muitas vezes, nem éramos capazes de supor que quiséssemos guardar.

OBJETIVOS DO CAPÍTULO

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

- identificar situações nas quais é importante utilizar a alocação dinâmica;
- alocar e liberar memória;
- criar aplicações com alocação dinâmica de memória.

Para começar

O que devemos fazer se tivermos de armazenar mais informações do que tínhamos previsto?

Algumas vezes, já sabemos a quantidade de informações que vamos armazenar quando implementamos nosso sistema. Podemos ter, por exemplo, 10 números inteiros ou 500 registros com informações sobre nosso amigos. Outras vezes, porém, a quantidade de informações a serem armazenadas não pode ser prevista *a priori*, ou seja, antes que nosso sistema seja usado na prática. Se não tivermos previsto um número suficiente de variáveis ou um tamanho suficiente para nossas estruturas de dados (como vetores ou matrizes), não teremos espaço para armazenar todas as informações.

Imagine que nosso cliente ou o usuário do nosso sistema não possa cadastrar um novo produto ou executar uma venda porque nós não previmos um tamanho suficiente para nossas estruturas de dados. Ele não ficaria nada feliz, não é?

De outro lado, imagine que, para não correr esse risco, declaramos sempre nossas estruturas de dados com tamanhos muito grandes. Nesse caso, elas necessitariam de muita memória, possivelmente mais memória do que a disponível.

Situação complicada, não? Felizmente existem as *estruturas de dados dinâmicas*, que são aquelas que construímos *sob demanda*, isto é, ocupamos a memória conforme ela vai sendo necessária para armazenar os dados. Entre as estruturas de dados dinâmicas podemos citar as *listas ligadas*, as *filas*, as *pilhas*, as *árvores* e os *grafos*. Algumas dessas estruturas podem ser implementadas também com estruturas de dados convencionais, que chamaremos de *estáticas*.

Neste capítulo, vamos aprender a reservar (alocar) memória dinamicamente (sob demanda) para construir essas estruturas de dados.

Vamos lá!

Atenção

Nas estruturas de dados estáticas, a memória alocada tem tamanho predefinido. Mesmo que não armazenemos nada, o espaço reservado não poderá ser utilizado para armazenar outras informações. E, se precisarmos de mais espaço, isso não será possível; para tanto, precisaríamos reprogramar e recompilar o programa.

Já nas estruturas de dados dinâmicas, a memória é alocada conforme vai sendo necessária para armazenar os dados, ou seja, é alocada "em tempo de execução".

Uma questão importante é saber se devemos usar estruturas de dados estáticas ou dinâmicas. Quando sabemos antecipadamente quantas variáveis ou quantos valores vamos armazenar, utilizamos *variáveis estática*s. Podemos também utilizá-las se soubermos que a quantidade máxima de dados que teremos de armazenar será pequena (no máximo 100 valores, por exemplo). De outro lado, se não soubermos a quantidade de dados e se essa quantidade for grande, geralmente usamos *variáveis dinâmicas*.

Propomos, a seguir, algumas situações para você identificar quantas variáveis serão necessárias e qual o seu tamanho (por exemplo: duas variáveis inteiras, ou um vetor de inteiros com 20 posições). Quando não for possível saber a quantidade de dados, responda "variáveis dinâmicas".

1. Armazenar os tamanhos dos lados de um triângulo.
2. Identificar o maior entre 10 números reais.
3. Armazenar as notas das provas de Estruturas de Dados dos alunos da sua turma.
4. Calcular e imprimir a média das notas de cada aluno.
5. Ler e armazenar valores inteiros até que o valor lido seja -1.
6. Obter a temperatura (de um sensor) a cada hora e calcular a maior e a menor temperatura do dia.
7. Guardar a maior e a menor temperatura de cada dia.
8. Armazenar o nome e um e-mail de no máximo 1.000 amigos.
9. Armazenar o nome e um e-mail de todos os amigos.

Conseguiu identificar? Ficou com dúvida em muitos itens?

Vamos aprender um pouco mais para eliminar as dúvidas e utilizar a alocação dinâmica de memória.

Conhecendo a teoria para programar

As variáveis que temos usado até aqui (estáticas) são armazenadas na área de variáveis globais ou na pilha (*stack*) do processo (nosso programa, quando está sendo executado), sendo identificadas e acessadas por seus nomes. Assim, podemos ter duas variáveis inteiras, *i* e *j*, e uma *string* chamada *nome* com 20 caracteres, por exemplo.

As variáveis dinâmicas, de outro lado, são armazenadas no *heap* do processo. Como não sabemos quanta informação vamos armazenar, não podemos dar nomes a ela. Como poderemos acessá-las, então? Nesse caso, o acesso é feito por meio de ponteiros, ou seja, de seus endereços de memória.

Lembre-se

A maioria das variáveis armazena valores (inteiros, caracteres, números reais, ...). Um ponteiro também é uma variável, mas, em vez de armazenar o valor de um dado, armazena um endereço de memória.

No Capítulo 14 do livro *Algoritmos e programação de computadores*, também de nossa autoria, você encontrará detalhes sobre os ponteiros.

Para obter espaço na memória enquanto o programa está sendo executado (chamamos a isso de *alocação de memória*), usamos principalmente o comando *malloc*, que está presente na biblioteca *stdlib.h*. O comando *malloc* possui apenas um parâmetro, que define quantos *bytes* serão alocados. Assim, malloc(4), no Código 6.1, aloca 4 *bytes*, e malloc(20) aloca 20 *bytes* na área de *heap*.

Além disso, o comando *malloc* retorna o endereço do primeiro *byte* alocado. Por que ele faz isso? Para podermos usar esse espaço de memória por meio de seu endereço, já que esse espaço não tem um nome (como *i*, *j* ou *nome*, por exemplo). Como o comando *malloc* não sabe que tipo de informação nós vamos armazenar nesse espaço de memória, ele retorna o endereço (ponteiro) para um *void*. Nós devemos informar qual tipo de informação vamos armazenar, por meio de um *cast*.

Conceito

Um *cast* nada mais é que uma conversão de tipos. O comando *malloc* retorna um *void**, isto é, um ponteiro (ou endereço) para *void*. Como não é isso que vamos armazenar naquele espaço de memória, devemos indicar o que será. Para armazenar um valor inteiro naquele endereço, faremos um *cast* para *int**. Se formos armazenar um número real, faremos um *cast* para *float**, e assim por diante.

Vamos agora alocar espaço para armazenar um número inteiro e uma *string* com 20 caracteres. Em seguida, vamos ocupar esse espaço de memória por meio de seus endereços, armazenados em ponteiros (ponteiro *p* para armazenar o endereço do número inteiro, e ponteiro *s* para armazenar o endereço do [primeiro] caractere).

Código 6.1

```
int *p;                          // ponteiro para inteiro
char *s;                         // ponteiro para caracter
p = (int *) malloc(4);           // supomos que um número inteiro ocupa 4 bytes.
*p = 7;
s = (char *) malloc(20);         // supomos que cada caractere ocupa 1 bye.
printf("Digite a string:\n");
scanf("%s",s);
printf("Conteudo de p=%d  Conteudo de s=%s\n",*p,s);
```

Bem, como estamos desenvolvendo programas cada vez mais complexos, podemos querer executá-los em diferentes ambientes (com arquiteturas, sistemas operacionais ou compiladores diferentes).

E se eles usassem tamanhos diferentes para os tipos de dados, como 2 bytes, 4 bytes ou 8 bytes, por exemplo, para armazenar um número inteiro? Nesse caso, teríamos que alterar o programa para executar na nova instalação.

Para evitar esse problema e dar *portabilidade* aos nossos programas, usaremos a função *sizeof*, que retorna o tamanho do parâmetro em *bytes*. Assim, se mudarmos de ambiente de desenvolvimento, poderemos recompilar o mesmo programa sem precisar alterá-lo. Ele é mostrado no Código 6.2.

Código 6.2

```
int *p;                                  // ponteiro para inteiro

char *s;                                 // ponteiro para caracter

p = (int *) malloc(sizeof(int));         // alocamos espaço para um inteiro

*p = 7;

s = (char *) malloc(20*sizeof(char));    // alocamos espaço para 20 caracteres

printf("Digite a string:\n");
scanf("%s",s);

printf("Conteudo de p=%d  Conteudo de s=%s\n",*p,s);
```

Na Figura 6.1, podemos ver o que aconteceu com a memória na execução desse programa. Vemos que as variáveis *p* e *s* foram alocadas na pilha (*stack*) do processo, nos endereços 100 e 104, respectivamente (Figura 6.1a).

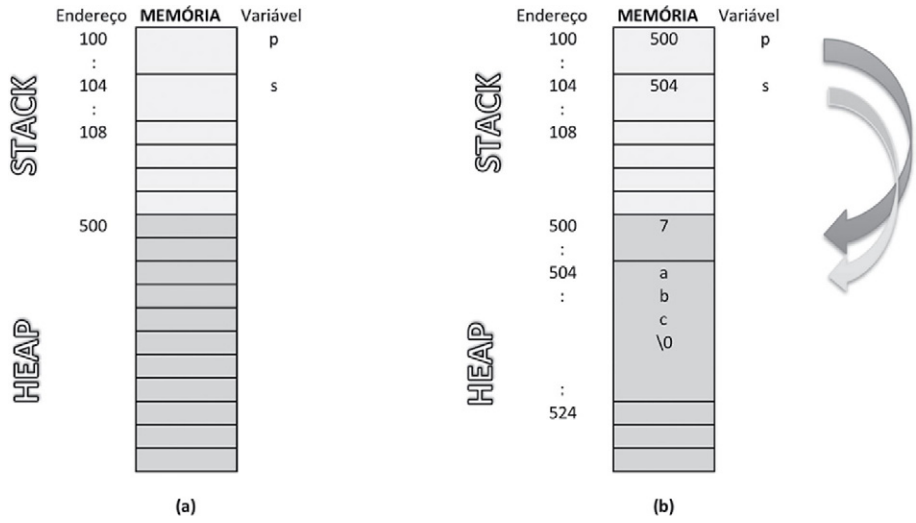


FIGURA 6.1: Alocação de um inteiro e uma *string* de 20 caracteres na memória.

O primeiro comando *malloc* vai alocar 4 bytes (*sizeof(int)*) na área de *heap*, que é onde ficam as variáveis dinâmicas. Informamos ao comando *malloc* que vamos armazenar um número inteiro naquele espaço de memória por meio do *cast(int *)*. Além disso, o comando *malloc* retorna o endereço do primeiro byte alocado (500), que armazenamos na variável *p* (lembre-se de que *p* é um ponteiro para inteiro, ou seja, ele armazenará o endereço de um número inteiro). Assim, vemos, na Figura 6.2b, que a variável *p* recebeu o valor 500. Ela é chamada de ponteiro porque os dados serão armazenados no endereço 500, e não no endereço 100, onde a variável *p* está. Para acessar os dados a partir de *p*, temos que seguir uma seta (ponteiro) até o endereço 500. Assim, quando escrevemos **p* estamos seguindo essa seta e acessando o *conteúdo* do endereço 500. Dessa forma, o comando **p = 7* armazena o valor 7 no endereço 500. Por enquanto, deve parecer estranho fazer isso, mas esse mecanismo vai ser muito útil na construção das próximas estruturas de dados.

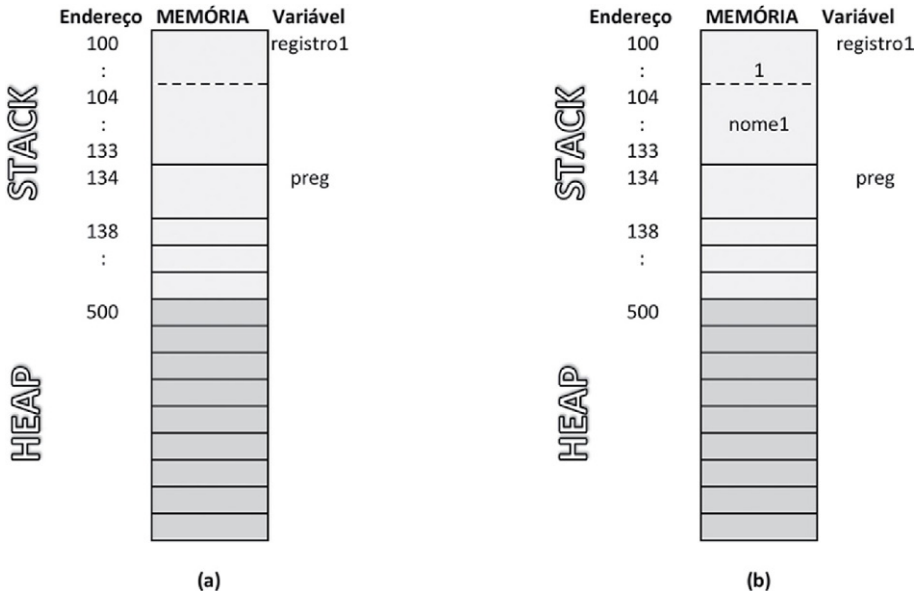


FIGURA 6.2: Registro e ponteiro na pilha (*stack*).

Dica

Quando não há memória disponível no *heap*, o comando *malloc* retorna o endereço especial *NULL* (zero). Se atribuirmos o valor de retorno do comando *malloc* a um ponteiro *p*, por exemplo, podemos testar se *p* é *NULL* para sabermos se a área de *heap* está cheia.

Além disso, como os compiladores C associam o valor *zero* ao *booleano* FALSE e qualquer valor diferente de zero ao *booleano* TRUE, se o valor de *p* for *NULL* (FALSE), não havia memória disponível; se *p* for TRUE, a memória foi alocada com sucesso.

A execução do segundo comando *malloc* é similar. Ele vai alocar 20 bytes na área de *heap*, a partir do endereço 504. Informamos por meio do *cast(char *)* que nesse espaço de memória serão armazenados caracteres. O comando *malloc* retorna o endereço do primeiro byte alocado (504), que atribuiremos à variável *s*. Agora uma diferença: para acessar o endereço 504 por meio do ponteiro *s* não utilizamos o operador* de ponteiros (*s), mas somente o nome da variável (*s*). Por quê? Tente descobrir isso! Dica: lembre-se do funcionamento das *strings*, e de como *strings*, vetores, matrizes e registros são passados como parâmetros para funções. Um bom livro sobre algoritmos e programação poderá ajudá-lo.

Papo técnico

Sabemos que um ponteiro armazena um endereço de memória. Os ponteiros *p* e *s* da Figura 6.1 têm 4 bytes cada, ou seja, 32 *bits*. Isso significa que eles podem armazenar 2^{32} valores diferentes. Como $2^{32} = 4G$, um ponteiro com 4 bytes pode endereçar até 4G endereços de memória. Se cada endereço tiver 1B (1 byte), ele poderá endereçar uma memória de até 4GB.

Nos ambientes com 64 *bits*, os ponteiros têm até 64 *bits*, ou seja, 8 *bytes*. Eles podem armazenar até 2^{64} valores, ou seja, 16 EB (*exabytes*), o que equivale a 16 milhões de *terabytes*!

Nossas próximas estruturas de dados – listas, filas, pilhas e árvores – usarão registros quando implementadas com alocação dinâmica. Vamos ver um exemplo que mostra a diferença no uso de registros quando estiverem na pilha e no *heap*. Conforme for lendo o código, tente imaginar o que ocorrerá na memória do computador.

Código 6.3

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//Definição do registro
typedef struct tipo_registro registro;
struct tipo_registro {
     int codigo;
     char nome[30];
      };
int main()
{
       //Declaração das variáveis
       registro registro1; // registro convencional
       registro *preg;       // ponteiro para um registro

// Atribuição de valores para o registro convencional
registro1.codigo =  1;
strcpy(registro1.nome,"nome1");

//Alocação de um registro na área de heap
preg=(registro *) malloc (sizeof(registro));
if ( !preg ) printf("Não há memória disponível\n");
      else{
              //Atribuição de valores para o registro alocado
              preg->código = 2;
              strcpy(preg->nome, "nome2");

              // Liberação da memória alocada
              free(preg);
              }
}
```

No Código 6.3, vemos a declaração do novo tipo, chamado *registro*, que é uma *struct r* (a *struct r*, definida a seguir, possui um campo inteiro e um campo *string*). Essa declaração não gera código executável, e, assim, não ocupa memória do processo. As variáveis *registro1* e *preg* são variáveis locais (da função *main*), e, assim, são alocadas na pilha do processo. A Figura 6.2a mostra a variável *registro1* alocada no endereço 100 de memória. Ela ocupa 34 bytes, sendo 4 bytes para o atributo *codigo* (inteiro) e 30 bytes para o atributo *nome* (30 caracteres). Ocupa, assim, as posições de memória de 100 a 133. A variável *preg* está alocada no endereço 134 e ocupa 4 bytes (ponteiro).

A seguir, o programa mostra a atribuição de valores para o *registro1*. Como sabemos, o acesso aos atributos do registro é feito colocando-se um ponto entre o nome da variável e o nome do atributo (por exemplo, registro1.codigo). O atributo *codigo* recebe o valor 1 e o atributo *nome* recebe o valor "nome1" (lembre-se de que a atribuição de valores é realizada por meio do comando *strcpy* em *strings*). A Figura 6.2b mostra a memória após as atribuições de valores.

Depois, o programa aloca memória para armazenar um registro. O comando *malloc* solicita um número de bytes equivalente a *sizeof(registro)*, ou seja, 34 bytes no nosso exemplo. Os bytes solicitados pelo comando *malloc* são alocados na área de *heap*, nos endereços 600 a 633. Como o comando *malloc* retorna o endereço do primeiro byte alocado, que é 600, esse valor é guardado no ponteiro *preg*, como mostra a Figura 6.3a. Se não houvesse memória suficiente no *heap*, o comando *malloc* retornaria o ponteiro especial *NULL* (endereço zero), e o programa imprimiria uma mensagem de erro.

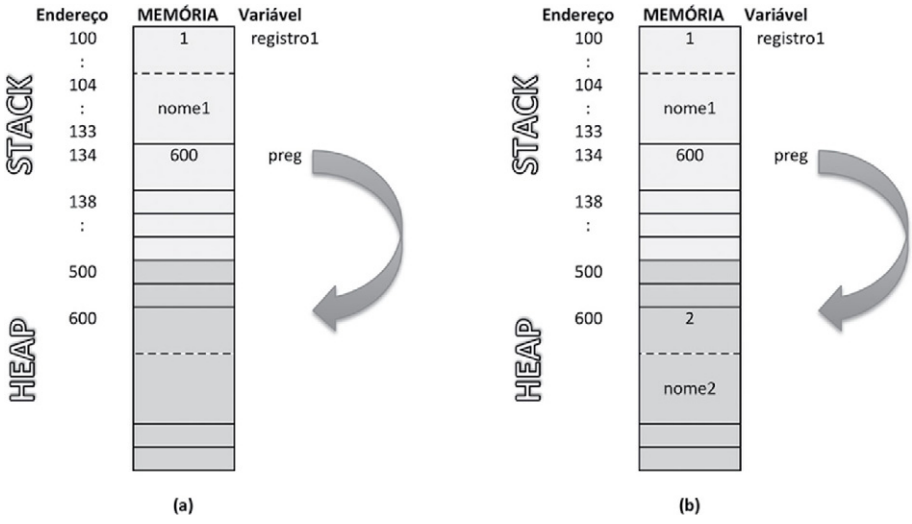


FIGURA 6.3: Alocação de um registro no *heap*.

Atenção

Uma sequência de comandos *malloc* não vai necessariamente alocar posições consecutivas de memória. Ela poderia alocar memória primeiro na posição 600, depois na posição 1200 e então na posição 700, por exemplo.

Então, o programa armazena valores no *heap*, no espaço que acabamos de alocar. Note que não podemos identificar os atributos do registro, como fizemos com o *registro1* (e utilizar, por exemplo, preg.codigo), porque ele não está nas posições de memória da variável *preg*. Para acessar o registro alocado, temos de seguir uma seta (ou um ponteiro) até o endereço de memória contido em *preg*, ou seja, seguir uma seta até o endereço 600. É lá que vamos acessar os atributos do registro. Para fazer isso, em vez de identificarmos os atributos com "ponto", vamos identificá-los com uma "seta", formada pelo caractere *hífen* seguido pelo caractere *maior*, ou seja, - >. Assim, o programa atribui o valor 2 ao atributo *codigo* e o valor *nome2* ao atributo *nome*. A Figura 6.3b mostra a memória após essas atribuições.

Por fim, o programa mostra o uso do comando *free*, que libera memória do *heap* a fim de que possa ser utilizada para armazenar outras informações. No exemplo, o comando *free(preg)* faz com que o espaço de memória iniciado no endereço 600 (valor de *preg*) seja liberado.

Dica

Tome muito cuidado com o uso dos ponteiros. Alguns erros comuns:

1. acessar um atributo por meio de um ponteiro cujo valor é zero (NULL). Nesse caso, ao tentar acessar o endereço zero de memória, o programa vai abortar, isto é, interromper sua execução;
2. inicialize sempre os ponteiros, mas não por meio do comando *malloc*. O comando *malloc* deve ser usado exclusivamente quando tivermos uma nova informação para armazenar, e então solicitaremos mais memória;
3. ao executar o comando *free*, a área de memória será liberada, mas o ponteiro continuará guardando o mesmo valor. Se tentarmos acessá-lo novamente, não teremos como saber qual será o valor armazenado, pois os endereços liberados já podem estar ocupados com outras informações.

Bem, mas aí surge uma questão: se precisássemos ter uma variável do tipo ponteiro para cada registro, teríamos de saber a quantidade de registros a serem armazenados quando estivéssemos programando. Nosso próximo passo, portanto, é conhecer a estratégia que usaremos para acessar um registro a partir de outro registro, e não a partir de uma variável. O segredo é que nossos registros vão armazenar o endereço de um (ou mais) registros, e assim seguiremos uma cadeia de endereços. Parece complicado, não é? Vamos, então, ver um exemplo para entender melhor esse processo.

Em primeiro lugar, vamos redefinir nosso registro da seguinte forma:

```
typedef struct tipo_registro registro;

struct tipo_registro {

     int codigo;

     char nome[30];

      struct tipo_registro *prox;

      };
```

Observamos que, além das informações que queremos armazenar sobre amigos, clientes ou produtos (representadas no exemplo pelos campos *codigo* e *nome*), inserimos outro campo, chamado *prox* (que armazenará o endereço do próximo registro). Como esse campo armazenará o endereço de um registro, será um ponteiro para o registro, isto é, seu tipo será *registro**.

No Código 6.4, temos dois ponteiros para registros: *p1* e *p2*. Como os ponteiros são apenas endereços, precisamos alocar memória para cada um e, assim, armazenar informações neles. Mas desta vez vamos inserir o endereço do segundo registro dentro do primeiro (no campo *prox*). Vamos ver como isso vai funcionar.

Código 6.4

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct tipo_registro registro;
struct tipo_registro {     //Definição do registro
     int codigo;
     char nome[30];
      struct tipo_registro *prox;
      };
int main()
{
registro *p1, *p2;   // ponteiros para os registros
p1=(registro *) malloc (sizeof(registro);    //Alocação dos registro na área de heap
p2=(registro *) malloc (sizeof(registro);
if  ((!p1) || (!p2)) printf("Não há memória disponível\n");
       else{    //Atribuição de valores para o primeiro registro
              p1->código = 1;
              strcpy(p1->nome, "nome1");
              p1->prox = p2;
              //Atribuição de valores para o segundo registro
              p2->código = 1;
              strcpy(p2->nome, "nome2");
              p2->prox = NULL;
              // Exemplos de acesso aos registros
              printf("Codigo de p1: %d\n", p1->codigo);
              printf("Nome de p2: %s\n", p2->nome);
              printf("Endereco contido no campo prox de p1: %d\n", p1->prox);
              printf("Endereco contido no campo prox de p2: %d\n", p2->prox);
              printf("Codigo de p2: %d\n", p2->codigo);
              printf("Codigo de p2: %d\n", p1->prox->codigo);
              }
}
```

Neste programa, temos apenas duas variáveis na função *main*, *p1* e *p2*. Como sabemos, elas são alocadas na pilha do processo. A Figura 6.4a mostra *p1* alocada no endereço 100 e *p2* no endereço 104 de memória.

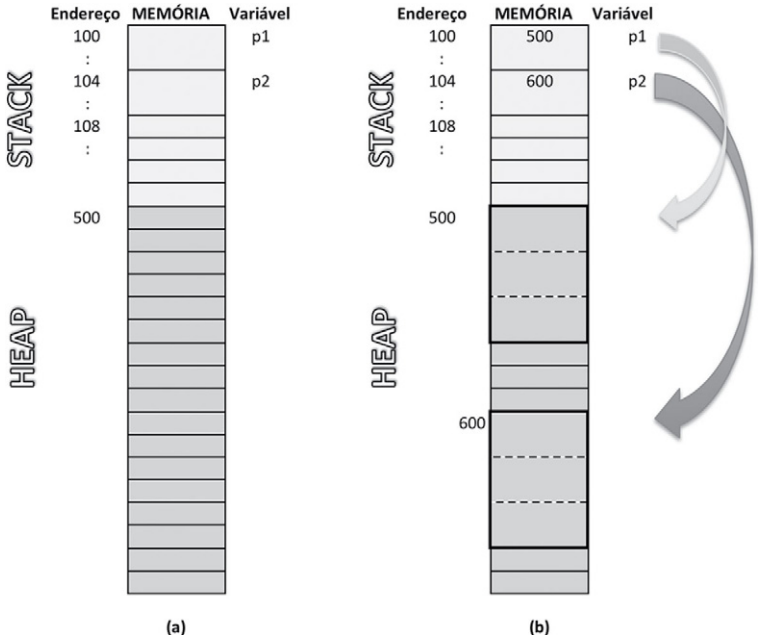


FIGURA 6.4: Alocação de dois registros.

Na execução do primeiro comando *malloc*, uma área de memória suficiente para armazenar um *registro* é alocada, e seu endereço (500) é atribuído à variável *p1*. De forma similar, o segundo comando *malloc* aloca espaço a partir do endereço 600, e esse valor é atribuído à variável *p2*, como mostra a Figura 6.4b.

Os registros alocados nos endereços 500 e 600 foram redesenhados na horizontal, como podemos ver a seguir. Uma vez que o ponteiro *p1* está armazenando 500, que é o endereço do primeiro registro, colocamos o ponteiro *p1* apontando para esse registro. Fizemos o mesmo para *p2*, que contém 600 (endereço do segundo registro).

Na sequência do Código 6.4, atribuímos valores aos atributos de *p1 e p2*, acessando-os por meio de setas, como fizemos anteriormente. O resultado pode ser visto na Figura 6.5.

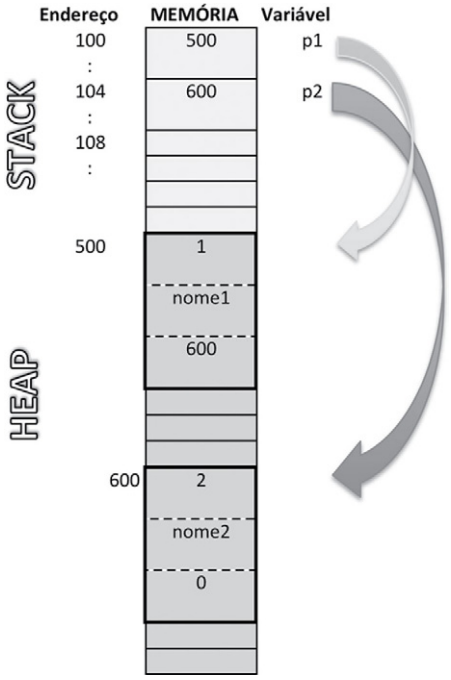
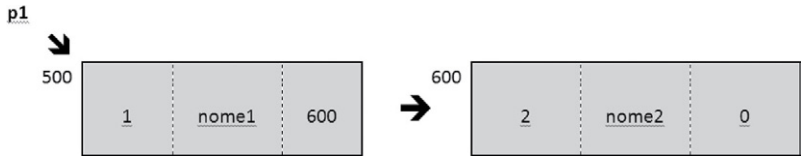


FIGURA 6.5: Atribuição de valores aos dois registros.

Observe que, no campo *prox* do primeiro registro, inserimos o valor 600 (valor de *p2*) para indicar que, após o primeiro registro, está o segundo registro. No campo *prox* do segundo registro inserimos *NULL* para indicar que não há nenhum registro subsequente. Na nossa segunda forma de representação, os registros armazenados na memória teriam a seguinte estrutura:



O primeiro registro, armazenado no endereço 500, contém três campos: o campo *codigo* armazena o valor 1, o campo *nome* armazena o valor

"nome1" e o campo *prox* armazena o valor 600. Como 600 é o endereço do segundo registro, nesse campo desenhamos uma seta para o segundo registro, indicando que ele "aponta" para o registro que está no endereço 600. No campo *prox* do segundo registro está armazenado o endereço especial *NULL* (zero), indicando que não há nenhum registro à frente. Nas próximas figuras não vamos mostrar os valores nos campos de endereços (campo *prox*), apenas as setas indicando para quais registros eles apontam. O ponteiro *NULL* será representado através do símbolo "\".

Há mais um detalhe: você percebeu que é possível acessar o conteúdo do segundo registro sem utilizar o ponteiro *p2*? Observe novamente os dois últimos comandos do programa. O primeiro comando acessa o campo código do segundo registro pelo ponteiro *p2*, cujo valor é 600. Assim, para avaliar a expressão p2->codigo, o compilador[1] verifica primeiro o conteúdo de *p2* (que é 600). Como existe uma seta (->), ele sabe que deve seguir para o endereço 600, e é lá que ele vai procurar o campo *codigo*. O campo *codigo* do endereço 600 contém o valor 2, e é esse valor que será impresso.

E a execução do segundo comando? Para avaliar a expressão p1-> prox-> codigo, o compilador verifica o conteúdo de *p1* (que é 500). Então, ele segue para o endereço 500, e lá vai procurar o campo *prox*. No campo *prox* do registro que está armazenado no endereço 500, ele acha o valor 600. De novo, ele vê uma seta e sabe que tem que seguir para o endereço 600. Lá, ele procura o campo *codigo*, onde encontra e imprime novamente o valor 2.

Assim, vimos que é possível acessar o conteúdo do segundo registro a partir do primeiro registro. Vamos exercitar um pouco e passar às listas ligadas, em que armazenaremos quantos registros quisermos, podendo acessá-los a partir de um único ponteiro (que conterá o endereço do primeiro registro).

Vamos programar

Java

Java faz alocação de memória automaticamente, por isso não é necessário solicitá-la de forma explícita por meio de um comando como *malloc*. Por isso, os Códigos 6.1 e 6.2 foram omitidos. A seguir, você encontrará exemplos para armazenar dados em registros, acessá-los e ligá-los.

1Mais especificamente, o compilador gera o código que fará isso.

Código 6.3

```
public class Registro {

  private int codigo;
  private String nome;
  private Registro prox;

  public Registro(){
  }

  public int getCodigo(){
    return this.codigo;
  }

  public String getNome(){
    return this.nome;
  }

  public void setNome(String nome){
    this.nome = nome;
  }

  public void setCodigo(int codigo){
    this.codigo = codigo;
  }
}

##MAIN##
public class JavaApplication2 {

  public static void main(String[] args) {

    Registro registro1 = new Registro();
    registro1.setCodigo(1);
    registro1.setNome("nome1");
    Registro registro2 = new Registro();
    registro2.setCodigo(2);
    registro2.setNome("nome2");
    System.out.println("Nome:    "    +    registro1.getNome()    +    "    Codigo:    "    +
registro1.getCodigo());
    System.out.println("Nome:    "    +    registro2.getNome()    +    "    Codigo:    "    +
registro2.getCodigo());
      }
}
```

Código 6.4

```
public class Registro {

  private int codigo;
  private String nome;
  private Registro prox;

  public Registro(){
  }

  public int getCodigo(){
    return this.codigo;
  }

  public String getNome(){
    return this.nome;
  }

  public void setNome(String nome){
    this.nome = nome;
  }

  public void setCodigo(int codigo){
    this.codigo = codigo;
  }

  public Registro getProx(){
    return this.prox;
  }

  public void setProx(Registro prox){
    this.prox = prox;
  }
 }

##MAIN##

public class JavaApplication2 {

  public static void main(String[] args) {

    Registro p1 = new Registro();
    Registro p2 = new Registro();

    p1.setCodigo(1);
    p1.setNome("nome1");
    p2.setProx(p2);
    p2.setCodigo(1);
    p2.setNome("nome2");
    p2.setProx(null);

    System.out.println("Codigo de p1: " + p1.getCodigo());
    System.out.println("Nome de p2: " + p2.getNome());
    System.out.println("Endereco    contido    no    campo    prox    de    p1:    "    +
System.identityHashCode(p1.getProx()));
    System.out.println("Endereco    contido    no    campo    prox    de    p2:    "    +
System.identityHashCode(p2.getProx()));
    System.out.println("Codigo de p2: " + p1.getCodigo());
    System.out.println("Codigo de p2: " + p2.getCodigo());
}
}
```

Python

Assim como Java, Python também faz alocação de memória automaticamente, por isso não é necessário solicitá-la de forma explícita por meio de um comando como *malloc*. Por isso, os Códigos 6.1 e 6.2 foram omitidos. A seguir, você encontrará exemplos para armazenar dados em registros, acessá-los e ligá-los.

Código 6.3

```
#Definição do registro
class Registro :
  #constructor
  def __init__(self, codigo=None, nome=None):
    self.codigo = codigo
    self.nome = nome
  #toString()
  def __str__(self):
    return str('Codigo:%s\nNome:%s' %(self.codigo,self.nome))

#main()
#Declaração das variaveis e alocação de um registro na área de heap
registro1 = Registro()
#Atribuição de valores para o registro convencional e alocado
registro1.codigo=1
registro1.nome="nome1"
registro2 = Registro()
registro2.codigo=2
registro2.nome="nome2"
#toString
print registro1
print registro2
```

(Continua)

Código 6.4

```
#Definição do registro
class RegistroLigado :
  #construtor
  def __init__(self, codigo=None, nome=None, prox=None):
    self.codigo = codigo
    self.nome = nome
    self.prox = prox
  #toString()
  def __str__(self):
    return str('Codigo:%s\nNome:%s' %(self.codigo,self.nome))

#main()
#Alocação dos registro na área de heap
p1 = RegistroLigado()
p2 = RegistroLigado()

#Atribuição de valores para o primeiro registro
p1.codigo=1
p1.nome="nome1"
p1.prox=p2

#Atribuição de valores para o segundo registro
p2.codigo=2
p2.nome="nome2"

#Exemplos de acesso aos registros
print "Codigo de p1: %d" %(p1.codigo)
print "Nome de p2: %s" %(p2.nome)
print "Endereco contido no campo prox de p1:", hex(id(p1.prox))
#Só para confirmar se o endereço está correto
print "Endereco de p2:", hex(id(p2))
print "Endereco contido no campo prox de p2:", hex(id(p2.prox))
print "Codigo de p2: %d" %(p2.codigo)
print "Codigo de p2: %d" %(p1.prox.codigo)
```

Para fixar

1. Implemente um programa que aloque memória para três registros, utilizando um ponteiro para cada registro. Lembre-se de preencher todos os campos dos registros e ligá-los (o endereço do segundo registro deve ser armazenado no primeiro; o endereço do terceiro registro deve ser armazenado). Não se esqueça de inserir NULL no último registro (isso é muito importante, como veremos no Capítulo 7).
2. Em seguida, imprima os dados armazenados. Verifique como acessar as informações somente a partir do ponteiro para o primeiro registro (por exemplo, *p1*). Você também pode acessar as informações do segundo e terceiro registros a partir do ponteiro para o segundo registro. Faça o teste!

Para saber mais

Os capítulos 13 e 14 (Registros e Ponteiros) do livro *Algoritmos e programação de computadores* (Piva *et al.*) apresenta detalhes do uso de registros e ponteiros que podem ser muito úteis no estudo da alocação dinâmica de memória e uso de estruturas dinâmicas.

Navegar é preciso

Há vários materiais que exemplificam de outras formas o uso de alocação dinâmica de memória. Você poderá encontrá-los, por exemplo, em http://www.inf.puc-rio.br/~inf1007/material/slides/alocacaodinamica.pdf e em http://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html. No primeiro *link* há vários exemplos e figuras que mostram o uso de alocação dinâmica, incluindo seu uso com vetores. O segundo explica em detalhes o funcionamento dos comandos *malloc* e *free*. Nele você vai ficar sabendo por que alocar grandes porções de memória poucas vezes é melhor do que alocar pequenas porções muitas vezes.

Exercício

Verifique como seu compilador/ambiente de desenvolvimento aloca espaço na área de *heap*. Para isso, aloque memória para vários registros e verifique em quais endereços eles foram alocados. Algumas possibilidades que você pode encontrar:

- os endereços são alocados sequencialmente, em ordem crescente;
- os endereços são alocados sequencialmente, em ordem decrescente;
- os endereços não são alocados em sequência.

A seguir, execute um *loop* de alocação de memória até que não haja mais memória disponível no *heap*. Qual é o tamanho do *heap*?

Por fim, libere memória por meio do comando *free* e, em seguida, faça um *loop* de alocação de memória. Os endereços liberados foram utilizados novamente? Demorou para que isso acontecesse?

Você ainda pode verificar se é possível alterar o tamanho do *heap* (muitas vezes, você poderá precisar de mais memória). A maioria dos compiladores permite que você faça isso por meio de diretivas que alteram o esquema de memória.

Glossário

***Processo*:** trata-se de um programa em execução. Assim, você pode ter centenas de programas instalados no seu HD (ou SSD), mas apenas alguns deles executarão em um dado momento. Processos são, portanto, unidades gerenciadas e escalonadas (colocadas para serem executadas) pelo sistema operacional.

***Portabilidade*:** característica dos sistemas que podem ser portados (ou transportados) para ambientes diferentes (computadores, sistemas operacionais etc.) e executados sem necessidade de modificação.

Referência bibliográfica

PIVA, D. *et al. Algoritmos e programação de computadores*. Rio de Janeiro: Campus Elsevier, 2012.

O QUE VEM DEPOIS

Agora que você já experimentou alocar memória, acessá-la e ligar uma estrutura à outra, chegou a hora de experimentar nossa primeira estrutura de dados dinâmica: as listas ligadas!

Bons estudos!