Introdução à Computação

Rosane Minghim e Guilherme P. Telles

9 de Agosto de 2012

Capítulo 8

Alocação Dinâmica de Memória e Uso de Ponteiros

8.1 Considerações Iniciais

Em um programa de computador, quando são feitas alocações de memória para o conteúdo das variáveis, o compilador reserva um espaço em uma área específica de memória. Vamos chamar esta área de área de dados. Isso acontece para as variáveis declaradas no programa principal.

Em um sistema computacional genérico, as principais estruturas de dados do programa são declaradas aí, e posteriormente passadas por referência a subprogramas para acesso, tratamento e alteração de informações nelas contidas.

Localmente, em cada subprograma, o compilador também faz automaticamente as alocações de memória para as variáveis locais (nelas incluindo os parâmetros passados por valor), numa região que disputa espaço com o espaço de dados mencionados acima (veja Figura 8.1).

Essa região de alocação de variáveis é pré-fixada pelo compilador no início do programa. As variáveis têm tamanho fixo no momento da alocação, originando o termo alocação estática. Mesmo os vetores, que possuem quantidade variável de elementos, são na realidade definidos com um tamanho máximo (fixo) que não pode ser alterado durante a execução do programa. Com base na alocação estática, fica difícil gerenciar estruturas de dados que possuem flutuação de conteúdo, ora expandindo ou contraindo dependendo das condições do sistema e dos seus dados.

Para apoiar principalmente essas estruturas de tamanho variável, normalmente as linguagens de programação admitem acesso a uma região de memória externa àquela inicialmente reservada para o programa, chamada

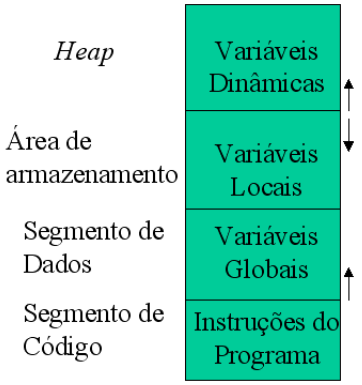


Figura 8.1: Modelo conceitual da alocação de memória de um programa

de *Heap*. Enquanto que a área de alocação estática tem um tamanho fixo, definido no momento de carga do programa, a heap pode crescer quase que indefinidamente, em princípio expandindo até ocupar toda a memória disponível do computador. Conceitualmente estes conceitos de áreas delimitadas pode ser entendido na forma ilustrada na Figura 8.1.

Na heap a alocação de memória possui a vantagem de ser variável, ou seja, o programador pode reservar uma quantidade de memória conveniente ao objetivo que pretende atingir, utilizar esta memória armazenando e manipulando dados, e 'liberar' a memória no momento em que não é mais necessária. Esta flexibilidade, que não existe nas formas de definição de variáveis estáticas vistas nos demais capítulos, é muito útil para manter a ocupação de memória do sistema sob controle. Este tipo de manipulação do espaço de armazenamento é denominado *alocação dinâmica de memória*.

A desvantagem do processo é que o gerenciamento do espaço ocupado exige um grau maior de controle do programador, e também um cuidado adicional de desocupar espaços alocados que não são mais necessários para o programa.

Embora esta forma de alocação, em geral, não deva influenciar a solução algoritmica de alto nível de um problema, ela adiciona um grau de complexidade nos comandos de armazenamento e utilização de variáveis. Com isso, o pseudo-código irá fornecer elementos para utilização explícita de alocação dinâmica.

Este capítulo tem por objetivo demonstrar a sintaxe dos comandos de alocação, acesso e liberação de área de memória heap, bem como alguns exemplos de utilização do recurso em pseudo-código.

A próxima seção descreve o tipo de dados utilizado para gerenciar alocação dinâmica de memória (denominado apontador), e exemplifica sua utilização.

As seções seguintes demonstram sua utilização e funcionamento e exemplificam algumas soluções de problemas usando ponteiros.

8.2 Apontadores em pseudo-código

O tipo de dados que apóia a alocação e liberação de memória dinâmica é demominado **apontador** ou **ponteiro**. Ele pode ser declarado de forma estática, isto é, como qualquer outra variável definida até o momento. A declaração de um apontador em pseudo-código utiliza o caracter '> ', para indicar que aquele é um ponteiro. Além disso, é preciso definir, no momento da declaração, qual o tipo de dado que ele vai apontar, isto é, que tipo de dados será alocado em memória dinâmica com a utilização do apontador. A declaração de uma variável apontador assume a forma:

```
variável

  identificador: > tipo-apontado
```

O exemplo abaixo ilustra declarações de apontadores.

Exemplo 8.1

```
tipo
  aluno = registro
             nome: cadeia[30]
             número: cadeia[8]
             ano_nascimento: inteiro
          fim registro


var
  apont_aluno : > aluno
  apont_índice: > inteiro
  apont_valor: > real
```

No exemplo acima a variável identificada como apont_aluno é do tipo apontador uma vez acionada para alocar e liberar memória, irá apontar para um item de dado do tipo aluno, isto é, um registro. Da mesma forma, a variável apont_valor está preparada para manipular alocação dinâmica e acesso a valores reais e apont_índice a valores inteiros.

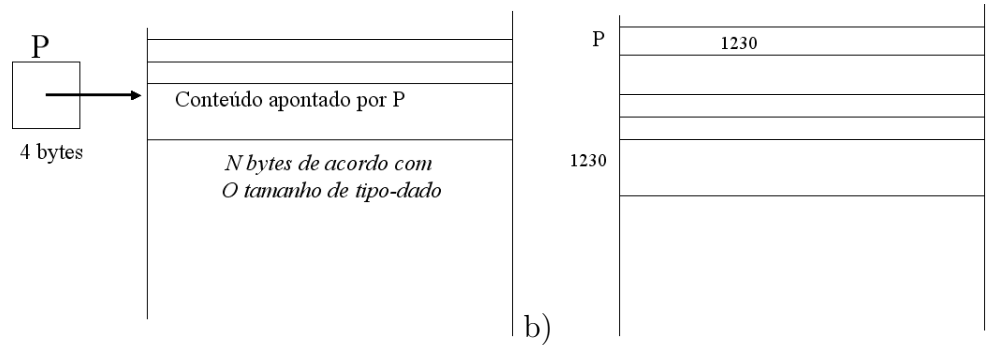


Figura 8.2: a) Representação do uso de um apontador. b)Efeito em memória do uso do apontador representado em a)

Antes porém de estudar alocação dinâmica, devemos entender o funcionamento do tipo de dados apontador. Uma variável do tipo apontador contém um endereço de memória. Este endereço referencia a posição de memória onde o dado 'apontado', conforme indicado na Figura 8.2a. Com este comportamento, um apontador pode referenciar qualquer posição de memória acessível, não apenas as áreas de memória heap. O apontador em si é uma variável comum, que contém um endereço de memória, e que está armazenada em memória como as demais variáveis do programa, conforme indicado na Figura 8.2b. Nesta figura, o endereço armazenado em P é 1230 (em decimal). O conteúdo desta posição de memória é de fato o valor 'apontado' por P.

Um variável do tipo apontador, uma vez declarada, não possui valor válido armazenado, ou seja, não se pode usar o conteúde de memória apontado por ela. Ela precisa ser inicializada. A forma de inicialização de um apontador é feita através da constante NULO. O comando para inicialização é dado a seguir.

```
apont_aluno ← NULO
```

O efeito, conforme ilustrado na Figura 8.3, é o de 'apontar para o vazio'. Com isso é atribuído à variável um valor que pode posteriormente ser testado como nulo. Não é uma boa política deixar de inicializar um apontador.

Um apontador pode apontar qualquer posição de memória válida (existem regiões de memória que não são acessíveis a partir de um programa genérico e são protegidas pelo sistema operacional). Para manipulação de apontadores, a dos conteúdos apontados por eles existem operadores.

Um dos operadores muito usados no contexto do uso de ponteiro é o operador unário de endereço. Este operador, indicado pelo símbolo '>' (o

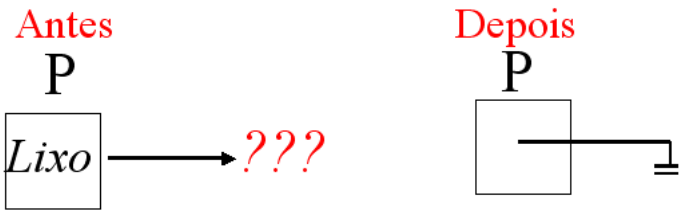


Figura 8.3: Resultado da inicialização do apontador p ←NULO

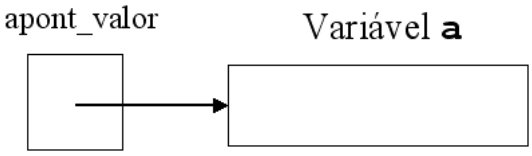


Figura 8.4: Utilização de um apontador para apontar uma variável pré-existente.

mesmo da declaração de ponteiros), fornece o endereço em memória onde uma determinada variável está armazenada.

Assim, supondo a declaração:

```
variável
   a:real
```

o comando:

```
> a
```

devolve o endereço da variável `a`.

Com o uso do operador acima, é possível utilizar um apontador para referenciar uma variável já alocada em memória. Ou seja, asssumindo as declarações do Exemplo 8.1, o comando:

```
apont_valor ← > a
```

aponta para a variável `a`.

A Figura 8.4 ilustra o efeito do comando acima.

Um outro operador associado geralmente ao uso de ponteiros é o operador unário de conteúdo. Este operador recupera o valor armazenado em memória na posição apontada por um determinado apontador. Seu símbolo é o caracter '^'. Um exemplo é dado a seguir.

Exemplo 8.2 *Utilização do operador de conteúdo.*

```
1  variável
2     a: real
3     apont_valor: > real
4
5
6  ...
7
8     a ← 2,5
9     apont_valor ← > a
10    escreva (apont_valor^)
11
12 ...
13
```

No Exemplo 8.2, como a variável `apont_valor` aponta a posição de memória da variável `a` (linha 9), quando o conteúdo de apont_valor é impresso (linha 10), o valor escrito é 2,5 (o mesmo valor de `a`, já que ambos referenciam a mesma posição de memória).

O comando de conteúdo também pode ser usado à esquerda do sinal de atribuição, isto é, o comando abaixo é válido:

```
apont_valor^ ← 3,75
```

Por consequência de se utilizar apontadores para referenciar elementos que já existem, é preciso cuidado na manipulação de seus conteúdos. Como ilustração, no Exemplo 8.2 após a linha 9, todas as modificações feitas usando `^apont_valor` alteram o valor de `a` e vice-versa.

Pergunta 2 *O que é impresso no final do trecho de algoritmo do Exemplo 8.3?*

Exemplo 8.3

```
variável
   a: real
   apont_valor: > real


...
```

```
   a ← 2,5
   apont_valor ← > a
   apont_valor^ ← 4,2

   escreva (a)

...
```

Resposta 2 *No final do trecho do Exemplo 8.3 é escrito o valor* `4,2`.

O exemplo a seguir apresenta uma sequência de comandos com o uso de ponteiros. Acompanhe a execução pela Figura 8.5.

Exemplo 8.4 *Manipulação de apontadores para variáveis.*

```
1  variável
2     x:inteiro
3     p,q:>inteiro
4
5  x ← 10
6  p ← >x
7  q^← 30
8  q ← p
9  q^← p^+ 10
10 escreva("Resultado = ",p^)
```

O resultado do Exemplo 8.4 após linha 10 é a impressão:
`Resultado = 20`.

Note o erro do Exemplo 8.4 na linha 7. No momento em que o comando tenta atribuir um valor à região de memória apontada por q, este ponteiro ainda não aponta nenhuma região de memória, ou pior, aponta uma região genérica de memória. Isso gera um erro de execução do algoritmo.

Um ponteiro pode apontar tipos compostos, como registros. O exemplo a seguir ilustra esta aplicação.

Exemplo 8.5

```
1  tipo
2    rec_dados = registro
3                    dado: real
4                    nome: cadeia[10]
```

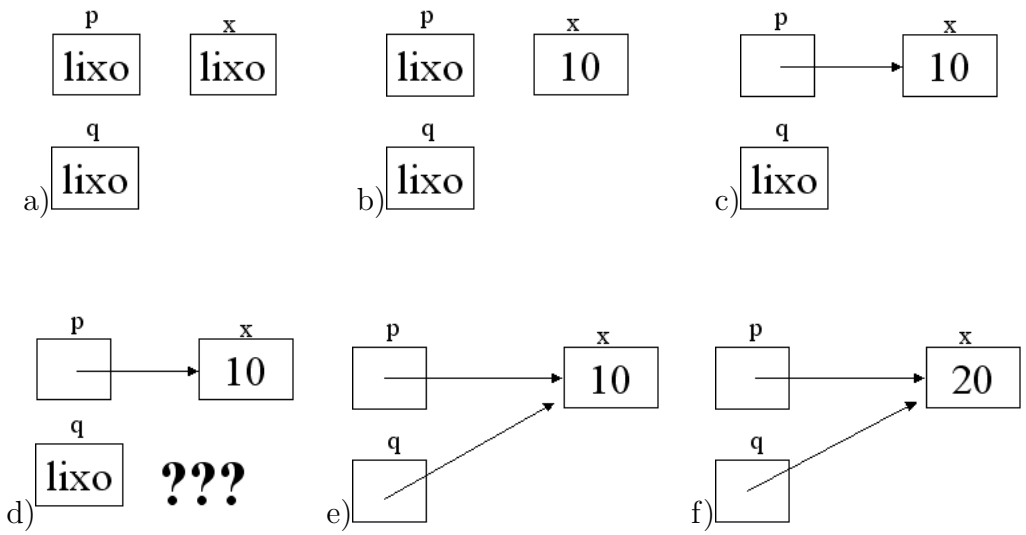


Figura 8.5: Manipulações de um apontador, conforme Exemplo 8.4. a) Resultado após linha 3. b) Resultado após linha 5. c) Resultado após linha 6. d) Resultado após linha 7. e) Resultado após linha 8, ignorando a linha 7. f) Resultado após linha 9.

```
5                  fim registro
6
7   variável
8     x:rec_dados
9     p:>rec_dados
10
11  p ← x
12  p^.dado ← 2,3
13  p^.nome ← 'meu_nome'
14  p^.dado ← p^.dado + 5,0
15  escreva ('dado = ',p^.dado,' nome = ',p^.nome)
16  escreva ('dado = ',x.dado,' nome = ',x.nome)
```

No Exemplo 8.5, primeiramente são declaradas duas variáveis: `x` (linha 8) e `p` (linha 9).

Em seguida, a variável `p` passa a apontar `x` (linha 11). A partir deste momento, conforme exemplificado anteriormente, a manipulação de `x` ou de `p^` leva ao mesmo resultado.As duas impressões ao final do algoritmo são, portanto, equivalentes, ou seja, o algoritmo imprime:

```
dado = 7,3 nome = meu_nome
dado = 7,3 nome = meu_nome
```

Ou, seja, nas linhas 12, 13 e 14, o valor de campos da variável `x` estão sendo alterados. Respectivamente, a codificação daquelas três linhas poderia ter sido feita da forma abaixo, obtendo-se o mesmo resultado:

```
x.dado ← 2,3

x.nome ← 'meu_nome'

x.dado ← x.dado + 5,0
```

Encadeando Registros

Para familiarizar o leitor com um uso clássico de ponteiros, exemplificamos a seguir o encadeamento de registros usando ponteiros. Acompanhe o exemplo e tente desenhar o resultado, usando a visualização empregada até o momento para ponteiros.

Exemplo 8.6 *Encadeamento básico de registros*

```
1  tipo
2    rec = registro
3            valor: real
4            pont: >rec
5          fim registro
6
7  variável
8    x,x1:rec
9    p,q:>rec
10
11 x.valor ← 10,5
12 x.pont ← >x1
13 p ← >x
14 x1.valor ← 0,2
15 x1.pont ← NULO
16 q ← x.pont
```

No Exemplo 8.6 são declarados dois registros do tipo rec (`x` e `x1`), e dois ponteiros para este tipo (`p` e `q`). O tipo rec possui dois campos: um de valor real e outro do tipo ponteiro. Essa definição recursiva é permitida em pseudo-código e apenas significa que um dos campos é um ponteiro para um dado que possui a mesma estrutura.

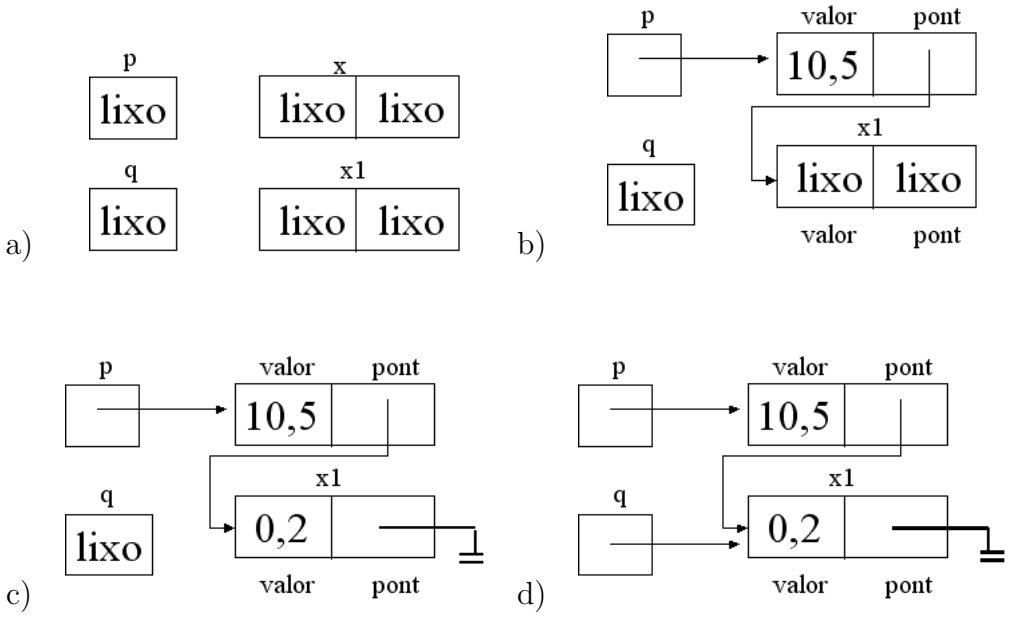


Figura 8.6: Encadeamento de registro usando apontadores, conforme Exemplo 8.6. a) Resultado após linha 9. b) Resultado após linha 13. c) Resultado após linha 15. d) Resultado após linha 16.

Nas primeiras linhas do algoritmo (11 e 12), os campos da variável `x` estão sendo atualizados. Um valor é atribuído ao primeiro campo, e o ponteiro do segundo campo aponta para a variável `x1`.

Em seguida o ponteiro `p` passa a apontar a variável `x`.

Na sequência a variável `x1` é atualizada.

Note o comando na linha 16. Nela, o ponteiro `q` aponta o mesmo que é apontado pelo campo pont da variável `x`, ou seja, a variável `x1`. A situação final e as intermediárias do algoritmo do Exemplo 8.6 são ilustradas na Figura 8.6.

Após a execução dos comandos do Exemplo 8.6, o acesso aos campos dos dois registros pode ser feito de várias maneiras. Por exemplo, para verificar se o ponteiro do campo pont de `x1` é nulo, qualquer um dos comandos abaixo é válido:

```
se x1.pont = NULO então

se x.pont^.pont = NULO então

se p^.pont^.pont = NULO então
```

```
se q^.pont = NULO então
```

Pergunta 3 *No contexto do Exemplo 8.6, qual é o resultado dos comandos abaixo após a execução da linha 16?*

```
escreva(p^.valor)
escreva(q^.valor)
escreva(p^.pont^.valor)
escreva(x.valor)
escreva(x.pont^.valor)
escreva(x1.valor)
```

Resposta 3 *O resultado dos comandos de impressão da Pergunta 3 é:*

```
10,5
0,2
0,2
10,5
0,2
0,2
```

O texto e exemplos aprestados até o momento serve para ilustrar o funcionamento e operações básicas com ponteiros e variáveis do tipo ponteiro.

No entanto, a utilidade de apontadores no contexto exemplificado é muito limitado. Existe pouco objetivo em utilizar ponteiros para endereçar variáveis já alocadas em memória.

A aplicação mais relevante de ponteiros é a de apoiar alocação dinâmica, isto é, permitir o gerenciamento de espaço de memória adicional àquele da memória de dados pelo programador. A próxima seção trata deste uso de apontadores.

8.3 Uso de Apontadores para Alocação Dinâmica

Apontadores são utilizados para apoiar a reserva de espaço em memória heap, bem como sua liberação após o uso. Na verdade, cada sistema operacional tem sua própria forma de permitir alocação e liberação de memória, que é controlada de forma transparente ao usuário. No entanto, do ponto de vista de programação, alocações são feitas com base no tipo do ponteiro, ou melhor, no tamanho que aquele tipo ocupa.

Imagine as seguintes declarações de ponteiros, com base nos tipos definidos previamente:

```
variável
  p1:rec
  p2:rec_dados
  p3: >real
```

Quando uma alocação for feita com base em `p1`, será alocado espaço necessário para conter uma variável do tipo `rec`. Quando uma alocação for feita com base em `p2`, será alocado espaço necessário para conter uma variável do tipo `rec`. Quando uma alocação for feita com base em `p3`, será alocado espaço necessário para conter um número real. A liberação de memória com base nesses apontadores também se dá da mesma maneira.

A operação para alocação de memória é denotada por:

aloque(p)

onde p é qualquer apontador válido.

A operação aloca espaço de memória suficiente para armazenar um dado do tipo apontado por p. Na variável p é devolvido o endereço inicial da memória alocada. A situação após a alocação é similar àquela ilustrada na Figura 8.2.

A operação para liberação de memória em pseudo-código é denotada por:

libere(p)

onde p é qualquer apontador apontando uma região válida de memória.

Abaixo é oferecido um exemplo simples de alocação, utilização e liberação de memória através do uso de ponteiros.

Exemplo 8.7

```
variável
  p,q:>inteiro

aloque(p)
p^ ← 10
q ← p
q^ ← p^ + 10
escreva(p^)
libere(p)
```

A Figura 8.7 ilustra o efeito do algoritmo passo a passo.

O efeito de alocar um ponteiro faz com que uma área da heap seja reservada para o conteúdo apontado por ele. O efeito de liberar um espaço apontado por um apontador é o de 'devolver' para a heap o espcaço ocupado. Quando esse espaço é devolvido, nenhum ponteiro previamente usado para apontar esta região é alterado (veja efeito do comando `libere(p)` na

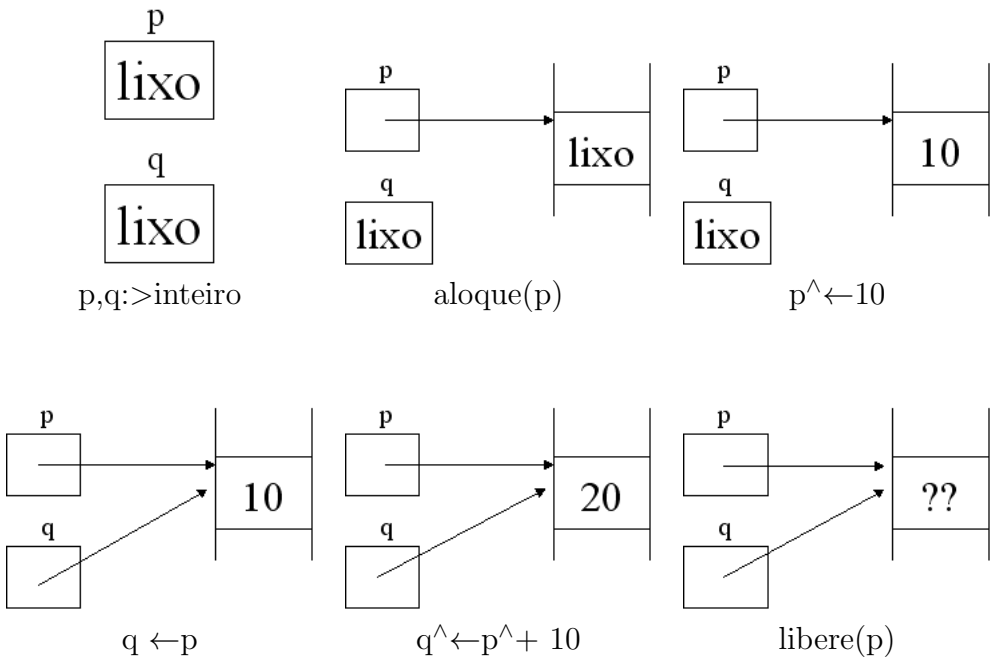


Figura 8.7: Alocação dinâmica e manipulação da Heap, conforme Exemplo 8.7

Figura 8.7). No entanto, o espaço apontado pode ser utilizado para um outro fim, já que está liberado.

Assim, é recomendável que, para que se evite acidentes, toda vez que a região apontada por um ponteiro é liberada, o ponteiro seja anulado em seguida. No Exemplo 8.7, isso significa anular os ponteiros `p` e `q`, para evitar acidentes. A Figura 8.8 demonstra a operação.

Sugestão 12 *Sempre anule o ponteiro após liberação de memória*

O exemplo a seguir apresenta a sequência de execução de um encadeamento como aquele apresentado no Exemplo 8.6, agora utilizando apenas

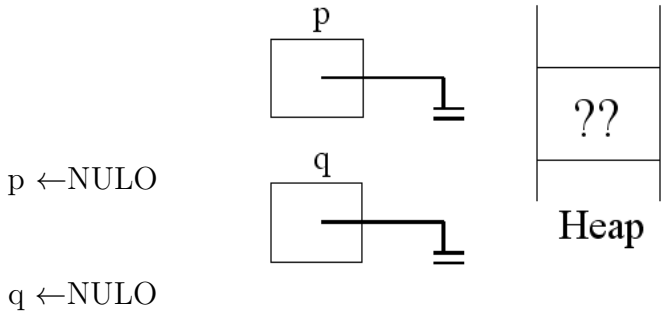
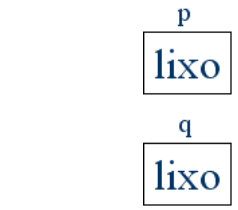
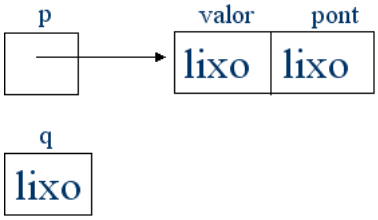
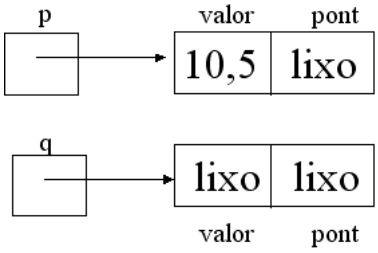
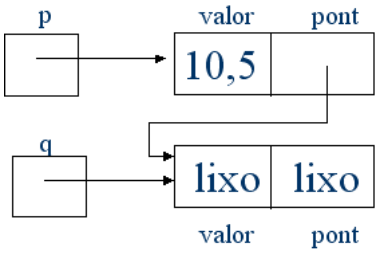
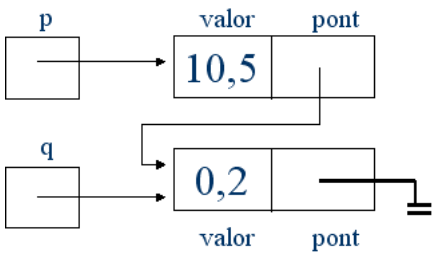


Figura 8.8: Anulando ponteiros após liberação de memória

alocação dinâmica.

Exemplo 8.8

tipo rec = registro *dado: real* *pont: >rec* *fim registro* *variável p,q :>rec*	
aloque(p)	
p^.valor←10,5 *aloque(q)*	
p^.pont ←q	
q^.valor ←0,2 *q^.pont ←NULO*	

No Exemplo 8.8, p e q são utilizados para reservar espaço em memória suficiente para armazenar dois registros: um apontado por p e outro apontado por q. Com base apenas nos dois ponteiros, é possível alocar e armazenar valores nos registros apontados por eles.

Pergunta 4 *No contexto do Exemplo 8.8, qual é o resultado dos comandos abaixo após a execução da última linha?*

```
escreva(p^.valor)
escreva(q^.valor)
escreva(p^.pont^.valor)
```

Resposta 4 *O resultado dos comandos de impressão da Pergunta 4 é:*

```
10,5
0,2
0,2
```

Ainda no Exemplo 8.8, veja que o ponteiro q é desnecessário. É possível contruir a estrutura sem a utilização dele. Sua utilização facilita o entendimento, mas uma vez montada a estrutura, ele não é mais necessário, já que todos os elementos (todos os campos dos dois registros) são acessíveis apenas a partir de p.

Por exemplo, se após a última linha do Exemplo 8.8 executássemos o comando:

`q ←NULO`

como faríamos para:

Pergunta 5 *acessar o valor 0,2?*

Resposta 5 *p^.pont^.valor*

Pergunta 6 *verificar se o segundo registro da cadeia é nulo?*

Resposta 6 *Se p^.pont^.pont = NULO então*

Pergunta 7 *eliminar o segundo registro da cadeia, devolvendo seu espaço para a memória, e anular o ponteiro do primeiro registro?*

Resposta 7 *libere(p^.pont)*
p^.pont ←NULO

Pergunta 8 *eliminar o PRIMEIRO registro da cadeia, devolvendo seu espaço para a memória, e manter a referência para o segundo registro?*

Resposta 8 *Neste caso, precisamos de um ponteiro auxiliar, pois não devemos liberar p e depois de liberar esse espaço utilizar ainda um campo de p (pont). Aí q é necessário como um ponteiro auxiliar no processo. O procedimento fica:*

$q \leftarrow p^{\wedge}.pont$
libere(p)
$p \leftarrow q$
$q \leftarrow NULO$

Desenhe o resultado desta última sequência de comandos.