Introdução à Computação II

BCC - EMA 8597

Prof. Rafael Oliveira[1]

[1]rpaes@icmc.usp.br

Universidade Estadual Paulista “Júlio de Mesquita Filho”
UNESP
Rio Claro 2012

Alocação de Memória

- Em um programa, 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.
- 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

Alocação de Memória

- 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 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 ora contraindo dependendo das condições do sistema e dos seus dados.

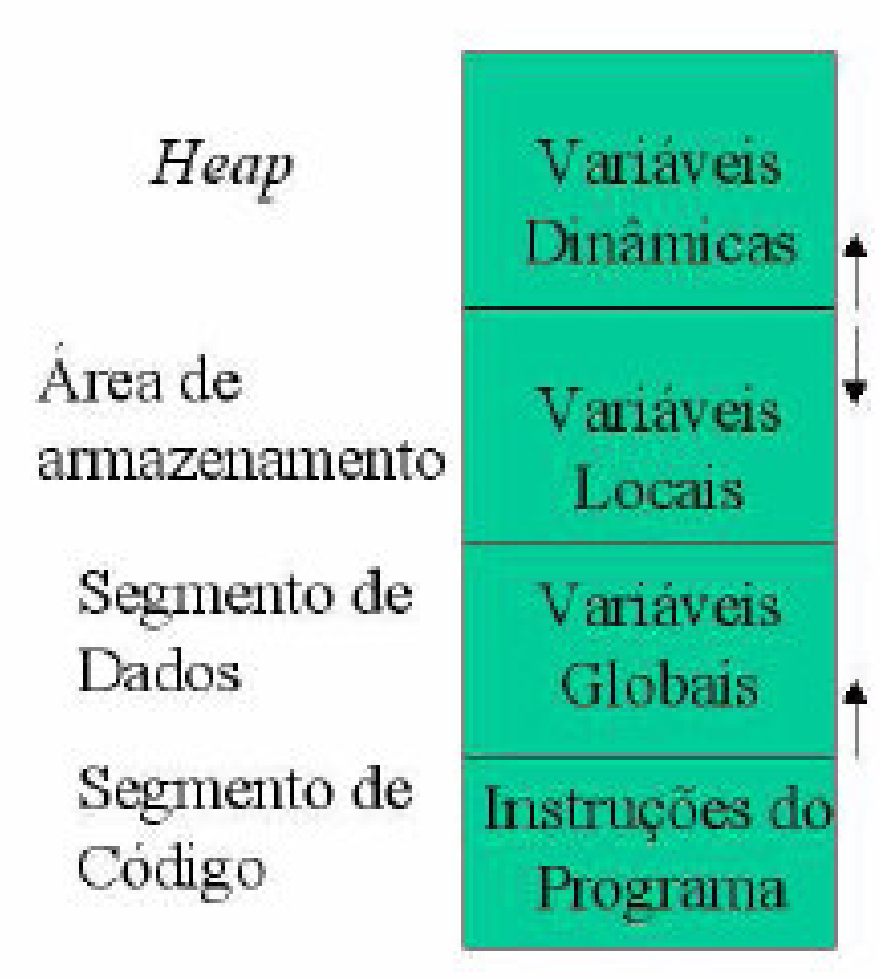
Alocação de Memória

- Para apoiar principalmente essas estruturas de tamanho variável, normalmente as linguagens de programação admitem acesso a uma região de memória adicional àquela inicialmente reservada para o programa, chamada de **Heap**.
- O tamanho da memória alocada na heap pode crescer quase que indefinidamente, em princípio expandindo até ocupar toda a memória disponível do computador.

Alocação de Memória

- 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.
- 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.

Alocação de Memória



Alocação de Memória

- Alocar memória dinamicamente significa gerenciar memória (isto é, reservar, utilizar e liberar espaço) durante o tempo de execução. Isso significa que o programador é responsável por controlar a reserva, ocupação e liberação de memória, de forma a ajustar a memória alocada às necessidades de um programa em cada etapa da sua execução.

Ponteiros

- 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 pseudocódigo utiliza o caracter $>$, ou $->$ para indicar que aquele é um ponteiro.
- Nas próximas transparências veremos exemplos de código em Pascal

Ponteiros

- 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 daquele apontador.

Ponteiros

Declaração

- Na linguagem algorítmica há o tipo de dados ponteiro.
- A declaração de um tipo ponteiro se faz com a seguinte sintaxe:

```
Tipo
 nome_do_tipo = >tipo de dado;
ou
Variável
 nome_da_variável : >tipo de dado;
```

Ponteiros

Exemplos

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

Ponteiros

Exemplos

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

Ponteiros

Exemplos

- No exemplo a variável identificada como apontAluno 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 apontValor está preparada para manipular alocação dinâmica e acesso a valores reais e apontÍndice a valores inteiros.

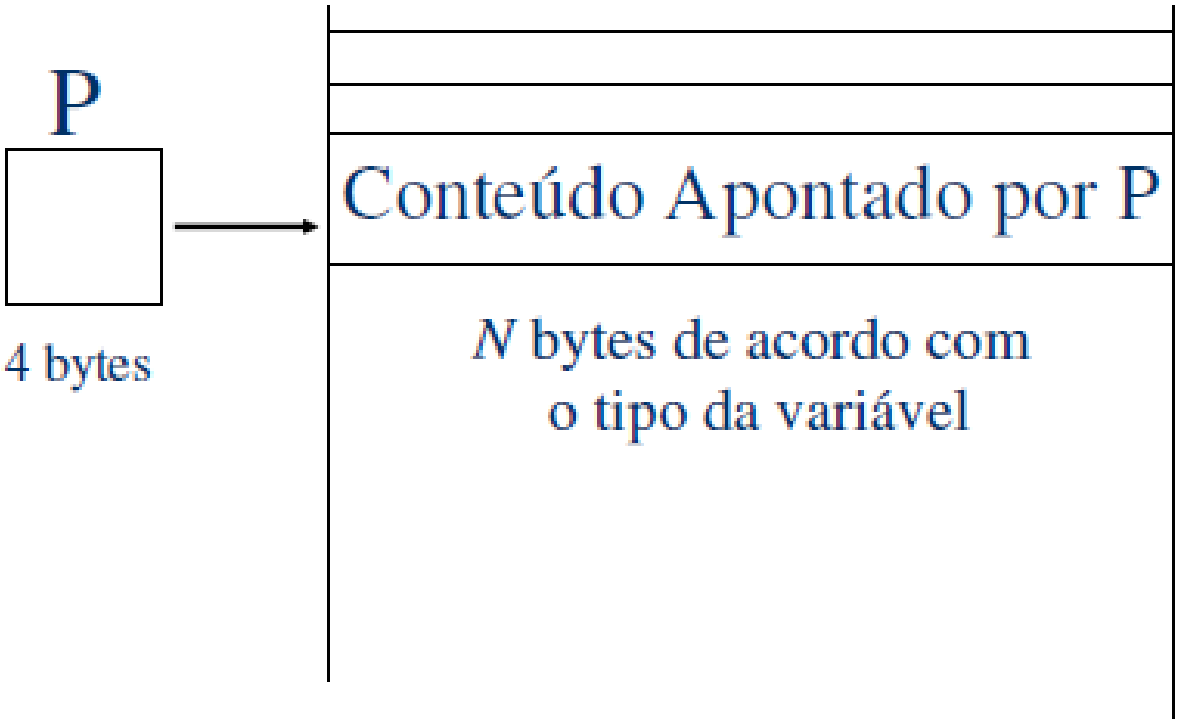
Ponteiros

Exemplos

- As variáveis do tipo ponteiro são armazenadas no segmento de dados junto com outras variáveis estáticas do programa.
- Como um ponteiro armazena apenas um endereço de memória, o seu tamanho em bytes é o tamanho necessário para armazenar tal endereço: em geral são usados 4 bytes (o tamanho de um endereço de memória no computador).
- Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está “apontando” para essa variável.

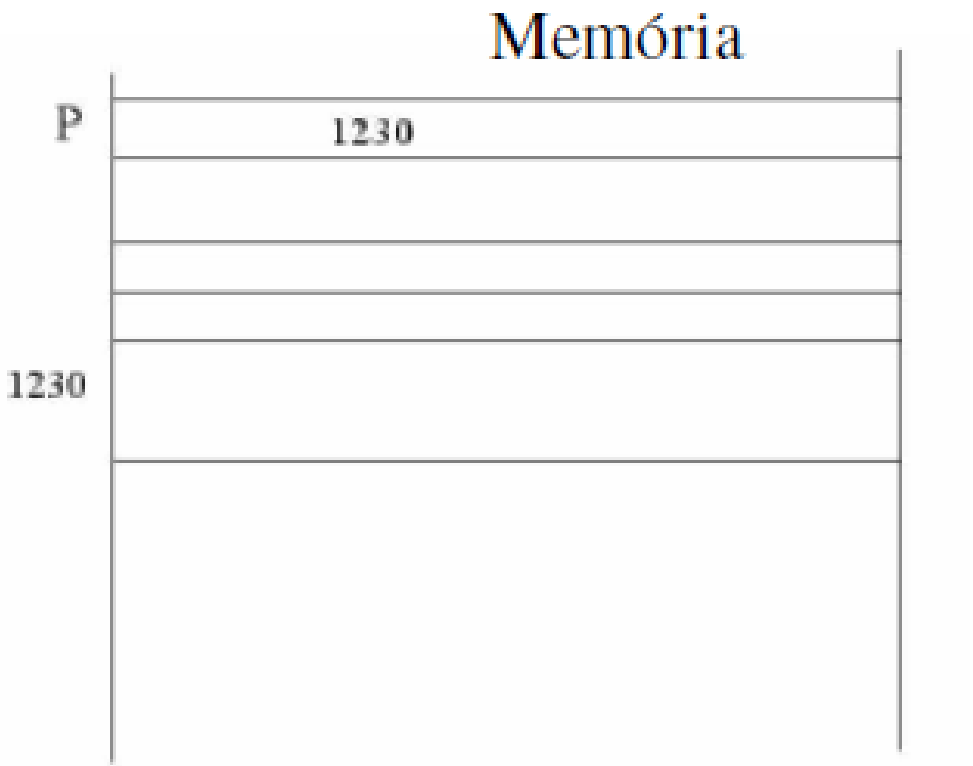
Ponteiros

Representação de uso de apontador



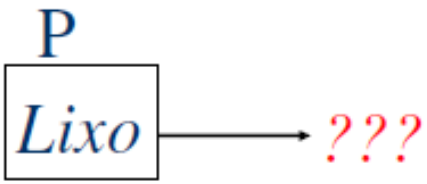
Ponteiros

Efeito em Memória



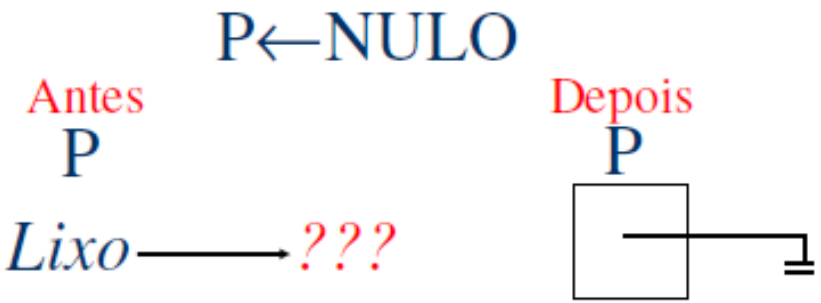
Ponteiros

- Ao ser declarado, a reserva de memória para o ponteiro é feita e o espaço alocado pode conter algum “lixo”, que aponta para um endereço qualquer de memória.



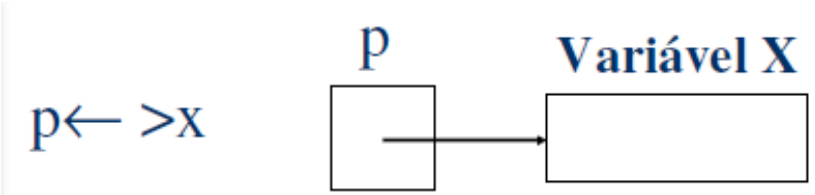
Ponteiros

- Para evitar tal situação (apontamento para “Lixo”), é necessário fazer com que os ponteiros apontem para o vazio. Essa é a forma de inicializar ponteiros.



Ponteiros

- Um ponteiro pode “apontar” para uma variável existente, isto é, armazenar o seu endereço. Para atribuir um valor para um ponteiro, é necessário indicar que o valor atribuído é um endereço e não o valor mantido pela variável.
- Para isso, utiliza-se o operador $>$ (ou $-$ $>$), que obtém o endereço de uma variável.

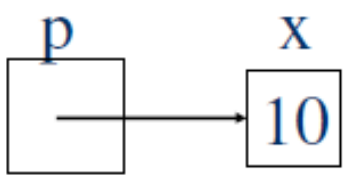


Ponteiros

- Para armazenar um valor no endereço de memória mantido por um ponteiro, é necessário indicar que o valor é armazenado no endereço apontado e não no próprio ponteiro.
- Para isso, utiliza-se o operador $\wedge$, que indica o conteúdo de um endereço

$p^{\wedge} \leftarrow 10$

(lê-se conteúdo do endereço apontado por p recebe 10)

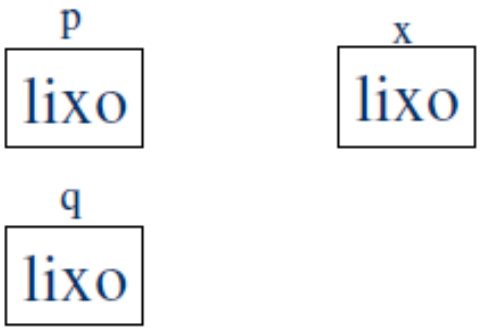


Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



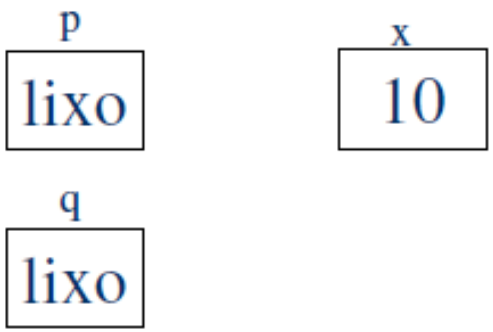

```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva(p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



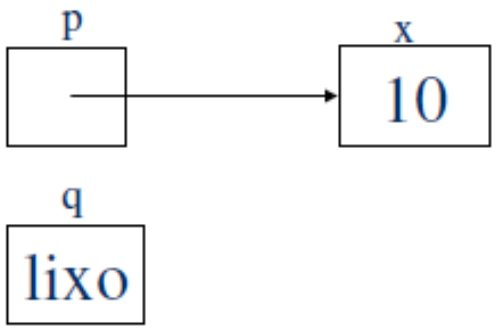

```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva(p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



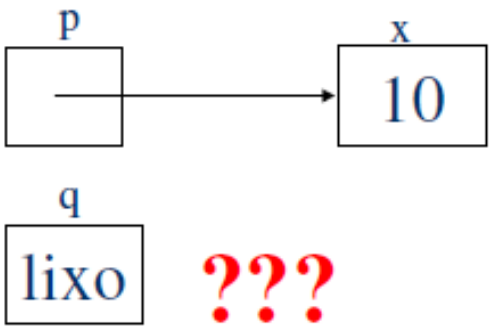

```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva(p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



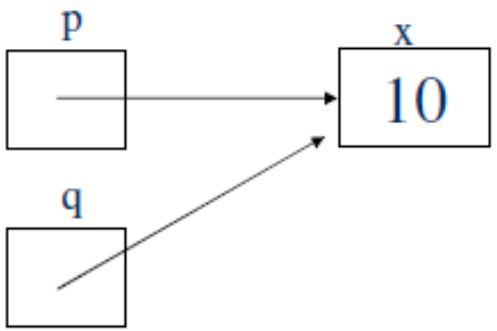

```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva(p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



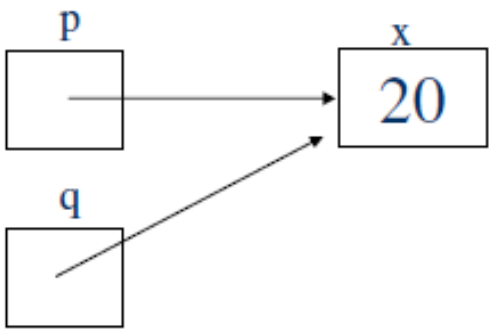

```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva(p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



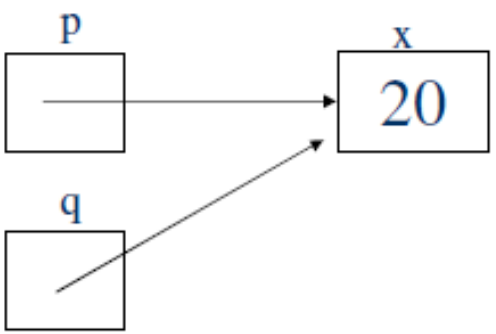

```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva(p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo

```
variável
x:inteiro
p,q:>inteiro
```



Resultado = 20

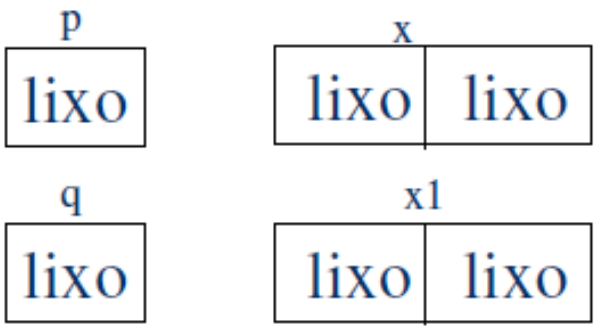
```
x ← 10
p ← >x
q^ ← 30 { ERRO: q aponta p/ lixo }
q ← p
q^ ← p^ + 10
escreva("Resultado = ",p^)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo 2

```
tipo
rec = registro
          dado: real
          pont: >rec
      fim registro
variável
x,x1:rec
p,q :>rec
```

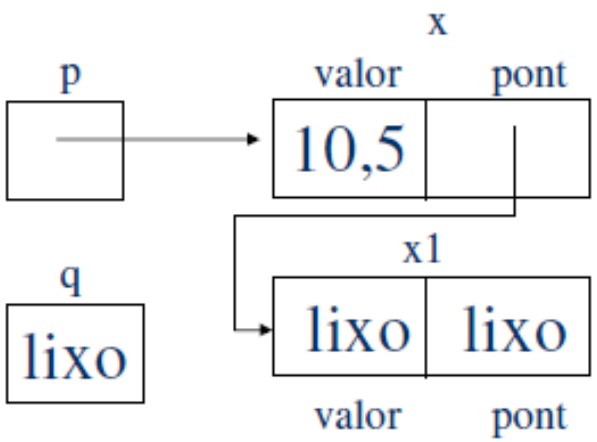


Ponteiros

Ponteiros (PseudoCodigo)

Exemplo 2

```
variável          x.valor ← 10,5
x,x1:rec          p ← >x
p,q:>rec          x.pont ← >x1
```

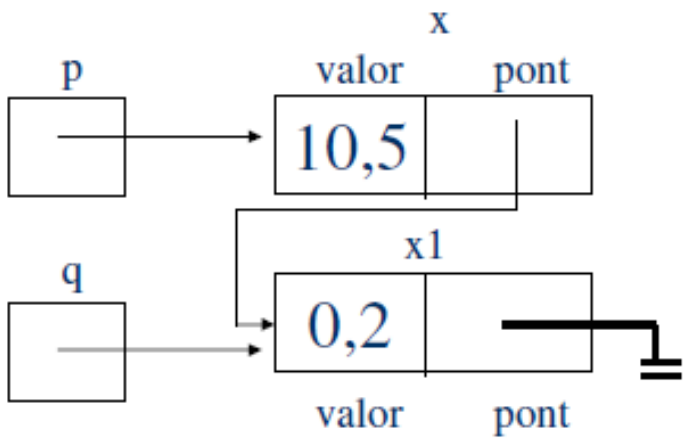


Ponteiros

Ponteiros (PseudoCodigo)

Exemplo 2

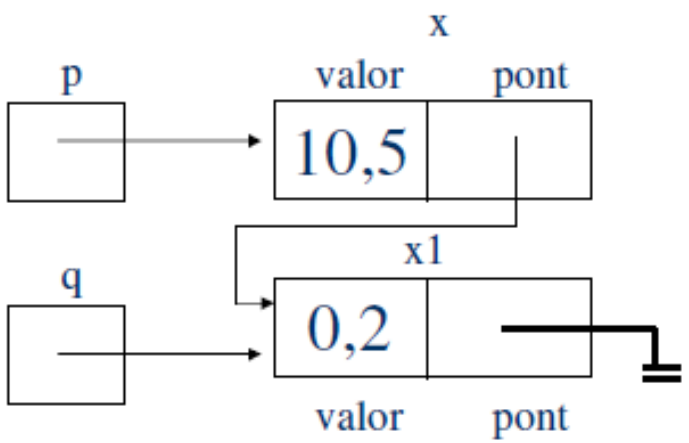
```
variável
x,x1:rec
p,q:>rec


x.valor ← 10,5
p ← >x
x.pont ← >x1
x1.valor ← 0,2
q ← x.pont
x1.pont ← NULO
```



Ponteiros

Ponteiros (PseudoCodigo)

Exemplo 2

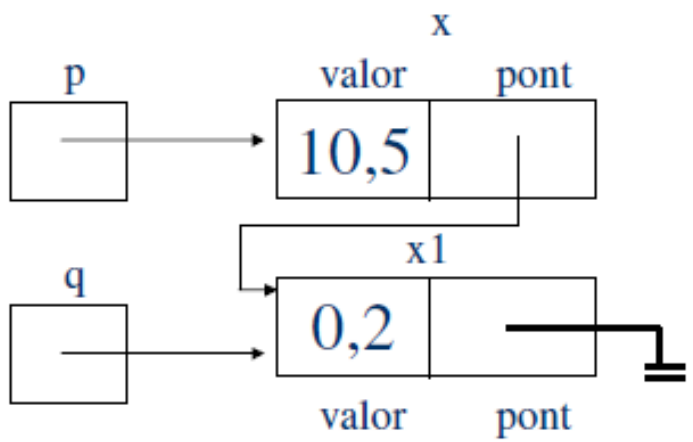



```
escreva(p^.valor)
escreva(q^.valor)
escreva(p^.pont^.valor)
escreva(x.valor)
escreva(x.pont^.valor)
escreva(x1.valor)
```

Ponteiros

Ponteiros (PseudoCodigo)

Exemplo 2




```
escreva(p^.valor)          10,5
escreva(q^.valor)          0,2
escreva(p^.pont^.valor)    0,2
escreva(x.valor)           10,5
escreva(x.pont^.valor)     0,2
escreva(x1.valor)          0,2
```

Ponteiros

- Apontadores são utilizados para apoiar a reserva de espaço em memória heap, bem como sua liberação após o uso
- Do ponto de vista de programação, alocações são feitas com base no tipo do ponteiro, ou melhor, no tamanho que aquele tipo ocupa

Ponteiros

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

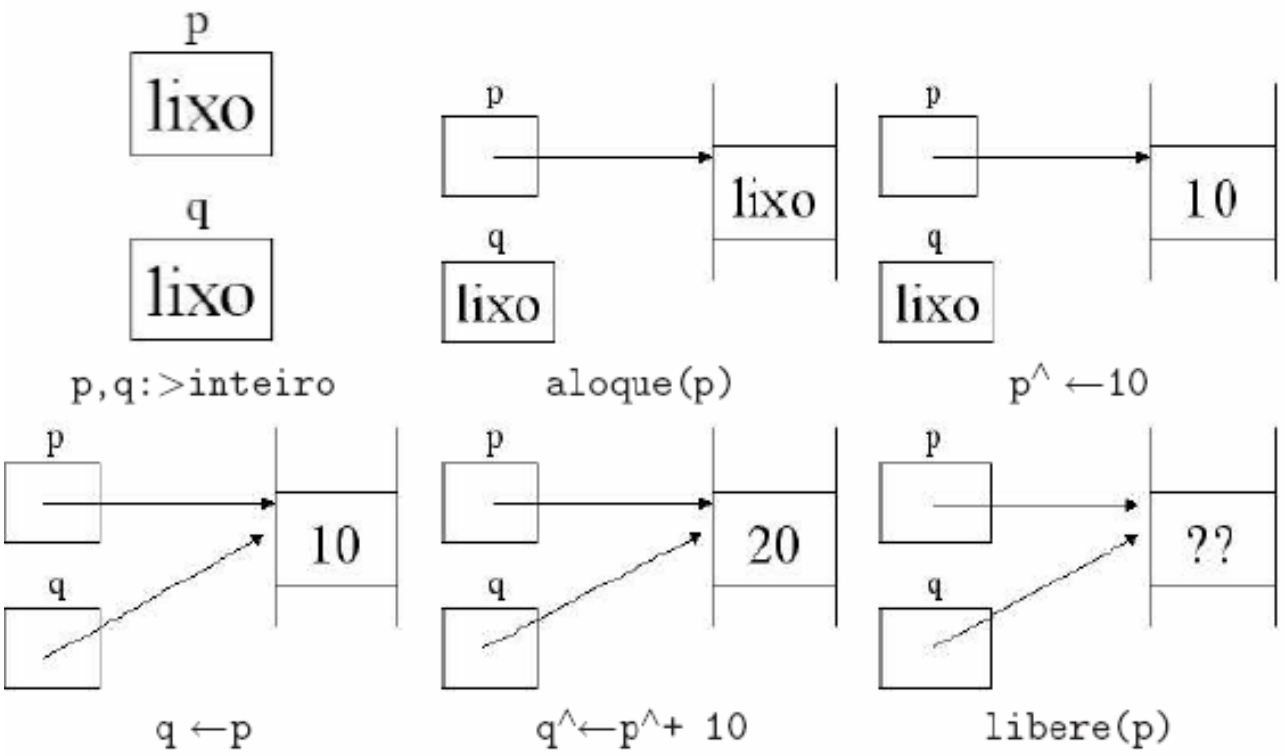
- Quando uma alocação for feita com base em p1, será alocado espaço necessário para conter uma variável do tipo rec
- Quando uma alocação for feita com base em p3, será alocado espaço necessário para conter um número real

Ponteiros

- 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 operação para liberação de memória em pseudocódigo é denotada por: **libere(p)**

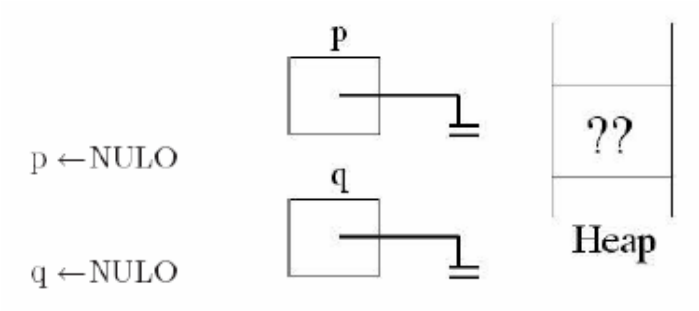
Ponteiros

Exemplo



Ponteiros

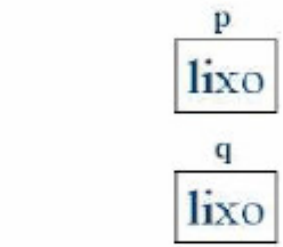
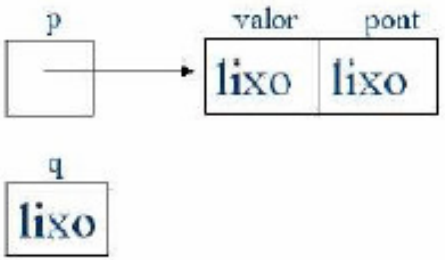
Exemplo



- 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

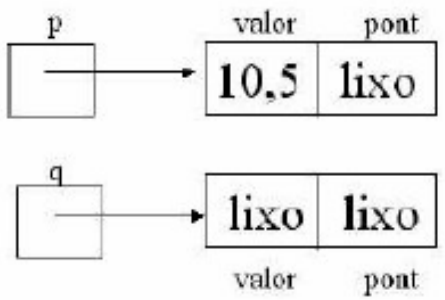
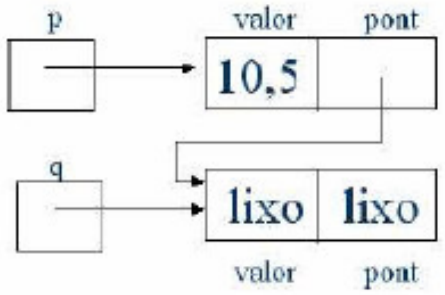
Ponteiros

Exemplo

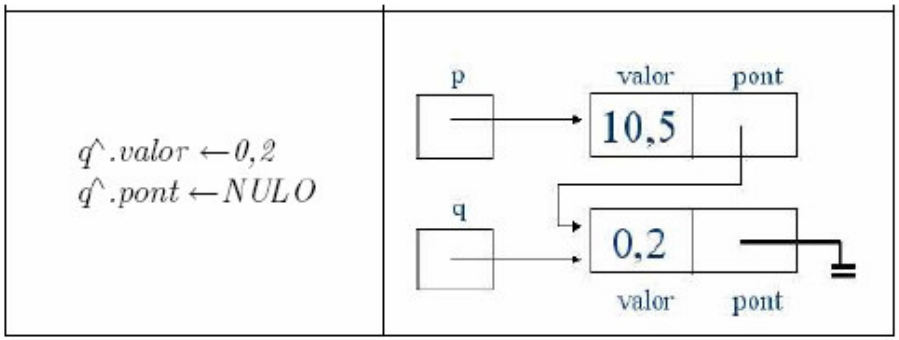
tipo rec = registro *dado: real* *pont: >rec* *fim registro* *variável p,q :>rec*	
aloque(p)	

Ponteiros

Exemplo

$p\hat{}.valor \leftarrow 10,5$ $aloque(q)$	
$p\hat{}.pont \leftarrow q$	

Ponteiros




```
                                Resposta
escreva(p^.valor)                10,5
escreva(q^.valor)                0,2
escreva(p^.pont^.valor)          0,2
```

Ponteiros

- Diversos conceitos podem ser associados à utilização de ponteiros.
- Antes de vermos a sintaxe do uso de ponteiros em linguagem Pascal, veremos a utilização de ponteiros para a definição de listas encadeadas

Lista Encadeada

- A alocação dinâmica é um instrumento que permite o aumento e diminuição do número de elementos de uma composição sem reserva prévia
- A Lista Encadeada é uma implementação de um tipo de dados mais genérico, chamado Lista
- Uma lista encadeada pode ser vista como uma coleção de elementos de mesmo tipo

Lista Encadeada

- A lista com a qual estamos lidando possui as seguintes características:
  - Os elementos da lista estão armazenados de forma ordenada (ordem alfabética)
  - O primeiro elemento da lista é indicado pelo valor de um ponteiro.
  - Cada elemento da lista possui um apontador para o próximo elemento da lista (prox)
  - O apontador que pertence ao último nó da lista aponta para NULO
  - A lista não admite repetições (ou seja, não pode possuir dois nomes exatamente iguais).

Lista Encadeada

- Com base nas premissas apresentadas, pode-se estabelecer um conjunto de operações a serem realizadas com a lista:
  - Inserção de um elemento
  - Eliminação de um elemento
  - Busca de um elemento
  - Recuperação do primeiro elemento
  - Recuperação do último elemento
  - Verificação de lista vazia
  - Verificação do número de elementos na lista
  - etc ...

Lista Encadeada

Modelo

Lista Encadeada - Implementação

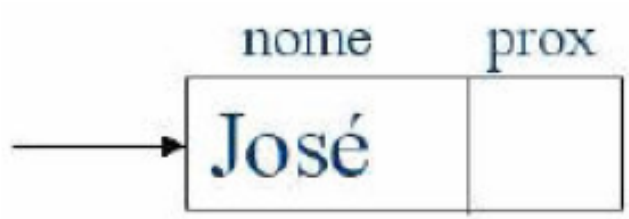
```
tipo nome = cadeia[20]
     apont_no = >no
     no = registro
      dado:nome
      prox:apont_no
    fim registro

  lista = apont_no
    Subprograma vazia(L):lógico
    e:L:lista
{ponteiro para o primeiro elemento da lista}
    início
    retorne(L=NULO)
    fim
```

Lista Encadeada - Implementação

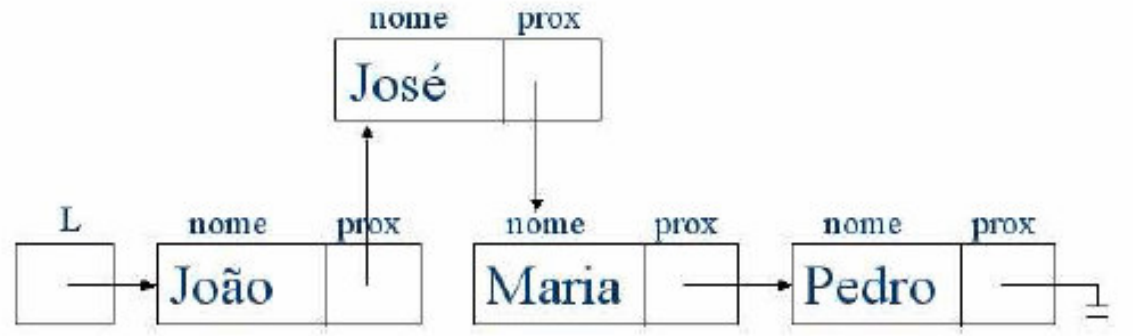
```
Subprograma tamanho(L):inteiro
  e:L:lista
{ponteiro para o primeiro elemento da lista}
  r: número de elementos da lista.
  variável cont:inteiro
  pont:apont_no {apontador auxiliar}
  início
  cont <- 0
  pont <- L
enquanto pont 6= NULO faça
  cont <- cont + 1
  pont <- pont^.prox
fim enquanto retorne(cont)
fim
```

Lista Encadeada

Inserção



Alocação e armazenamento de um novo elemento



Lista Encadeada - Inserção

- O trecho de algoritmo abaixo realiza a busca da posição do elemento a ser inserido com uso do ponteiro atrasado

```
...
aux ← NULO
pont ← L
enquanto pont^.dado < novo_nome e pont^.prox ≠ NULO faça
  aux ← pont
  pont ← pont^.prox
fim enquanto
...
```

Lista Encadeada - Inserção

- Trecho da inserção, que deve ser executado após a busca

```
...
aloque(novo_p)
novo_p^.dado ← novo_nome
novo_p^.prox ← aux^.prox
aux^.prox ← novo_p
...
```

Lista Encadeada

- Note que o trecho de algoritmo anterior funciona quando o elemento novo é do meio da lista e também quando ele é o último da lista
- No entanto, quando o elemento novo é o primeiro da lista, na busca o apontador aux retornou NULO, e portanto não pode ser usado da forma indicada no trecho de algoritmo acima
- Adicionalmente, quando o elemento já existe ele não pode ser inserido. Assim, falta prever esses dois casos

Lista Encadeada

Inserção

```
...
Se pont^.dado ≠ novo_nome então
  aloque(novo_p)
  novo_p^.dado ← novo_nome
  Se aux ≠ NULO então
    novo_p^.prox ← aux^.prox
    aux^.prox ← novo_p
  senão
    novo_p^.prox ← L
    L ← novo_p
  fim se
fim se
...
```

Lista Encadeada - Inserção Completa

```
Subprograma insere(L,novo_nome)

e: novo_nome : nome
```

Lista Encadeada - Inserção Completa

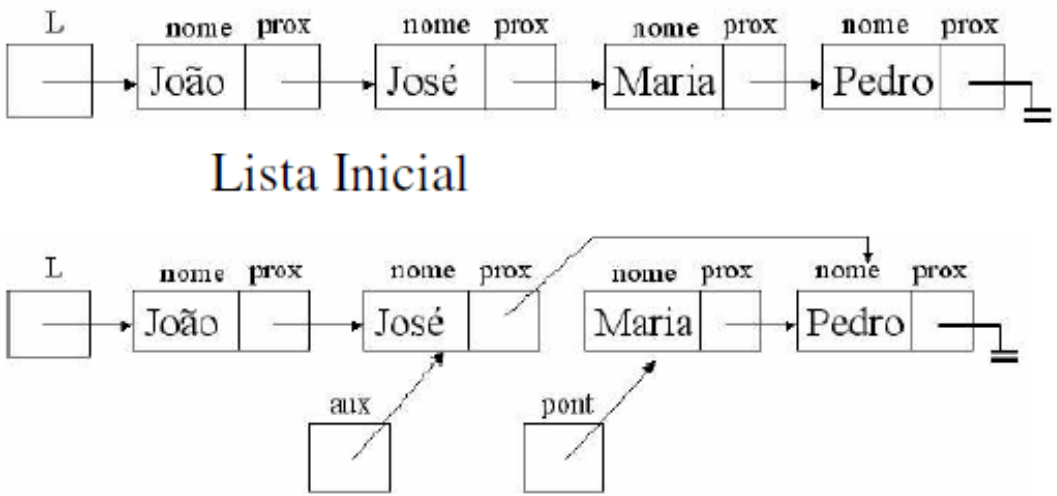
```
e/s: L:lista {a lista sai alterada, o próprio ponteiro da lista
pode sair também}

variável
  aux, pont, novo_p : apont_elemento


início

  Se vazia(L) então {Tratamento de lista vazia}
     aloque(novo_p)
     novo_p^.dado ← novo_nome
     novo_p^.prox ← NULO
     L ← novo_p
  senão
    aux ← NULO
    pont ← L
    enquanto pont^.dado < novo_nome e pont^.prox ≠ NULO
    faça
       aux ← pont
       pont ← pont^.prox
    fim enquanto
```

Lista Encadeada

Inserção Completa

```
    Se pont^.dado ≠ novo_nome então
       aloque(novo_p)
       novo_p^.dado ← novo_nome
       Se aux ≠ NULO então
          novo_p^.prox ← aux^.prox
          aux^.prox ← novo_p
       senão
          novo_p^.prox ← L
          L ← novo_p
       fim se
    fim se
  fim se
fim
```

Lista Encadeada - Após Eliminação



Lista Encadeada - Após Eliminação

```
Subprograma Elimina(L,nome_el)

e: nome_el : nome

e/s: L:lista {a lista sai alterada, o próprio ponteiro da lista
pode sair também}

variável
  aux, pont, novo_p : apont_elemento


início

  Se não vazia(L) então {Teste de lista vazia}
    {Encontre o elemento a ser eliminado }
    aux ← NULO
    pont ← L
    enquanto pont^.dado < novo_nome e pont^.prox ≠ NULO
    faça
       aux ← pont
       pont ← pont^.prox
    fim enquanto
```

Lista Encadeada - Após Eliminação

```
    {Elimine o elemento }
    Se pont^.dado = novo_nome então {o elemento existe}
       Se aux ≠ NULO então {o elemento não é o primeiro}
          aux^.prox ← pont^.prox
       senão
          L ← pont^.prox
       fim se
       {devolva o espaço ocupado pelo elemento à heap}
       libera(pont)
    fim se
  fim se
fim
```

Ponteiros

FIM