Ponteiros

Aula 06

Ivone P. Matsuno Yugoshi
ivone.matsuno@ufms.br

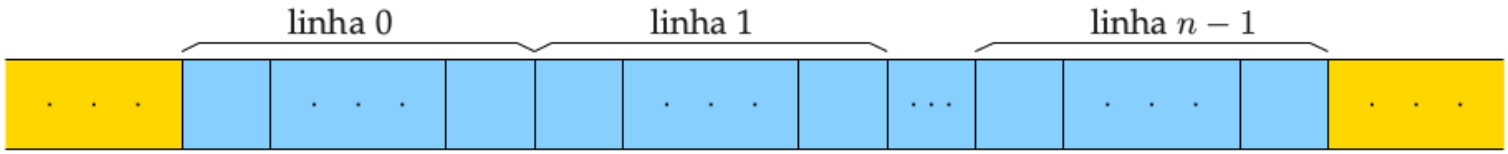
Ronaldo Fiorilo dos Santos
ronaldo.santos@ufms.br

Universidade Federal de Mato Grosso do Sul
Câmpus de Três Lagoas
Bacharelado em Sistemas de Informação

Algoritmos e Programação II

Ponteiros para elementos de uma matriz

- Em C as matrizes são armazenadas como uma sequência contínua de compartimentos de memória, com demarcações de onde começa e onde termina cada uma de suas linhas
  - Ou seja, uma matriz é armazenada como um vetor na memória, com marcas especiais em determinados pontos regularmente espaçados: os elementos da linha 0 vêm primeiro, seguidos pelos elementos da linha 1, da linha 2, e assim por diante.



  - Essa representação pode nos ajudar a trabalhar com ponteiros e matrizes.
- Se um ponteiro *p* apontar para a primeira célula de uma matriz (posição [0,0]), podemos então visitar todos os elementos da matriz incrementando o ponteiro *p* repetidamente.

Ponteiros para elementos de uma matriz

- Por exemplo, suponha que queremos inicializar todos os elementos de uma matriz de números inteiros com 0. Suponha que temos a declaração da seguinte matriz:

```
int A[LINHAS][COLUNAS];
```

- A forma que aprendemos inicializar uma matriz pode ser aplicada à matriz *A* declarada acima e então o seguinte trecho de código realiza a inicialização desejada:

```
int i, j;
.
.
.
for (i = 0; i < LINHAS; i++)
   for (j = 0; j < COLUNAS; j++)
      A[i][j] = 0;
```

Ponteiros para elementos de uma matriz

- Se vemos a matriz *A* da forma como é armazenada na memória, isto é, como um vetor unidimensional, podemos trocar o par de estruturas de repetição por uma única estrutura de repetição:

```
int *p;
.
.
.
for (p = &A[0][0]; p <= &A[LINHAS-1][COLUNAS-1]; p++)
   *p = 0;
```

Processamento das linhas de uma matriz

- Podemos processar uma linha de uma matriz – percorrê-la, usar seus valores, modificá-los, etc – também usando ponteiros.
- Por exemplo, para visitar os elementos da linha *i* de uma matriz *A* podemos usar um ponteiro *p* apontar para o elemento da linha *i* e da coluna 0 da matriz *A*, como mostrado abaixo:

```
p = &A[i][0];
```

ou poderíamos fazer simplesmente

```
p = A[i];
```

já que a expressão `A[i]` é um ponteiro para o primeiro elemento da linha *i*.

Processamento das linhas de uma matriz

- A justificativa para isso vem da aritmética com ponteiros
- Lembre-se que para um vetor A:
  - $A[i]$ é equivalente a $*(A + i)$
  - $\&A[i][0]$ é o mesmo que $\&(*(A[i] + 0))$
    - que é equivalente a $\&*A[i]$
    - e que, por fim, é equivalente a $A[i]$.
- No trecho de código a seguir usamos essa simplificação para inicializar com zeros a linha i da matriz A:

```
int A[LINHAS][COLUNAS], *p, i;
.
.
.
for (p = A[i]; p < A[i] + COLUNAS; p++)
   *p = 0;
```

Processamento das linhas de uma matriz

- Como $A[i]$ é um ponteiro para a linha i da matriz A, podemos passar $A[i]$ para uma função que espera receber um vetor como argumento. Em outras palavras, uma função que foi projetada para trabalhar com um vetor também pode trabalhar com uma linha de uma matriz.
- Dessa forma, a função `int max(int n, int v[MAX])` pode ser chamada com a linha i da matriz A como argumento:

```
M = max(COLUNAS, A[i]);
```

Processamento das colunas de uma matriz

- O processamento dos elementos de uma coluna de uma matriz não é tão simples como os de uma linha, já que a matriz é armazenada linha por linha na memória.
- A estrutura a seguir, inicializa com zeros a coluna *j* da matriz *A*:

```
int A[LINHAS][COLUNAS], (*p)[COLUNAS], j;
.
.
.
for (p = &A[0]; p < &A[LINHAS]; p++)
   (*p)[j] = 0;
```

  - Declaramos *p* como um ponteiro para um vetor de inteiros com dimensão **COLUNAS**
  - A expressão *p*++ avança *p* para a próxima linha
  - Em $(*p)[j]$, $*p$ representa uma linha inteira de *A* e assim $(*p)[j]$ seleciona o elemento na coluna *j* da linha

Identificadores de matrizes como ponteiros

- O identificador de uma matriz também pode ser usado como um ponteiro, assim como nos vetores
  - No entanto, alguns cuidados são necessários quando ultrapassamos a barreira de duas ou mais dimensões
- Considere a declaração da matriz a seguir:

```
int A[LINHAS][COLUNAS];
```

- Neste caso, *A não* é um ponteiro para `A[0][0]`
  - Ao contrário, é um ponteiro para `A[0]`
- Isso faz mais sentido se olharmos sob o ponto de vista da linguagem C, que considera *A* não como uma matriz bidimensional, mas como um vetor
  - Quando usado como um ponteiro, *A* tem tipo `int (*)[COLUNAS]`, um ponteiro para um vetor de números inteiros de tamanho `COLUNAS`

Identificadores de matrizes como ponteiros

- Saber que *A* aponta para `A[0]` é útil para simplificar estruturas de repetição que processam elementos de uma matriz.
- Por exemplo, ao invés de escrever:

```
for (p = &A[0]; p < &A[LINHAS]; p++)
   (*p)[j] = 0;
```

para inicializar a coluna *j* da matriz *A*, podemos escrever

```
for (p = A; p < A + LINHAS; p++)
   (*p)[j] = 0;
```

Identificadores de matrizes como ponteiros

- Com essa idéia, podemos fazer o compilador acreditar que uma variável composta homogênea multi-dimensional é unidimensional, isto é, é um vetor
- Por exemplo, podemos passar a matriz A como argumento para a função `int max(int n, int v[MAX])` da seguinte forma:

```
M = max(LINHAS*COLUNAS, A[0]);
```

já que $A[0]$ aponta para o elemento na linha 0 e coluna 0 e tem tipo $int *$, assim essa chamada será executada corretamente.

Ponteiros para registros

- Suponha que definimos o registro **`data`** como a seguir:

```
struct data {
  int dia;
  int mes;
  int ano;
};
```

- Assim, podemos declarar variáveis do tipo **`struct data`**:

```
struct data hoje;
```

Ponteiros para registros

- E então, assim como fizemos com ponteiros para inteiros, caracteres e números de ponto flutuante, podemos declarar um ponteiro para o registro data da seguinte forma:

```
struct data * p;
```

- Podemos, a partir dessa declaração, fazer uma atribuição à variável *p* como a seguir:

```
p = &hoje;
```

Ponteiros para registros

- Além disso, podemos atribuir valores aos campos do registro de forma indireta:

```
(* p).dia = 11;
```

- Essa atribuição armazena o número inteiro 11 no campo `dia` da variável `hoje`, indiretamente através do ponteiro *p*
  - os parênteses envolvendo `*p` são necessários porque o operador `.`, de seleção de campo de um registro, tem maior prioridade que o operador `*` de indireção.
- A forma de acesso indireto aos campos de um registro pode ser substituída pelo operador `->` como no exemplo abaixo:

```
p->dia = 11;
```

Ponteiros para registros

```
#include <stdio.h>
struct data {
   int dia;
   int mes;
   int ano;
};

int main(void)
{
   struct data hoje, * p;
   p = &hoje;
   p->dia = 13;
   p->mes = 10;
   p->ano = 2010;
   printf("A data de hoje é %d/%d/%d\n", hoje.dia, hoje.mes, hoje.ano);    return 0;
}
```

Registros contendo ponteiros

- Podemos também usar ponteiros como campos de registros

```
struct reg_pts {
   int *pt1;
   int *pt2;
};
```

- A partir dessa definição, podemos declarar variáveis (registros) do tipo `struct reg_pts` como a seguir:

```
struct reg_pts bloco;
```

- Em seguida, a variável `bloco` pode ser usada como sempre fizemos

Registros contendo ponteiros

```
#include <stdio.h>
struct pts_int {
   int *pt1;
   int *pt2;
};

int main(void)
{
   int i1, i2;
   struct pts_int reg;
   i2 = 100;
   reg.pt1 = &i1;
   reg.pt2 = &i2;
   *reg.pt1 = -2;
   printf("i1 = %d, *reg.pt1 = %d\n", i1, *reg.pt1);
   printf("i2 = %d, *reg.pt2 = %d\n", i2, *reg.pt2);
   return 0;
}
```

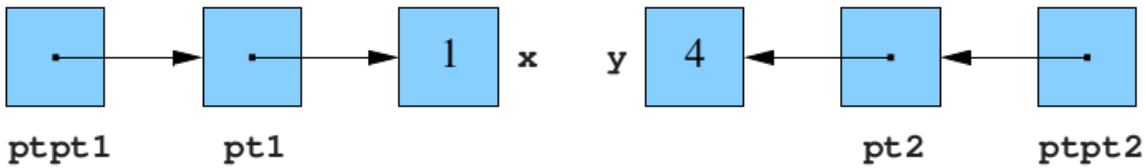
Ponteiros para ponteiros

- Uma variável que é um ponteiro para uma posição de memória que contém um outro ponteiro é chamada de **ponteiro para um ponteiro**
- Podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla
- Ponteiros para ponteiros têm diversas aplicações na linguagem C, especialmente no uso de matrizes

Ponteiros para ponteiros

```
#include <stdio.h>

int main(void)
{
   int x, y, *pt1, *pt2, **ptpt1, **ptpt2;

   x = 1;
   y = 4;
   printf("x=%d y=%d\n", x, y);

   pt1 = &x;
   pt2 = &y;
   printf("*pt1=%d *pt2=%d\n", *pt1, *pt2);

   ptpt1 = &pt1;
   ptpt2 = &pt2;
   printf("**ptpt1=%d **ptpt2=%d\n", **ptpt1, **ptpt2);

   return 0;
}
```

Ponteiros para ponteiros



Alocação dinâmica de memória

- Aprendemos que para declarar vetores e matrizes precisamos informar as dimensões dessas variáveis.
  - Muitas vezes o total de espaço alocado na memória não é usado durante a execução do programa
- **Alocação estática de memória**: antes da execução, o compilador reserva na memória um número fixo de compartimentos correspondentes à declaração (espaço é fixo e não pode ser alterado durante a execução do programa)

Alocação dinâmica de memória

- A linguagem C possui a habilidade de reservar espaços na memória principal durante a execução de um programa
- Podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa
- **Alocação dinâmica de memória**: significa que um programa solicita ao sistema computacional, durante a sua execução, blocos da memória principal que estejam disponíveis para uso

Alocação dinâmica de memória

- Uma forma de realizar a alocação dinâmica de memória é através da função **`malloc`**, que está declarada no arquivo **`stdlib.h`**

```
void *malloc(size_t tamanho)
```

- Caso haja espaço suficiente na memória, a função **`malloc`** reserva um bloco contínuo de `tamanho` bytes e devolve um ponteiro do tipo **`void`** para a primeira posição do bloco
- Caso não haja possibilidade de atender a solicitação de espaço, um ponteiro apontando para **`NULL`** é devolvido

Alocação dinâmica de memória

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
   int i, n, *vetor, *pt;

   scanf("%d", &n);
   vetor = (int *) malloc(n * sizeof(int));
   if (vetor != NULL) {
      for (i = 0; i < n; i++)
         scanf("%d", (vetor + i));
      for (pt = vetor; pt < (vetor + n); pt++)
         printf("%d ", *pt);
      printf("\n");
      for (i = n - 1; i >= 0; i--)
         printf("%d ", vetor[i]);
      printf("\n");
   }
   else
      printf("Impossível alocar espaço\n");
   return 0;
}
```

Alocação dinâmica de memória

- No programa anterior, reservamos *n* compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`
- O operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão
- Esse número é multiplicado por *n*, o número de compartimentos que desejamos para armazenar números inteiros
- A função `malloc` devolve um ponteiro do tipo `void` e, por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro

Exercícios

1. Reescreva a função abaixo usando aritmética de ponteiros em vez de índices de matrizes. Em outras palavras, elimine as variáveis *i* e *j* e todos os `[]`. Use também uma única estrutura de repetição.

```
int soma_matriz(int n, int A[DIM][DIM])
{
   int i, j, soma = 0;

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         soma = soma + A[i][j];

   return soma;
}
```

Exercícios

2. Qual a saída do programa descrito abaixo?

```
#include <stdio.h>
struct dois_valores {
   int vi;
   float vf;
};

int main(void)
{
   struct dois_valores reg1 = {53, 7.112}, reg2, * p = &reg1;
   reg2.vi = (* p).vf;
   reg2.vf = (* p).vi;
   printf("1: %d %f\n2: %d %f\n", reg1.vi, reg1.vf, reg2.vi, reg2.vf);
   return 0;
}
```

Exercícios

3. Simule a execução do programa descrito abaixo.

```
#include <stdio.h>
struct pts {
   char *c;
   int *i;
   float *f;
};

int main(void)
{
   char caractere;
   int inteiro;
   float real;
   struct pts reg;
   reg.c = &caractere;
   reg.i = &inteiro;
   reg.f = &real;
   scanf("%c%d%f", reg.c, reg.i, reg.f);
   printf("%c\n%d\n%f\n", caractere, inteiro, real);
   return 0;
}
```

Exercícios

4. Simule a execução do programa descrito abaixo.

```
#include <stdio.h>
struct celula {
   int valor;
   struct celula *prox;
};

int main(void)
{
   struct celula reg1, reg2, * p;
   scanf("%d%d", &reg1.valor, &reg2.valor);
   reg1.prox = &reg2;
   reg2.prox = NULL;
   for (p = &reg1; p != NULL; p = p->prox)
      printf("%d ", p->valor);
   printf("\n");
   return 0;
}
```