COM111 — Alocação Dinâmica

Hokama

UNIFEI

16 de Setembro de 2021

Disclaimer

Esses slides foram adaptados dos slides elaborados pelo Prof. Eduardo C. Xavier para a disciplina MC102 da UNICAMP.

Exemplo de Ponteiros e Alocação Dinâmica

Vamos criar uma aplicação que cria um vetor dinâmico com funções para implementar as seguintes operações:

- Inclusão de um elemento no final do vetor.
- Exclusão da primeira ocorrência de um elemento no vetor.
- Impressão do vetor.

Exemplo de Ponteiros e Alocação Dinâmica

- O tamanho do vetor deve se ajustar automaticamente: se elementos são inseridos devemos “aumentar"o tamanho do vetor para inclusão de novos elementos, e se elementos forem removidos devemos "diminuir"o tamanho vetor.
- Temos duas variáveis associadas ao vetor:
  - **size**: denota quantos elementos estão armazenados no vetor.
  - **maxSize**: denota o tamanho alocado do vetor.

Exemplo de Ponteiros e Alocação Dinâmica

Temos as seguintes regras para ajuste do tamanho alocado do vetor:

- O vetor deve ter tamanho alocado no mínimo igual a 4.
- Se o vetor ficar cheio, então devemos alocar um novo vetor com o dobro do tamanho atual.
- Se o número de elementos armazenados no vetor for menor do que 1/4 do tamanho alocado do vetor, então devemos alocar um novo vetor com metade do tamanho atual.

Exemplo de Ponteiros e Alocação Dinâmica

Implementaremos as seguintes funções:

- `int * initVet(int *size, int *maxSize);`

  Aloca um vetor inicial de tamanho 4, setando **size** com valor 0, **maxSize** com valor 4, e devolve o endereço do vetor alocado.

- `void printVet(int *v, int size, int maxSize);`

  Imprime o conteúdo e tamanhos associados ao vetor **v**.

- `int * addVet(int *v, int *size, int *maxSize, int e);`

  Adiciona o elemento **e** no final do vetor **v**. Caso não haja espaço, um novo vetor com o dobro do tamanho deve ser alocado. A função sempre retorna o endereço do vetor, sendo um novo alocado ou não. Além disso os valores de **size** e **maxSize** devem ser atualizados.

Exemplo de Ponteiros e Alocação Dinâmica

Implementaremos as seguintes funções:

- `int find(int *v, int size, int e);`

  Determina se o elemento **e** está presente ou não no vetor **v**. Caso esteja presente, retorna a posição da primeira ocorrência de **e** em **v**. Caso não esteja presente, retorna -1.

- `int * removeVet(int *v, int *size, int *maxSize, int e);`

  Remove a primeira ocorrência do elemento **e** do vetor **v** caso este esteja presente. O valor de **size** deve ser decrementado de 1. Caso o número de elementos armazenados seja menor do que $\frac{1}{4}$maxSize, então um novo vetor de tamanho $\frac{1}{2}$maxSize deve ser alocado no lugar de **v**. A função sempre retorna o endereço inicial do vetor alocado, sendo um novo vetor alocado ou não.

Exemplo de Ponteiros e Alocação Dinâmica

Com as funções implementadas podemos executar o exemplo:

```
int main(){
  int *vet, size, maxSize;
  int i;

  vet = initVet(&size, &maxSize);

  for(i=0; i<20; i++){
    vet = addVet(vet, &size, &maxSize, i);
  }
  printVet(vet, size, maxSize);

  vet = removeVet(vet, &size, &maxSize, 14);
  printVet(vet, size, maxSize);

  for(i=5; i<15; i++){
    vet = removeVet(vet, &size, &maxSize, i);
  }
  printVet(vet, size, maxSize);

  for(i=0; i<20; i++){
    vet = removeVet(vet, &size, &maxSize, i);
  }
  printVet(vet, size, maxSize);

  free(vet);
}
```

Exercício

Implemente a função de remoção de um elemento do vetor de tal forma que se o número de elementos armazenados (**size**) for menor do que $\frac{1}{4}$**maxSize**, então o tamanho do vetor alocado deve ter tamanho igual a metade do anterior. A função deve devolver o endereço do início do vetor, sendo um novo alocado ou não. Além disso a função deve atualizar os valores **size** e **maxSize** caso necessário.

Informações Extras: Alocação Dinâmica de Matrizes

- Em aplicações científicas e de engenharias, é muito comum a realização de diversas operações sobre matrizes.
- Em situações reais o ideal é alocar memória suficiente para conter os dados a serem tratados. Não usar nem mais e nem menos!
- Como alocar vetores-multidimensionais dinamicamente?

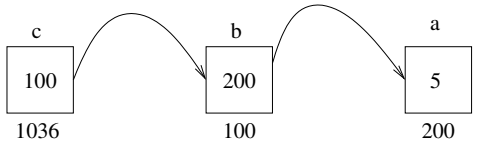
Informações Extras: Ponteiros para ponteiros

- Uma variável ponteiro está alocada na memória do computador como qualquer outra variável.
- Portanto podemos criar um ponteiro que contém o endereço de memória de um outro ponteiro.
- Para criar um ponteiro para ponteiro: **tipo **nomePonteiro;**

```c
int main(){
  int a=5, *b, **c;
  b = &a;
  c = &b;
  printf("%d\n", a);
  printf("%d\n", *b);
  printf("%d\n", *(*c));
}
```

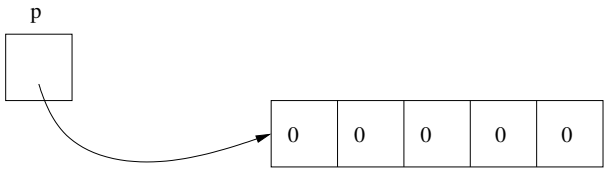

Informações Extras: Ponteiros para ponteiros

O programa imprime 5 três vezes, monstrando as três formas de acesso à variável **a**: **a, *b, **c**.



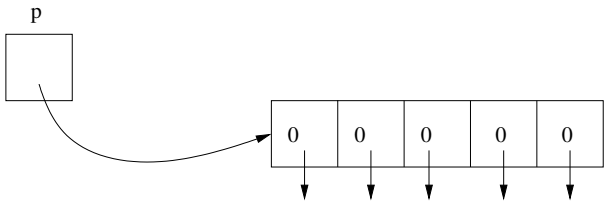
Informações Extras: Ponteiros para ponteiros

- Pela nossa discussão anterior sobre ponteiros, sabemos que um ponteiro pode ser usado para referenciar um vetor alocado dinamicamente.
  - ```
    int *p;
    p = calloc(5, sizeof(int));
    ```



Informações Extras: Ponteiros para ponteiros

- A mesma coisa acontece com um ponteiro para ponteiro, só que neste caso o vetor alocado é de ponteiros.
  - ```
    int **p;
    p = calloc(5, sizeof(int *));
    ```

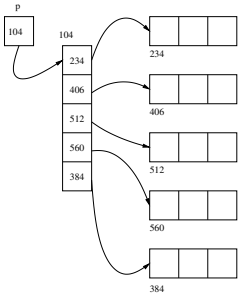


  - Note que cada posição do vetor acima é do tipo **int ***, ou seja, um ponteiro para inteiro!

Informações Extras: Ponteiros para ponteiros

- Como cada posição do vetor é um ponteiro para inteiro, podemos associar cada posição dinamicamente com um vetor de inteiros!
  - ```
    int **p;
    int i;
    p = calloc(5, sizeof(int *));

    for(i=0; i<5; i++){
      p[i] = calloc(3, sizeof(int));
    }
    ```



Informações Extras: Alocação Dinâmica de Matrizes

Esta é uma forma de se criar matrizes dinamicamente:

- Crie um ponteiro para ponteiro.
- Associe um vetor de ponteiros dinamicamente com este ponteiro de ponteiro. O tamanho deste vetor é o número de linhas da matriz.
- Cada posição do vetor será associada com um outro vetor do tipo a ser armazenado. Cada um destes vetores é uma linha da matriz (portanto possui tamanho igual ao número de colunas).

OBS: No final você deve desalocar toda a memória alocada!!
    ```

Informações Extras: Alocacão Dinâmica de Matrizes

```c
int main(){
  int **p, i, j;

  p = calloc(5, sizeof(int *));
  for(i=0; i<5; i++){
    p[i] = calloc(3, sizeof(int));
  } //Alocou matriz 5x3
  printf("Digite os valores da matriz\n");
  for(i = 0; i<5; i++)
    for(j=0; j<3; j++)
      scanf("%d", &p[i][j]);

  printf("Matriz lida\n");
  for(i = 0; i<5; i++){
    for(j=0; j<3; j++){
      printf("%d, ", p[i][j]);
    }
    printf("\n");
  }
  //desalocando memória usada
  for(i=0; i<5; i++)
    free(p[i]);
  free(p);
}
```


Informações Extras: Alocação Dinâmica de Matrizes

Mas a forma mais eficiente de criar matrizes é:

- Para uma matriz de dimensões $n \times m$, crie um vetor unidimensional dinamicamente deste tamanho.
- Use linearização de índices para trabalhar com o vetor como se fosse uma matriz.
- Desta forma tem-se um melhor aproveitamento da cache pois a matriz inteira está sequencialmente em memória.

No final você deve desalocar toda a memória alocada!!

Linearização de Índices

- Podemos usar sempre vetores simples para representar matrizes (na prática o compilador faz isto por você).
- Ao declarar uma matriz como **int mat[3][4]**, sabemos que serão alocados 12 posições de memória associadas com a variável **mat**.
- Poderíamos simplesmente criar **int mat[12]**. Mas perdemos a simplicidade de uso dos índices em forma de matriz.
  - Você não mais poderá escrever **mat[1][3]** por exemplo.

Linearização de Índices

- A *linearização de índices* é justamente a representação de matrizes usando-se um vetor simples.
- Mas devemos ter um padrão para acessar as posições deste vetor como se sua organização fosse na forma de matriz.

Linearização de Índices

- Considere o exemplo:
  int mat[12]; // ao invés de int mat[3][4]
- Fazemos a divisão por linhas como segue:
  - Primeira linha: **mat[0]** até **mat[3]**
  - Segunda linha: **mat[4]** até **mat[7]**
  - Terceira linha: **mat[8]** até **mat[11]**
- Para acessar uma posição $[i][j]$ usamos:
  - **mat[i*4 + j];**
    onde $0 \leq i \leq 2$ e $0 \leq j \leq 3$.

Linearização de Índices

- De forma geral, seja matriz **mat[n*m]**, representando **mat[n][m]**.
- Para acessar a posição correspondente à $[i][j]$ usamos:
  - **mat[i*m + j];**
    onde $0 \leq i \leq n-1$ e $0 \leq j \leq m-1$.
- Note que i pula de blocos de tamanho m, e j indexa a posição dentro de um bloco.

Linearização de Índices

- Podemos estender para mais dimensões. Seja matriz **mat[n*m*q]**, representando **mat[n][m][q]**.
  - As posições de 0 até $(m * q) - 1$ são da primeira matriz.
  - As posições de $(m * q)$ até $(2 * m * q) - 1$ são da segunda matriz.
  - Etc...
- De forma geral, seja matriz **mat[n*m*q]**, representando **mat[n][m][q]**.
- Para acessar a posição correspondente à $[i][j][k]$ usamos:
  - **mat[i*m*q + j*q + k];**

Linearização de Índices

```
int main(){
  int mat[40]; //representando mat[5][8]
  int i,j;

  for(i=0; i<5; i++)
    for(j=0;j<8; j++)
      mat[i*8 + j] = i*j;

  for(i=0; i<5; i++){
    for(j=0;j<8; j++)
      printf("%d, ",mat[i*8 + j]);
    printf("\n");
  }

}
```