Alocação Dinâmica de Memória

flavius@dct.ufrn.nr

Conceitos iniciais

- Gerenciamento de memória é um fator crítico no desenvolvimento de software
- Bons programas gerenciam a memória de forma eficiente, são mais rápidos e consomem menos recursos
- O gerenciamento eficiente de memória é crítico em alguns tipos de sistemas, tais como: celulares, dispositivos móveis, sistemas embarcados, robótica, etc.
- Ex: Um programa que precisa calcular a média e a variância de um conjunto indeterminado de valores
  - Quanto de memória deve ser alocado?
  - Como dimensionar um vetor se os dados que irão preenchê-lo estiverem sendo lidos de um arquivo?

Uso de memória

Há basicamente três formas de reservar espaço de memória para o armazenamento de informações em um programa:

- Uso de variáveis globais (estáticas)
- Uso de variáveis locais (escopo limitado)
- Requisitando memória ao sistema operacional em tempo de execução (alocação dinâmica)

Um espaço de memória alocado dinamicamente permanece reservado ao programa até o seu encerramento ou até que seja explicitamente liberado (desalocação)

Outras funções do programa podem usar esse espaço de memória

Vantagens e desvantagens de cada uma

- Variáveis globais e estáticas
  - Vantagens: facilidade de uso, praticidade
  - Desvantagens: riscos, desperdício de memória
- Variáveis locais
  - Vantagens: bom gerenciamento de memória
  - Desvantagens: estruturas rígidas, tamanho fixo
- Variáveis alocadas em tempo de execução
  - Vantagens: flexibilidade, economia de memória
  - Desvantagens: exigem atenção ao alocar e desalocar

Variáveis globais versus locais

```c
#include <stdio.h>
int vet[1000];
void preencheVetor(int);
void exibeVetor(int);
int main(void) {
 int n;
 printf("Informe tamanho do vetor: \n");
 scanf("%d", &n);
 preencheVetor(n);
 exibeVetor(n);
 return 0;
}
void preencheVetor(int n) {
 for (int i=0; i<n; i++) {
   printf("v[%d]: ", i+1);
   scanf("%d", &vet[i]);
 }
}
void exibeVetor(int n) {
 for (int i=0; i<n; i++) {
   printf("v[%d] = %d\n", i+1, vet[i]);
 }
}
```

```c
#include <stdio.h>
void preencheVetor(int, int*);
void exibeVetor(int, int*);
int main(void) {
 int n, vet[1000];
 printf("Informe tamanho do vetor: \n");
 scanf("%d", &n);
 preencheVetor(n, vet);
 exibeVetor(n, vet);
 return 0;
}
void preencheVetor(int n, int *v) {
 for (int i=0; i<n; i++) {
   printf("v[%d]: ", i+1);
   scanf("%d", &v[i]);
 }
}
void exibeVetor(int n, int *v) {
 for (int i=0; i<n; i++) {
   printf("v[%d] = %d\n", i+1, v[i]);
 }
}
```

Alocação dinâmica de memória

- O gerenciamento de memória da maneira anterior só é eficiente quando o programador sabe EXATAMENTE quanto de memória é necessário em cada campo
- Se o programador aloca mais memória do que o necessário, há desperdício de memória
- Se o programador aloca menos memória do que o necessário, a estrutura não atende às necessidades
- A solução é gerenciar a memória de forma eficiente, solicitando memória apenas quando necessário e na quantidade desejada
- Da mesma forma, o programador deve devolver o espaço de memória solicitado quando este não for mais necessário

Alocação dinâmica de memória

Linguagens de programação mais eficientes possuem operadores para alocar e desalocar memória, quando necessário:

Na linguagem C:

```
void* malloc(size_t size);

void free(void *ptr);
```

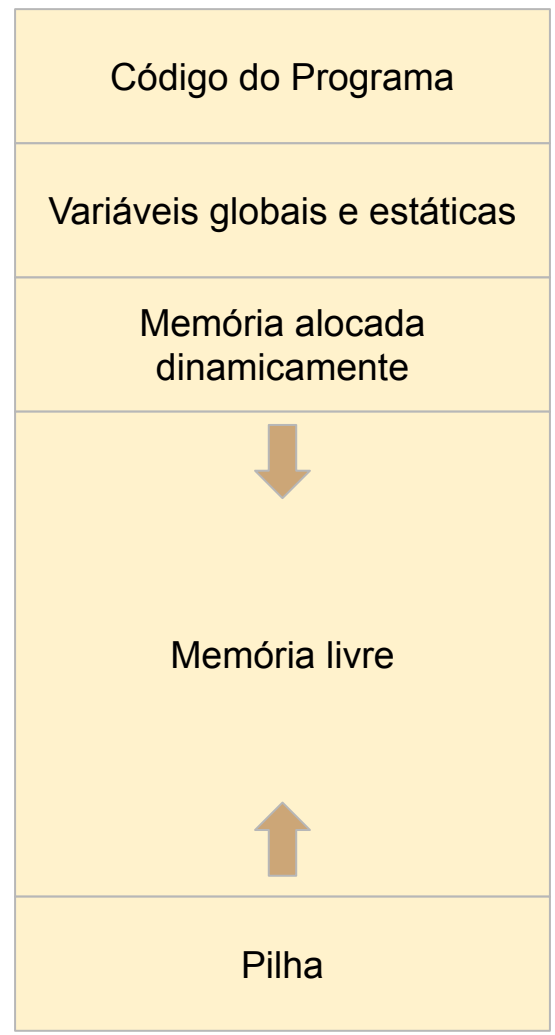
Na linguagem C++:

```
void* new size_t;

void delete *ptr;
```

```
void* new type_t[size_t];

void delete[] *ptr;
```

Exemplo de código com alocação dinâmica

```c
#include <stdio.h>
#include <stdlib.h>
void preencheVetor(int, int*);
void exibeVetor(int, int*);
int main(void) {
 int n, *vet;
 printf("Informe tamanho do vetor: \n");
 scanf("%d", &n);
 vet = (int*) malloc(n*sizeof(int));
 preencheVetor(n, vet);
 exibeVetor(n, vet);
 free(vet);
 return 0;
}
void preencheVetor(int n, int *v) {
 for (int i=0; i<n; i++) {
   printf("v[%d]: ", i+1);
   scanf("%d", &v[i]);
 }
}
void exibeVetor(int n, int *v) {
 for (int i=0; i<n; i++) {
   printf("v[%d] = %d\n", i+1, v[i]);
 }
}
```



Verificando disponibilidade de memória

Eventualmente, pode ser que não haja memória suficiente para realizar a alocação solicitada

Nesse caso, o sistema operacional retornará um ponteiro NULL

NULL significa um endereço nulo, que não pode ser acessado ou referenciado

O programa deve tratar adequadamente a inexistência de memória disponível

```
//... Trecho de programa

   int *vet;
   ...
   vet = (int*) malloc(tam * sizeof(int));
   if (vet == NULL) {
       printf("Não foi possível alocar memória!\n");
       exit(1);
   }

//...
```

Problemas com o uso de alocação dinâmica

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

void preencheVetor(int, int*);
void exibeVetor(int, int*);

int main(void) {
 int n, *vet;
 printf("Informe tamanho do vetor: \n");
 scanf("%d", &n);
 vet = (int*) malloc(n*sizeof(int));
 preencheVetor(n, vet);
 exibeVetor(n, vet);
 free(vet);

// Problema de acesso a área de memória não disponível
 printf("Vetor[0]: %d\n", vet[0]);

 return 0;
}
```

O uso de alocação dinâmica requer alguns cuidados adicionais

- Não se pode acessar área de memória que não estejam mais disponíveis
- O código ao lado tenta acessar uma área de memória que não está mais disponível, pois já foi devolvida ao sistema operacional

Problemas com o uso de alocação dinâmica

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

void preencheVetor(int, int*);
void exibeVetor(int, int*);

int main(void) {
 int n, *vet;
 printf("Informe tamanho do vetor: \n");
 scanf("%d", &n);
 vet = (int*) malloc(n*sizeof(int));
 preencheVetor(n, vet);
 exibeVetor(n, vet);

// Problema de espaço de memória perdido
 vet = (int*) malloc(n*sizeof(int));
 preencheVetor(n, vet);
 exibeVetor(n, vet);
 free(vet);
 return 0;
}
```

O uso de alocação dinâmica requer alguns cuidados adicionais

- Não se pode “largar” áreas de memória que não tenham sido liberadas adequadamente para o sistema operacional
- O código ao lado desperdiça uma área de memória que não está mais sendo utilizada
- Essa área de memória não poderá mais ser recuperada, pois o seu apontador foi perdido

Uma aplicação para a alocação dinâmica

Nessa implementação, a variável é criada de forma estática dentro da função e não existirá quando a função for encerrada

```
float* prodVetorial(float *u, float *v) {
 float w[3];
 w[0] = u[1]*v[2] - v[1]*u[2];
 w[1] = u[2]*v[0] - v[2]*u[0];
 w[2] = u[0]*v[1] - v[0]*u[1];
 return w;
}
```

Nessa implementação, a variável é criada de forma dinâmica dentro da função e continuará existindo quando a função for encerrada

```
float* prodVetorial(float *u, float *v) {
 float *w = (float*) malloc(3*sizeof(float));
 w[0] = u[1]*v[2] - v[1]*u[2];
 w[1] = u[2]*v[0] - v[2]*u[0];
 w[2] = u[0]*v[1] - v[0]*u[1];
 return w;
}
```

Alocação de vetores multidimensionais

As matrizes e demais vetores multidimensionais sofrem das mesmas limitações que os vetores unidimensionais

Quando o tamanho da matriz é conhecido previamente, pode-se fazer a alocação de memória de forma estática. Porém, quando este não é conhecido, necessita-se utilizar alocação dinâmica

A função `malloc()` e outras funções similares de alocação de memória só reservam blocos de memória contígua

Para trabalhar com matrizes alocadas dinamicamente, é necessário criar algumas abstrações conceituais a partir de vetores

Matriz representada por vetor simples

Cada elemento da matriz m_{ij} é mapeado em um elemento de um vetor v_k

Alocação de memória para um vetor de tamanho m x n

```
v = (int*) malloc(m*n*sizeof(int));
```

mat[i][j] ≈ v[k]

k = i * n + j

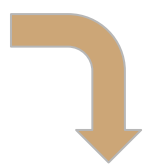
i: linha do elemento

j: coluna do elemento

n: número de colunas da matriz

mat

1	2	3	4
5	6	7	8
9	10	11	12



v

1	2	3	4	5	6	7	8	9	10	11	12

Matriz representada por vetor simples

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

void preencheMatriz(int, int, int*);
void exibeMatriz(int, int, int*);

int main(void) {
 int m, n, *mat;
 printf("Informe tamanho da matriz: \n");
 printf("Linhas: ");
 scanf("%d", &m);
 printf("Colunas: ");
 scanf("%d", &n);
 mat = (int*) malloc(m*n*sizeof(int));
 printf("\nInforme os elementos da matriz\n");
 preencheMatriz(m, n, mat);
 printf("\nMatriz Lida\n");
 exibeMatriz(m, n, mat);
 free(mat);
 return 0;
}
```

```c
void preencheMatriz(int m, int n, int *v) {
 int k;
 for (int i=0; i<m; i++) {
   for (int j=0; j<n; j++) {
     printf("mat[%d][%d]: ", i+1, j+1);
     k = i*n + j;
     scanf("%d", &v[k]);
   }
 }
}

void exibeMatriz(int m, int n, int *v) {
 for (int k=0; k<m*n; k++) {
   printf("%3d", v[k]);
   if ((k+1)%n==0) {
     printf("\n");
   }
 }
}
```

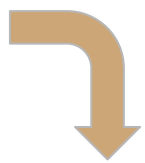
https://repl.it/@flaviusgorgonio/MatrizDinamicaRepresentadaPorVetor#main.c

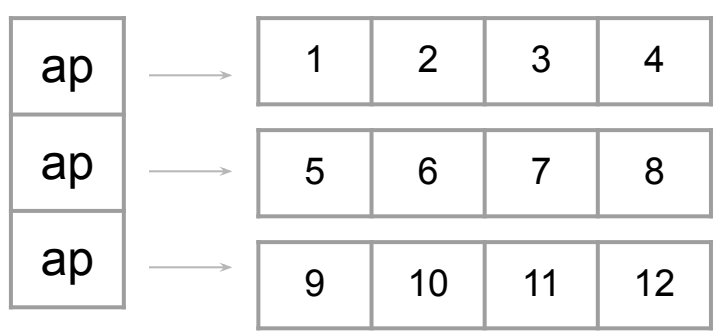
Matriz com vetor de apontadores

Cada linha da matriz é representada por um vetor independente

A matriz é representada por um vetor de vetores (na verdade, de apontadores)

1	2	3	4
5	6	7	8
9	10	11	12





Cada elemento do vetor armazena o endereço do primeiro elemento de cada vetor linha, onde estão os elementos

```
mat = (int**) malloc(m*sizeof(int*));
for (int k=0; k<m; k++) {
  mat[k] = (int*) malloc(n*sizeof(int))
}

for (int k=0; k<m; k++) {
  free(mat[i]);
}
free(mat);
```

Matriz com vetor de apontadores

```c
#include <stdio.h>
#include <stdlib.h>
void preencheMatriz(int, int, int**);
void exibeMatriz(int, int, int**);
int main(void) {
 int m, n, **mat;
 printf("Informe tamanho da matriz: \n");
 printf("Linhas: ");
 scanf("%d", &m);
 printf("Colunas: ");
 scanf("%d", &n);
 mat = (int**) malloc(m*sizeof(int*));
 for (int k=0; k<m; k++) {
   mat[k] = (int*) malloc(n*sizeof(int));
 }
 printf("\nInforme os elementos da matriz\n");
 preencheMatriz(m, n, mat);
 printf("\nMatriz Lida\n");
 exibeMatriz(m, n, mat);
 for (int k=0; k<m; k++) {
   free(mat[k]);
 }
 free(mat);
 return 0;
}
```

```c
void preencheMatriz(int m, int n, int **mat) {
 for (int i=0; i<m; i++) {
   for (int j=0; j<n; j++) {
     printf("mat[%d][%d]: ", i+1, j+1);
     scanf("%d", &mat[i][j]);
   }
 }
}
```

```c
void exibeMatriz(int m, int n, int **mat) {
 for (int i=0; i<m; i++) {
   for (int j=0; j<n; j++) {
     printf("%3d", mat[i][j]);
   }
   printf("\n");
 }
}
```

https://repl.it/@flaviusgorgonio/MatrizDinamicaRepresentadaPorApontadores#main.c

Criação de uma matriz transposta

```
int* transposta(int, int, int*);
...
int* transposta(int m, int n, int* mat) {
 int* trp = (int*) malloc(m*n*sizeof(int));
 for (int i=0; i<m; i++) {
   for (int j=0; j<n; j++) {
     trp[j*m+i] = mat[i*n+j];
   }
 }
 return trp;
}
...
int main(void) {
 int m, n, *mat, *trp;
 ...
 trp = transposta(m, n, mat);
 printf("\nMatriz Transposta\n");
 exibeMatriz(m, n, trp);
 free(mat);
 free(trp);
 return 0;
}
```

```
int** transposta(int, int, int**);
...
int** transposta(int m, int n, int** mat) {
 int** trp = (int**) malloc(n*sizeof(int*));
 for (int i=0; i<n; i++) {
   trp[i] = (int*) malloc(m*sizeof(int));
 }
 for (int i=0; i<m; i++) {
   for (int j=0; j<n; j++) {
     trp[i][j] = mat[j][i];
   }
 }
 return trp;
}
...
int main(void) {
 int m, n, **mat, **trp;
...
 for (int k=0; k<n; k++) {
   free(trp[k]);
 }
 free(trp);
 return 0;
}
```

Bibliografia Recomendada

Celes, W. *et al*. Introdução a Estruturas de Dados com Técnicas de Programação em C, 2ª ed. Rio de Janeiro: Elsevier, 2016.

Sebesta, R. W., Conceitos de Linguagens de Programação, 9ª ed. Porto Aleg Bookman, 2016.

Varejão, F., Linguagens de Programação: Conceitos e Técnicas, Rio de Janei Campus, 2004.