Estrutura de Dados

Aula 3 : Estruturas Dinâmicas

Prof. MSc. Fausto Sampaio

https://github.com/Fausto14/estrutura_de_dados

Centro Universitário UniFanor - Wyden

5 de novembro de 2019

Sumário

Vetores e alocação dinâmica

Vetores e alocação dinâmica

- A forma mais simples de estruturarmos um conjunto de dados é por meio de vetores;
- Definimos um vetor em C da seguinte forma: *intv*[10];
- Esta declaração diz que v é um vetor de inteiros dimensionado com 10 elementos, isto é, reservamos um espaço de memória contínuo para armazenar 10 valores inteiros. Assim, se cada int ocupa 4 bytes, a declaração reserva um espaço de memória de 40 bytes;

Vetores e alocação dinâmica

- O acesso a cada elemento do vetor é feito através de uma indexação da variável V .

Em C, a indexação de um vetor varia de zero a $n - 1$, onde n representa a dimensão do vetor.

- v[0] acessa o primeiro elemento de v;
- v[1] acessa o segundo elemento de v ...
- v[10] invade a memória (erro).

Vetores e alocação dinâmica

```
float v[10];
int i;
/* leitura dos valores */
for (i = 0; i < 10; i++)
    scanf("%f", &v[i] );
```

- Passamos para scanf o endereço de cada elemento do vetor (&v[i]), pois desejamos que os valores capturados sejam armazenados nos elementos do vetor.
- Se v[i] representa o (i+1)-ésimo elemento do vetor, &v[i] representa o endereço de memória onde esse elemento está armazenado.
- Existe uma associação entre vetores e ponteiros, pois a variável v , que representa o vetor, é uma constante que armazena o endereço inicial do vetor, isto é, v , sem indexação, aponta para o primeiro elemento do vetor.

Vetores e alocação dinâmica

- Em um vetor temos as seguintes equivalências:
  - v + 0 : aponta para o primeiro elemento do vetor
  - v + 1 : aponta para o segundo elemento do vetor
  - v + 9 : aponta para o último elemento do vetor
- Portanto, escrever &v[i] é equivalente a escrever (v+i).
- De maneira análoga, escrever v[i] é equivalente a escrever *(v+i) (é lógico que a forma indexada é mais clara e adequada).
- Note que o uso da aritmética de ponteiros aqui é perfeitamente válido, pois os elementos dos vetores são armazenados de forma contínua na memória.

Vetores e alocação dinâmica

- Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor. Se passarmos um valor de endereço, a função chamada deve ter um parâmetro do tipo ponteiro para armazenar este valor. Assim, se passarmos para uma função um vetor de int , devemos ter um parâmetro do tipo int* , capaz de armazenar endereços de inteiros.
- Sendo assim, a expressão "passar um vetor para uma função" deve ser interpretada como "passar o endereço inicial do vetor". Os elementos do vetor não são copiados para a função, o argumento copiado é apenas o endereço do primeiro elemento.

Vetores e alocação dinâmica

```
/* Incrementa elementos de um vetor */
#include <stdio.h>
void incr_vetor ( int n, int *v ) {
      int i;
      for (i = 0; i < n; i++)
          v[i]++;
}
int main ( void ) {
    int a[ ] = {1, 3, 5};
    incr_vetor(3, a);
    printf("%d %d %d \n", a[0], a[1], a[2]);
    return 0;
}
```

A saída do programa é 2 4 6 , pois os elementos do vetor serão incrementados dentro da função.

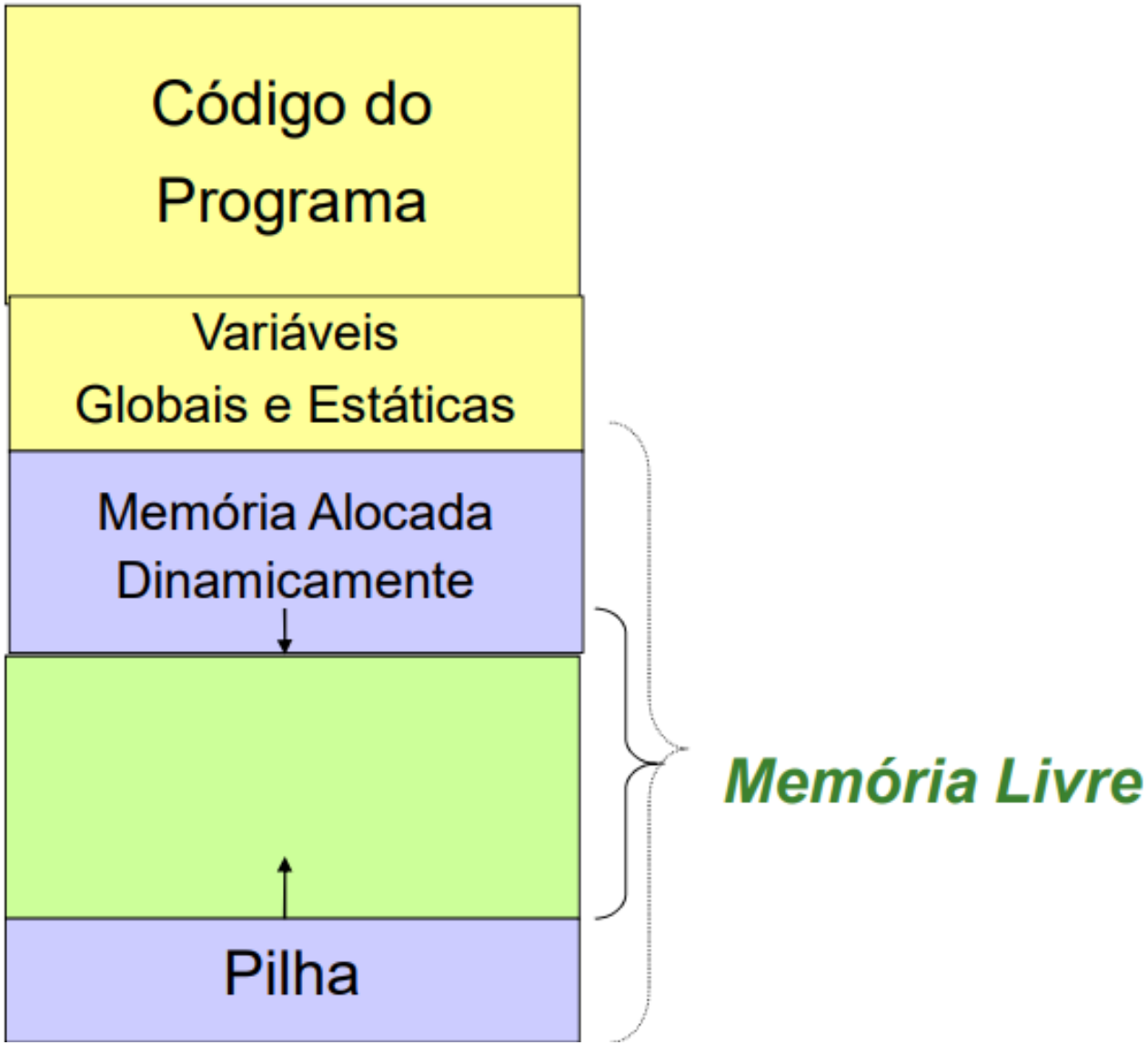
Alocar memória dinamicamente

- A linguagem C oferece meios de requisitarmos espaços de memória em tempo de execução.
- **Uso de Memória**:
  1. uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado.
  2. uso de variáveis locais. Neste caso, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Assim, a função que chama não pode fazer referência ao espaço local da função chamada.
  3. reservar memória requisitando ao sistema, em tempo de execução, um espaço de um determinado tamanho.

Alocar memória dinamicamente

- O espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa. Por isso, podemos alocar dinamicamente um espaço de memória numa função e acessá-lo em outra.
- A partir do momento que liberarmos o espaço, ele estará disponibilizado para outros usos e não podemos mais acessá-lo. Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar.

Alocação da Memória Principal



Alocação da Memória Principal

- Para executar um determinado programa, o S.O. carrega na memória o **código do programa**, em linguagem de máquina. Além disso, o S.O. reserva os espaços necessários para armazenar as **variáveis globais** (e estáticas) do programa.
- O restante da **memória livre** é utilizado pelas variáveis locais e pelas variáveis **alocadas dinamicamente**.

Alocação da Memória Principal

- Cada vez que uma função é chamada, o S.O. reserva o espaço necessário para as variáveis locais da função. Este espaço pertence à **pilha** de execução e, quando a função termina, é liberado.
- A memória não ocupada pela pilha de execução **pode ser requisitada dinamicamente**. Se a pilha tentar crescer mais do que o espaço disponível existente, dizemos que ela "estourou" e o programa é abortado com erro.

Alocação Dinâmica de Memória

- As funções *calloc* e *malloc* permitem alocar blocos de memória em tempo de execução.

```
void * malloc( size    );
```


```
/*
  retorna um ponteiro void para n bytes de memória não iniciados.
  Se não há memória disponível malloc retorna NULL
*/
```

Alocação da Memória Principal

- EXEMPLOS:
  - Código que aloca 1000 bytes de memória livre:

```
char *p;
p = malloc(1000);
```

  - Código que aloca espaço para 50 inteiros:

```
int *p;
p = malloc(50*sizeof(int));
```

Obs.: O operador sizeof() retorna o número de bytes de um inteiro.

Funções para Alocar e Liberar memória

- Com **malloc**, se não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo NULL , definido em stdlib.h).
- Podemos tratar o erro na alocação do programa verificando o valor de retorno da função malloc.
- Ex: imprimindo mensagem e abortando o programa com a função exit, também definida na stdlib.

```
...
Int * v;
v = (int*) malloc(10*sizeof(int));
if (v == NULL) {
     printf("Memoria insuficiente.\n");
     exit(1);
     /* aborta o programa e retorna 1 para o SO */
} ...
```

Liberação de Memória

```
int *pi = (int *) malloc (sizeof(int));
/* aloca espaco para um inteiro */
```

```
int *ai = (int *) calloc (n, sizeof(int));
/* aloca espaco para um array de n inteiros */
```

- toda memória não mais utilizada deve ser liberada através da função **free()**:

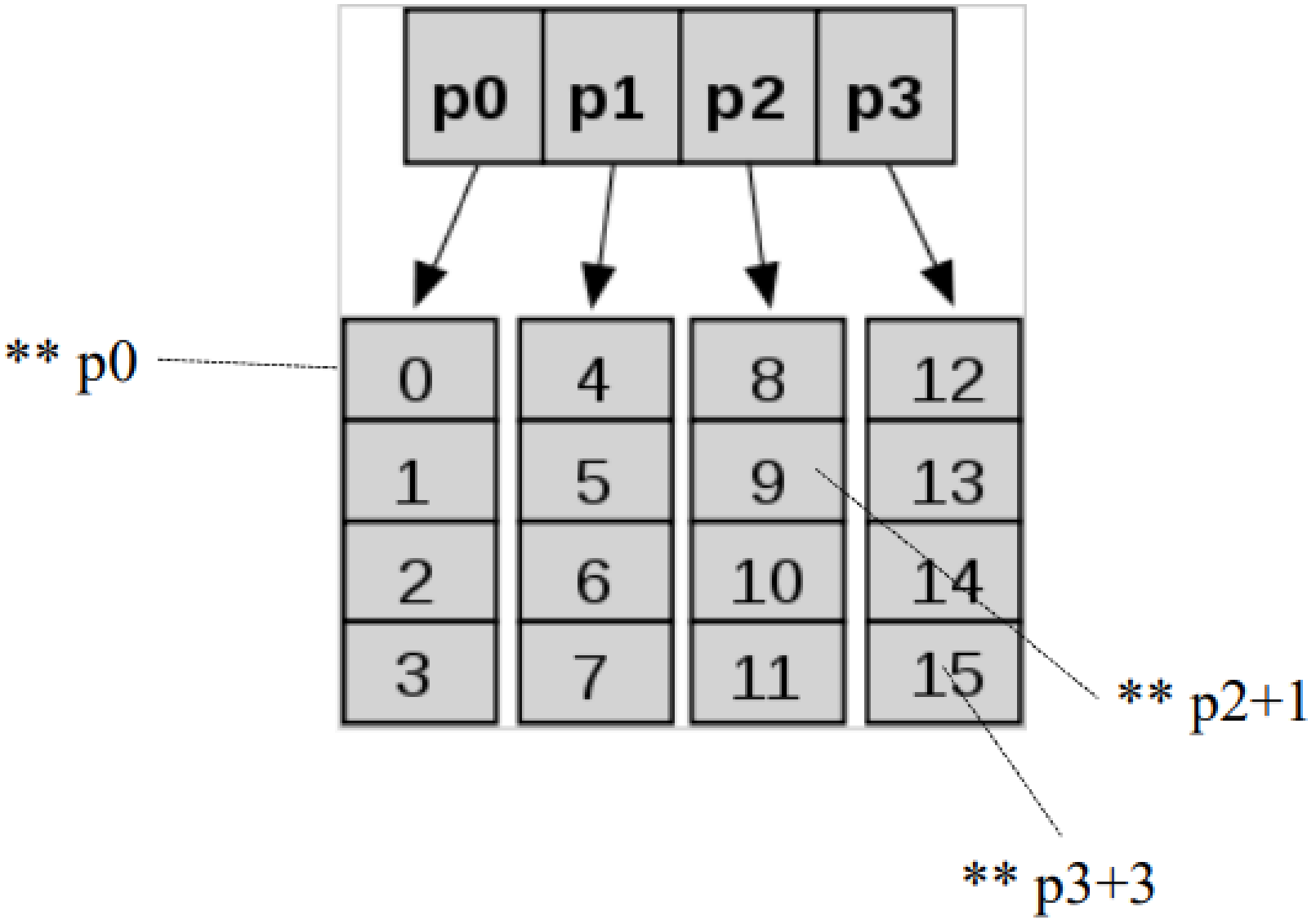
```
free(ai); /* libera todo o array */
free(pi); /* libera o inteiro alocado */
```

Alocação dinâmica de matrizes

Alocação dinâmica de matrizes

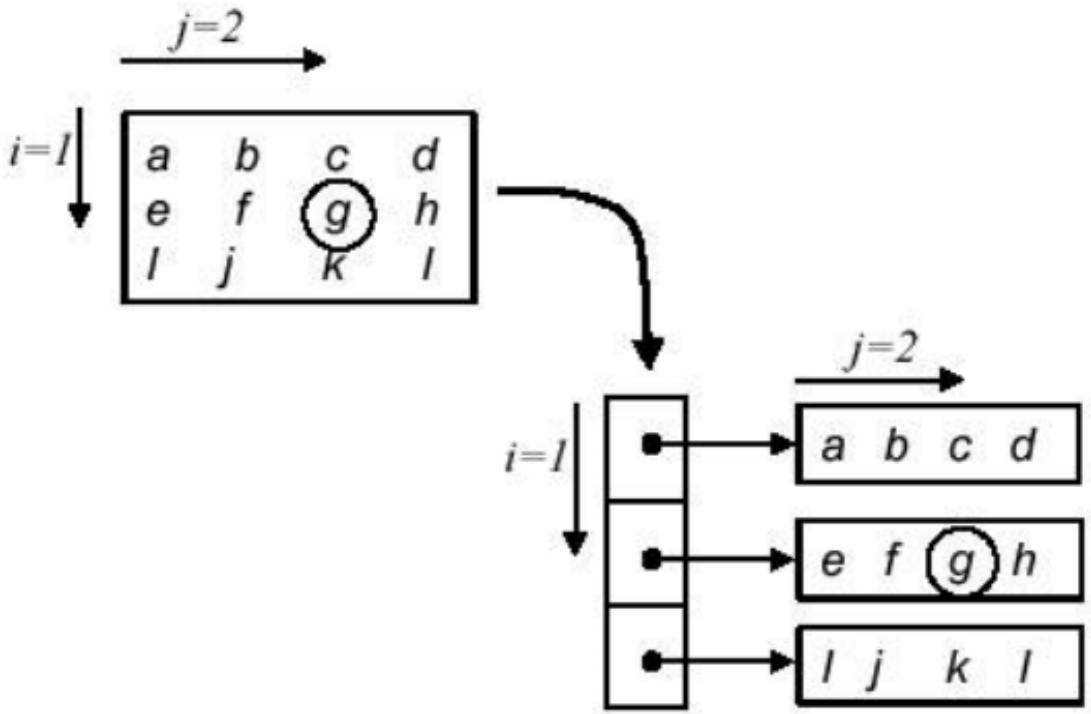
- A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, o que é denominado indireção múltipla.
- A indireção múltipla pode ser levada a qualquer dimensão desejada.

Alocação dinâmica de matrizes



Alocação dinâmica de matrizes

- Cada linha da matriz é representada por um vetor independente.
- A matriz é então representada por um vetor de vetores, ou vetor de ponteiros, no qual cada elemento armazena o endereço do primeiro elemento de cada linha.



Alocação dinâmica de matrizes

```c
float **Alocar_matriz_real (int m, int n) {
    float **v; /* ponteiro para a matriz */
    int i; /* variavel auxiliar */

    if (m < 1 || n < 1) { /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (NULL);
    }
    /* aloca as linhas da matriz */
    v = (float **) calloc (m, sizeof(float *));
    if (v == NULL) {
        printf ("** Erro: Memoria Insuficiente **");
        return (NULL);
    }

    /* aloca as colunas da matriz */
    for ( i = 0; i < m; i++ ) {
        v[i] = (float*) calloc (n, sizeof(float));
        if (v[i] == NULL) {
            printf ("** Erro: Memoria Insuficiente **");
            return (NULL);
        }
    }
    return (v); /* retorna o ponteiro para a matriz */
}
```

Alocação dinâmica de matrizes

```c
float **Liberar_matriz_real (int m, int n, float **v) {
    int i; /* variavel auxiliar */
    if (v == NULL) return (NULL);

    if (m < 1 || n < 1) { /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (v);
    }
    for (i = 0; i < m; i++) free (v[i]); /* libera as linhas da matriz */
        free (v); /* libera a matriz */

    return (NULL); /* retorna um ponteiro nulo */
}

void main (void)
{
    float **mat; /* matriz a ser alocada */
    int l, c; /* numero de linhas e colunas da matriz */
    ... /* outros comandos, inclusive inicializacao para l e c */
    mat = Alocar_matriz_real (l, c);
    ... /* outros comandos utilizando mat[][] normalmente */
    mat = Liberar_matriz_real (l, c, mat);
...
}
```