Algoritmos e Estrutura de Dados II

Ponteiros e Alocação Dinâmica

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso
Faculdade de Engenharia

Agenda

Ponteiros

- Ponteiros ou apontadores, são variáveis que armazenam o endereço de memória de outras variáveis.
- Dizemos que um ponteiro “aponta” para uma varíável quando contém o endereço da mesma.
- Os ponteiros podem apontar para qualquer tipo de variável. Portanto temos ponteiros para int, float, double, etc..
- Um ponteiro pode ter o valor especial NULL, quando não contém nenhum endereço.

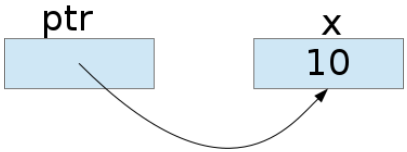
Por que usar ponteiros?

- Ponteiros são muito úteis quando uma variável tem que ser acessada em diferentes partes de um programa.
- Neste caso, o código pode ter vários ponteiros espalhados por diversas partes do programa, "apontando" para a variável que contém o dado desejado.
- Caso este dado seja alterado, não há problema algum, pois todas as partes do programa tem um ponteiro que aponta para o endereço onde reside o dado atualizado.
- Existem várias situações onde ponteiros são úteis, por exemplo:
  - Alocação dinâmica de memória
  - Manipulação de arrays.
  - Para retornar mais de um valor em uma função.
  - Referência para listas, pilhas, árvores e grafos.

Operadores de Ponteiros

- O operador & retorna o endereço de memória de uma variável.
- O operador * acessa o conteúdo do endereço indicado pelo ponteiro.

```
int *ptr, x = 10;
ptr = &x;
printf("ptr = %d\n",*ptr);
printf("Endereco ptr = %p\n", ptr);
printf("Endereco x = %p\n", &x);
```



- Imprime :

```
ptr = 10
Endereco ptr = 0x7ffc2b783634
Endereco x = 0x7ffc2b783634
```

Exemplo

```
// a, b e c sao variaveis locais, alocadas estaticamente
int a = 10, b = 20, c;
int *p, *q; // p e q sao ponteiros que apontam para um inteiro
p = &a; // o valor de p eh o endereco de a
q = &b; // q aponta para b
c = *p + *q; // c recebe o 10 + 20.
```

Exemplo

```
int a = 10; // a eh alocada estaticamente
int *p; // p eh um ponteiro que aponta para um inteiro
p = &a; // o valor de p eh o endereco de a
*p = *p + 1; // Incrementa em 1 o valor para o qual p aponta
printf(``a = %d'', a); // Imprime a = 11.
```

Exercício

Implemente um função que troque os valores de duas variáveis inteiras. Para isso, a função deve receber o endereço de duas variáveis, a fim de trocar o conteúdo das variáveis originais, e não apenas da cópia local.

Solução

```
void troca (int *p, int *q)
{
   int temp;
   temp = *p;
   *p = *q;
   *q = temp;
}
```

Alocação Estática x Alocação Dinâmica

- As vezes queremos queremos criar variáveis durante a execução.
- Mas por quê não declarar variáveis locais?
  - Não sabemos quantas variáveis e quando declará-las.
  - Uma função pode ter que criar uma variável para outras funções usarem.
  - Queremos usar uma organização mais complexa da memória (estrutura de dados).

Alocação Estática x Alocação Dinâmica

- Alocação Estática
  - O espaço para as variáveis é reservado e liberado automaticamente pelo compilador.
  - Exemplo:

```
int a; int b[20];
```

- Alocação Dinâmica
  - O espaço para as variáveis é reservado e liberado dinamicamente pelo programador.
  - Para isso, utiliza-se os comandos: malloc(), calloc() e resize().
  - Antes do término do programa, é necessário liberar a memória alocada.
  - Para isso, utiliza-se o comando free().

Função malloc()

- A função malloc aloca um bloco de bytes consecutivos na memória RAM e devolve o endereço desse bloco.
- No seguinte fragmento de código, aloca-se 1 byte:

```
char *ptr = malloc( 1 );
```

- Para alocar valores maiores, pode-se recorrer ao operador sizeof, que diz quantos bytes um determinado tipo ocupa. Exemplo:

```
int *ptr = malloc( sizeof( int ) );
```

Função calloc()

- A função calloc recebe como parâmetro o número de blocos de memória a serem alocados e o tamanho em bytes de cada bloco. Esse comando aloca a quantidade de memória atribuindo zero a todos os bits.
- Exemplo: alocar 1 inteiro:

```
int *p;
p = calloc(1, sizeof( int ) );
```

Função free()

- A função free desaloca a porção de memória alocada por malloc() ou calloc().
- A instrução free (ptr) avisa ao sistema que o bloco de bytes apontado por ptr está livre e disponível para reciclagem. Exemplo:

```
int *ptr;
ptr = malloc( sizeof( int ) );
free(ptr);
```

Memória Disponível

- Ao alocar memória, recomenda-se testar se há memória disponível.
- Caso não haja a função malloc() e calloc() retornam NULL. Exemplo:

```
int *ptr;
ptr = malloc(sizeof(int));
if (ptr == NULL) {
  printf("Nao ha mais memoria!\n");
  exit(0);
}
*ptr = 13;
printf("Endereco %p com valor %d.\n", ptr, *ptr);
free(ptr);
```

Imprime:
Endereco 0x23cd010 com valor 13.

Alocação Dinâmica

Regras da Alocação Dinâmica

- Devemos incluir a biblioteca stdlib.h.
- O tamanho gasto por um tipo pode ser obtido com sizeof().
- Devemos informar o tamanho a ser reservado para malloc().
- Devemos verificar se acabou a memória comparando com NULL.
- Devemos sempre liberar a memória após a utilização com free().

Pilha e Heap

- A memória de um programa é dividida em duas partes:
  - Pilha: Onde são armazenadas as variáveis locais, alocadas estaticamente.
  - Heap: Onde são armazenadas as variáveis alocadas dinamicamente, usando malloc(), calloc() ou resize().

Pilha e Heap

Pilha

- Ao declarar uma variável local, o compilador reserva um espaço na pilha.
- Ao realizar chamadas de funções, os valores são empilhados.
- O espaço reservado para uma variável local é liberado quando a função termina.

Pilha e Heap

Heap

- Ao alocar memória utilizando o comando malloc() ou calloc(), o compilador reserva um espaço no heap, em tempo de execução (*runtime*).
- O espaço reservado usando malloc() deve ser liberado manualmente pelo programador, caso contrário, acontece uma falha de sistema denominada vazamento de memória (*memory leak*).
- O programa valgrind ajuda a detectar vazamentos de memória.

Pilha e Heap

Representação da Memória

Alocação Dinâmica

- Também é possível alocar dinamicamente uma quantidade de memória contígua e associá-la com um ponteiro.
- Desta forma podemos criar programas sem saber a priori o número de dados a ser armazenado.
- A seguir, um exemplo de alocação dinâmica de um vetor.

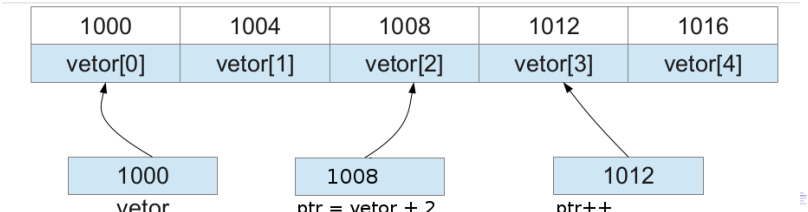
Exemplo: alocação de vetor

```
int *vetor, i, n;
scanf("%d", &n);
vetor = malloc(n * sizeof( int ) );
for (i = 0; i < n; i++)
  scanf("%d", &vetor[i] );
free(vetor);
```

Aritmética de ponteiros

- Podemos realizar operações aritméticas em ponteiros: soma, subtração, incremento e decremento.
- No entanto, ao realizar essas operações, estaremos realizando deslocamentos no endereço. Exemplo:

```
int vetor[] = {1, 2, 3, 4, 5}, *ptr;
ptr = vetor + 2;
printf("ptr: %d\n", *ptr);
ptr++;
printf("ptr: %d\n", *ptr);
```



Exercícios

- Escreva uma função que compara duas strings e retorna 1 se a primeira string vier antes da segunda na ordem alfabética.

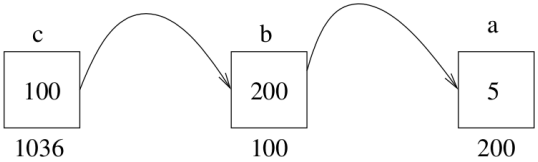
Ponteiros para ponteiros

- 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 duplo, ou seja, um ponteiro para outro ponteiro: tipo `**nomePonteiro`.

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

Ponteiros para Ponteiros

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

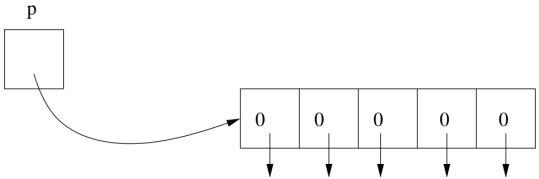


Ponteiros para Ponteiros

- A mesma coisa acontece se declararmos um vetor de ponteiros duplos.

```
int **p;
p = calloc(5, sizeof( int *) );
```

- Teremos um vetor, onde cada posição do vetor é do tipo int *, ou seja, é um ponteiro para um inteiro.

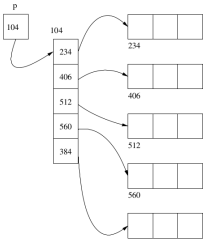


Ponteiros para Ponteiros

- 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 ) );
```

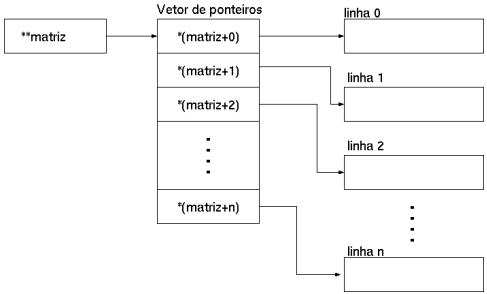
- Dessa forma, teremos uma matriz alocada dinamicamente.



Matriz Dinâmica

Esta é a 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 associado 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).



Matriz Dinâmica

- No final você deve desalocar toda a memória alocada.
- Para desalocar toda a memória alocada, percorre-se cada linha da matriz liberando a memória.

```
int **m = calloc(5, sizeof(int *));
for (int i = 0; i < 5; i++)
  m[i] = calloc(5, sizeof(int));
...
for (int i = 0; i < 5; i++)
  free(m[i]);
free(m);
```

Exercícios

- Escreva um programa que leia uma quantidade arbitrária de nomes e imprima esses nomes em ordem alfabética

Exercício

Crie um programa que multiplica duas matrizes quadradas do tipo double lidas do teclado. Seu programa de ler a dimensão n da matriz, em seguida alocar dinamicamente duas matrizes $n \times n$. Depois ler os dados das duas matrizes e imprimir a matriz resultante da multiplicação destas.

Conclusão

- Nesta aula foram apresentados conceitos e exemplos de alocação dinâmica de mémoria.
- Pontos de maior atenção para alocação e liberação de memória e passagem de parâmetros por valor e referência.
- Em seus programas você deverá estar atento ao momento exato de liberação da memória.
- Tipos Abstratos de Dados (TADs).

Material Recomendado

- DEITEL, Paul; DEITEL, Harvey. C: Como Programar. Capítulo 7. Pearson, 6ª ed. 2011. Disponível aqui.
- Tutorial de uso do software Valgrind. Disponível aqui.