Paradigma Imperativo
Funções e gerenciamentos de memória

Fabio Lubacheski
fabio.lubacheski@mackenzie.br

Funções – Capítulo 9 Tucker

- Funções são elementos fundamentais em toda linguagem de programação, já que são ferramentas essenciais para **abstração** em programação.

- Em linguagens diferentes, as funções são conhecidas como **procedimentos**, **sub-rotinas**, **subprogramas** ou **métodos**, e possuem diversas características em comum, assim como algumas diferenças importantes nas mais variadas linguagens.

Parâmetros – Capítulo 9 Tucker

- Funções derivam sua grande utilidade tanto em matemática quanto em linguagens de programação pela sua capacidade de receber parâmetros.
- Por exemplo, uma função de raiz quadrada não seria muito útil se não pudesse receber um argumento que especificasse o valor cuja raiz quadrada devesse ser calculada.

- Uma expressão que aparece em uma chamada de função é denominada **argumento**.

- Um identificador que aparece em uma declaração de função denominado **parâmetro**.

Exemplo parâmetro argumento

- Considere o exemplo:

```
// x e y parâmetros
void troca( int x, int y){
     int temp;
     temp = x;
     x = y;
     y = temp;
}
int main(){
    int a=10, b=20;
    troca(a,b); // a e b argumentos
    printf("a=%d b=%d\n",a,b);
    return 0;
}
```

Mecanismos de passagem de parâmetros – Capítulo 9 Tucker

- Os **valores** associados aos parâmetros durante a vida de uma função chamada são determinados com base em como esses **argumentos** são **passados** para a função pela chamada.
- Há cinco formas principais de **passar um argumento** para uma função:

  1 - Por valor,

  2 - Por referência,

  3 - Por resultado-valor,

  4 - Por resultado, e

  5 - Por nome.
- Entre essas formas, as duas primeiras são as mais usadas.

Passagem por valor – Capítulo 9 Tucker

- Passar um **argumento por valor** significa que o valor do argumento é calculado no tempo da chamada e **copiado para o parâmetro correspondente**.
- Considere a função que que efetua a troca dos valores de duas variáveis, a função é implementada na **linguagem C**.

```
void troca (int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

Passagem por valor – Capítulo 9 Tucker

- A chamada da função passa os argumentos (**a** e **b**) por valor para função **troca()**:

```
int main (void) {
   int a=10, b=20;
   troca(a,b);
   printf("a=%d b=%d\n",a,b);
```

Será que funciona ?

Passagem por valor – Capítulo 9 Tucker

- A chamada da função passa os argumentos (**a** e **b**) por valor para função **troca()**:

```
int main (void) {
   int a=10, b=20;
   troca(a,b);
   printf("a=%d b=%d\n",a,b);
```

Será que funciona ?

NÃO FUNCIONA, embora os valores dos parâmetros **x** e **y** sejam de fato trocados, esses valores não são **copiados** para os argumentos (**a** e **b**). Assim, a passagem por valor é, muitas vezes, chamada de mecanismo de **cópia**.

Passagem por Referência – Capítulo 9 Tucker

- Em C, uma função de troca pode ser escrita com o uso de ponteiros, da seguinte forma:

```
void troca( int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

- Nesse caso, os valores dos parâmetros são ponteiros para outras variáveis. Considere a chamada, Os parâmetros **x** e y contêm os endereços &a e &b, respectivamente.

```
int a=10, b=20;
troca(&a,&b);
```

Passagem por Referência – Capítulo 9 Tucker

- Analogamente, para trocar os valores de variáveis inteiras a e b podemos usar:

```
int a=10, b=20;
int *p, *q;
p = &a;
q = &b;
troca (p, q);
```

- Passar um argumento **por referência** (ou **por endereço**) significa que o **endereço de memória** do argumento é copiado para o parâmetro correspondente, de modo que o parâmetro se torna uma **referência** (**ponteiro**) indireta ao argumento real.

Exercícios

1) Leia a seção 9.4 do livro do Tucker e explique como funciona os mecanismos de passagem de parâmetro por **resultado-valor**, por **resultado** e por **nome**. Para cada um dos mecanismos de o exemplo de uma linguagem que utiliza o mecanismo.

2) Reescreva o exemplo que faz a troca entre dois valores usando passagem por referência na linguagem C++.

3) Por que o código abaixo está errado?

```
void troca (int *x, int *y){
   int *temp; *temp = *x; *x = *y; *y = *temp;
}
```

Exercícios

4) Considere a função `f()` abaixo:

```
void f(int *x, int*y){
    *x = *x + 1;
    *y = *y + 1;
}
```

Qual seria a saída a seguinte chamada ?

```
int a=10, b=20;
f(&a,&b);
printf("a=%d b=%d\n",a,b);
```

E para a chamada:

```
int a=10, b=20;
f(&a,&a);
printf("a=%d b=%d\n",a,b);
```

Exercícios

5) Considere a função `f()` abaixo:

```
void f(int *x, int*y){
    *x++;
    *y++;
}
```

Qual seria a saída a seguinte chamada ?

```
int a=10, b=20;
f(&a,&b);
printf("a=%d b=%d\n",a,b);
```

E para a chamada:

```
int a=10, b=20;
f(&a,&a);
printf("a=%d b=%d\n",a,b);
```

Exercícios

6) Escreva uma função `xxx()` que converta minutos em horas-e-minutos. A função recebe um inteiro `mnts` e os endereços de duas variáveis inteiras, digamos `h` e `m`, e atribui valores a essas variáveis de modo que `m` contenha o tempo em minutos (menor que 60) `h` as horas. Escreva também uma função `main()` que use a função `xxx()`.
7) Escreva uma função `mm()` que receba um vetor inteiro `v[]`, o número de elementos no vetor e os endereços de duas variáveis inteiras por parâmetro, digamos `min` e `max`, a função armazena nessas variáveis o valor do menor e maior elemento no vetor , respectivamente. Escreva também a função `main()` que use a função `mm()`.

Exercícios

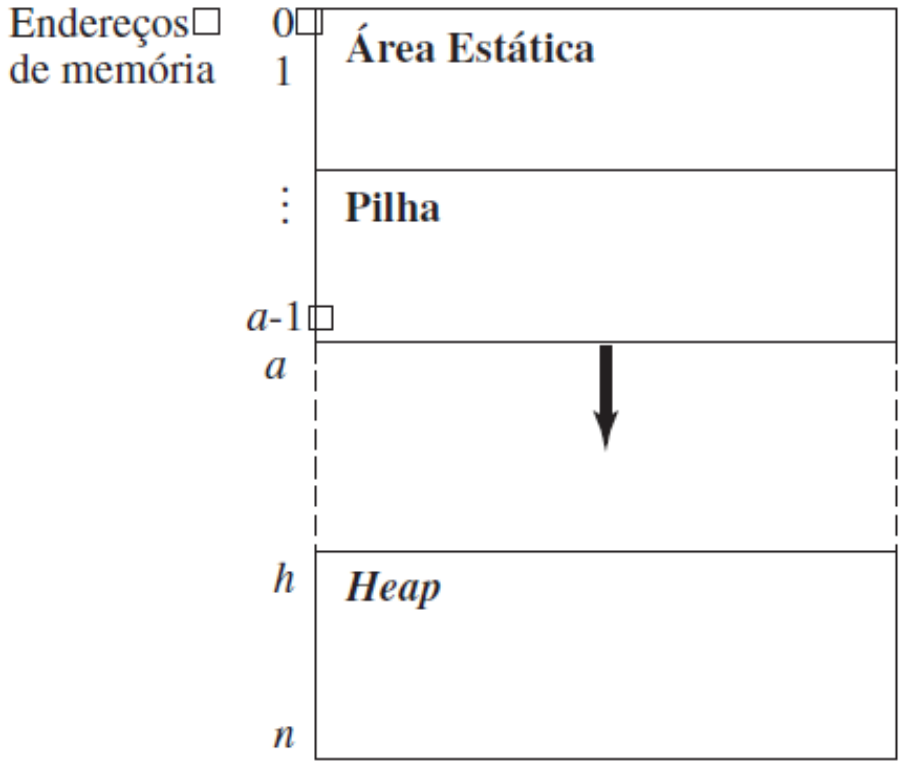
8) Escreva uma programa que receba três variáveis por parâmetro por referência, a sua função deve colocar as variáveis em ordem crescente, ou seja, na 1ª variável parâmetro o menor valor, na 2ª variável parâmetro o 2º menor valor e na 3ª variável parâmetro o maior valor. Escreva também uma função `main()` que teste a sua função.

Gerenciamento de memória – Capítulo 11 Tucker

- O gerenciamento em tempo de execução da memória dinâmica é uma atividade necessária para linguagens modernas de programação.
- Durante muito tempo, a automação dessa atividade tem sido um elemento-chave na implementação de linguagens de programação funcionais e lógicas.
- As linguagens C e C++ permitiram que o gerenciamento em tempo de execução da memória dinâmica fosse assumido pelo programador.
- Na linguagem Java o gerenciamento de memória é realizado pela Máquina Virtual Java (JVM), pois os criadores do Java consideram muito arriscado e propenso a erros deixar o gerenciamento de memória sob responsabilidade do programador

Memória heap – Capítulo 11 Tucker

- Nas linguagens como C, C++ e Java, a memória em tempo de execução (processo) pode ser visualizada como tendo três partes: a **área estática**, a **pilha de tempo de execução** e a **heap**.



Memória heap – Capítulo 11 Tucker

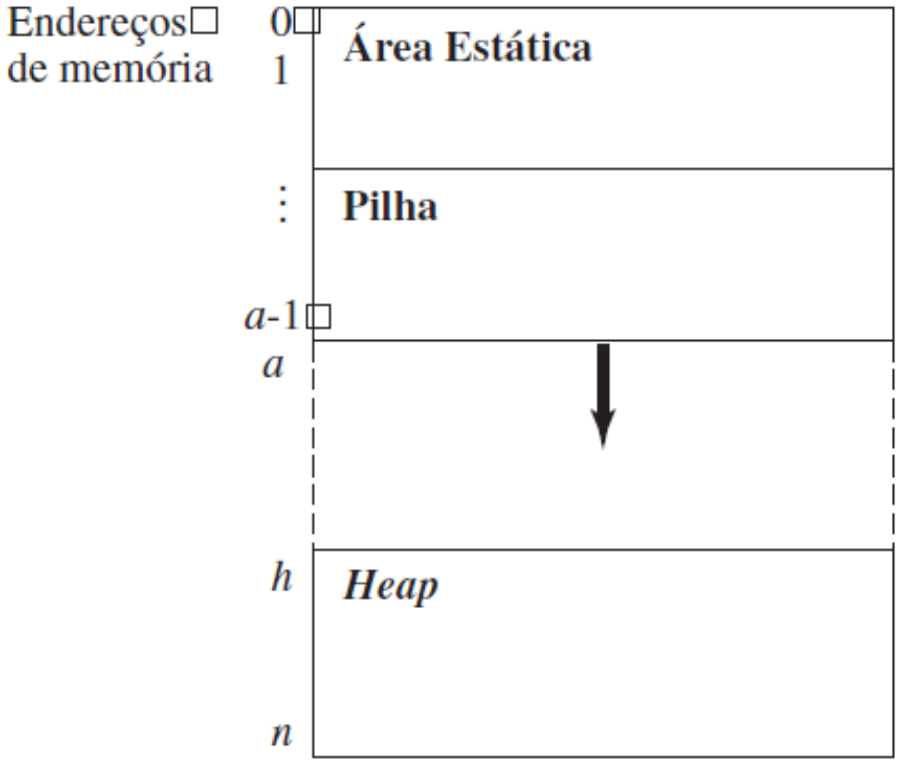
- A **memória estática** contém valores que são conhecidos antes do tempo de execução e permanecem constantes por toda a vida do programa em execução (variáveis globais).

- A **pilha de tempo de execução** é onde são armazenadas as variáveis declaradas localmente (dentro da função) e a ligação parâmetro-argumento (chamada de funções).

- A **memória heap** contém valores que são alocados e estruturados dinamicamente, como vetores e matrizes dinâmicas, objetos e diversas estruturas dinâmicas de dados como listas encadeadas.

Memória heap – Capítulo 11 Tucker

- A seguinte relação deve ser mantida por toda a execução do programa, para evitar o surgimento de um erro de **overflow de pilha** (stack overflow).



$$0 \leq a \leq h \leq n$$

Memória heap – Capítulo 11 Tucker

- As funções `calloc()` e `free()` de gerenciamento de memória do heap permitem ao programador na linguagem C obter e liberar um bloco contíguo de **`bytes`** na memória *heap*.
- Os blocos são referenciados usando variáveis ponteiros, cujos valores são endereços.
- A função `calloc()` retorna o endereço do primeiro **`byte`** de um bloco contíguo de `n_elementos*tamanho_elemento`

```
void *calloc(int n_elementos, int tamanho_elemento );
```

- A função retorna um ponteiro `void*,` se quisermos alocar um bloco de endereços para inteiros, fazemos um *casting* colocamos `int*` antes da chamada da função.

Memória heap – Capítulo 11 Tucker

```
int qtd, *vet;
printf("\nEntre com a quantidade de números: ");
scanf("%d", &qtd);
vet = (int*)calloc(qtd, sizeof(int));
if( vet==NULL)
    return 1;
. . . . . .
free(vet);
```

- Se a função **`calloc`** não consegue alocar o espaço, ele devolve `NULL`. Convém verificar essa possibilidade antes de prosseguir
- A função `free()` desfaz o efeito de `calloc()`, já que retorna desaloca o bloco contíguo de bytes que pertençam a um determinado endereço.

Exercícios

1) Considere o trecho de código abaixo qual é a saída, tente testar o código ?

```
int *v = (int *)calloc(4, sizeof(int));
v[0] = 10;
v[1] = 20;
v[2] = 30;
v[3] = 40;
printf(“%d”,*v);
printf(“%d”,*(v+1));
```

Exercícios

2) O que há de errado com o seguinte fragmento de código?

```
int *v;
v = (int*) calloc (100, sizeof (int));
v[0] = 999;
free (v+1);
```

3) PERGUNTA: Posso alocar um vetor estaticamente com número não-constante de elementos? Por exemplo, posso dizer

```
int v[n];
```

se o valor de `n` só se torna conhecido durante a execução do programa?

RESPOSTA: Não é uma boa ideia.

Exercícios

4) Escreva um programa que aloque dinamicamente um vetor v e o preencha com v[i] = 100*i, sendo que o número de elementos do vetor é lido do teclado.
5) Mude o programa anterior, escrevendo funções separadas para a) alocar o vetor e preenchê-lo com zeros; b) preencher o vetor; e c) imprimir o vetor.
6) Escreva um programa que aloque dinamicamente uma matriz m e a preencha com m[i][j] = i+j, sendo que o número de linhas e colunas são lidos do teclado.
7) Mude o programa anterior, escrevendo funções separadas para a) alocar a matriz; b) preencher a matriz; e c) imprimir a matriz

Atividade - calculadora de fração

Em matemática, uma **fração** é um modo de expressar uma **quantidade** a partir de uma **razão** de dois **números inteiros**. De modo simples, pode-se dizer que **uma fração**, representada de modo genérico como **a/b**, designa o inteiro **a dividido em b** partes iguais. Neste caso, **a** corresponde ao **numerador**, enquanto **b** corresponde ao **denominador**, que não pode ser igual a zero.

Operações entre frações:

Soma de fração: (a/b)+(c/d)=((a.d+c.b) / b.d)

Multiplicação de fração: (a/b) * (c/d)= ((a*c) / (b*d))

Divisão de fração: (a/b) / (c/d) = (a/b)*(d/c) = (a*d) / (b*c)

Igualdade: (a/b)== (c/d) se a*d == b*c

Atividade calculadora de fração

- Implemente uma calculadora de fração utilizando a linguagem C no paradigma imperativo, use os conceitos vistos nas aulas, ou seja, para representar uma fração você pode definir um novo tipo fração com `typedef struct`.
- As operações definidas para fração podem ser representadas como funções que recebem como parâmetro duas frações e retorna uma nova fração, isso para as operações de soma, multiplicação e divisão. A operação de igualdade pode retornar verdadeiro ou falso na linguagem C.
- Para testar as implementações das funções escreva uma função principal (`main()`) onde é feita a entrada (`scanf`) e saída (`printf`) dos dados.

Fim