Programação Estruturada
Prof. Rodrigo Hausen
http://progest.compscinet.org

Organização e Gerenciamento de Memória

AULA PASSADA - vetores ou arrays

Declaração de um vetor (array) em C:

```
tipo nome[tamanho];
```

Exemplos:

```
int numeros[100];

double notas[40];

char nomeDoMes[10];
```

Obs.: padrão C99 permite definir vetores com tamanho dependente de variável

```
int n = 40;
double notas[n];
```

VETORES

Acesso a elemento:

- nomeDoArray[posicao]

Obs.: em C, a posição do primeiro elemento de um array é sempre **0** (zero)

Inicialização com atribuição:

- int numeros[] = { 7, 9, 5, 2, 1 };

VETORES e ORGANIZAÇÃO DA MEMÓRIA

Exemplo: crie o programa testevetor.c com o seguinte conteúdo. Compile com o gcc e execute-o **na máquina virtual**.

```
#include <stdio.h>

int main(void) {
  int a[] = { 18, 33, 99 };
  int b[] = { 42, 21 };
  int i;

  for (i=0; i<7; ++i) {
    printf("b[%d] = %d\n", i, b[i]);
  }
}
```

VETORES e ORGANIZAÇÃO DA MEMÓRIA

```
int a[] = { 18, 33, 99 };
int b[] = { 42, 21 };
```

```
b[0] = 42
b[1] = 21
b[2] = 4195344
b[3] = 0
b[4] = 18
b[5] = 33
b[6] = 99
```

VETORES e ORGANIZAÇÃO DA MEMÓRIA

```
int a[] = { 18, 33, 99 };
int b[] = { 42, 21 };
```

Moral da história:

- C não verifica se estamos acessando posições fora dos limites do vetor!

- elementos de um vetor ocupam posições contíguas na memória

```
b[0] = 42
b[1] = 21
b[2] = 4195344
b[3] = 0
b[4] = 18
b[5] = 33
b[6] = 99
```

VETORES e ORGANIZAÇÃO DA MEMÓRIA

Elementos de vetores declarados com

tipo *nomeVet*[tamanho];

ou com

tipo *nomeVet*[] = { elmt0, elmt1, … };

ocupam **posições subsequentes** da memória do computador, geralmente na área denominada **pilha.**

A variável *nomeVet* apenas armazena a **posição (endereço)** do primeiro elemento.

Vamos alterar o programa para imprimir os endereços dos elementos do vetor.

VETORES e ORGANIZAÇÃO DA MEMÓRIA

```
// testevetor2.c

#include <stdio.h>

int main(void) {
  int a[] = { 18, 33, 99 };
  int b[] = { 42, 21 };
  int i;

  printf("a em %p e b em %p\n", a, b);

  for (i=0; i<7; ++i) {
    printf("%p: b[%d]=%d\n",&b[i],i,b[i]);
  }
}
```

VETORES e ORGANIZAÇÃO DA MEMÓRIA

```
#include <stdio.h>

int main(void) {
  int a[] = { 18, 33, 99 };
  int b[] = { 42, 21 };
  int i;

  printf("a em %p e b em %p\n", a, b);

  for (i=0; i<7; ++i) {
    printf("%p: b[%d]=%d\n",&b[i],i,b[i]);
  }
}
```

formato %p imprime endereço em hexadecimal

operador unário & obtém endereço de variável

VETORES e ORGANIZAÇÃO DA MEMÓRIA

```
int a[] = { 18, 33, 99 };
int b[] = { 42, 21 };
```

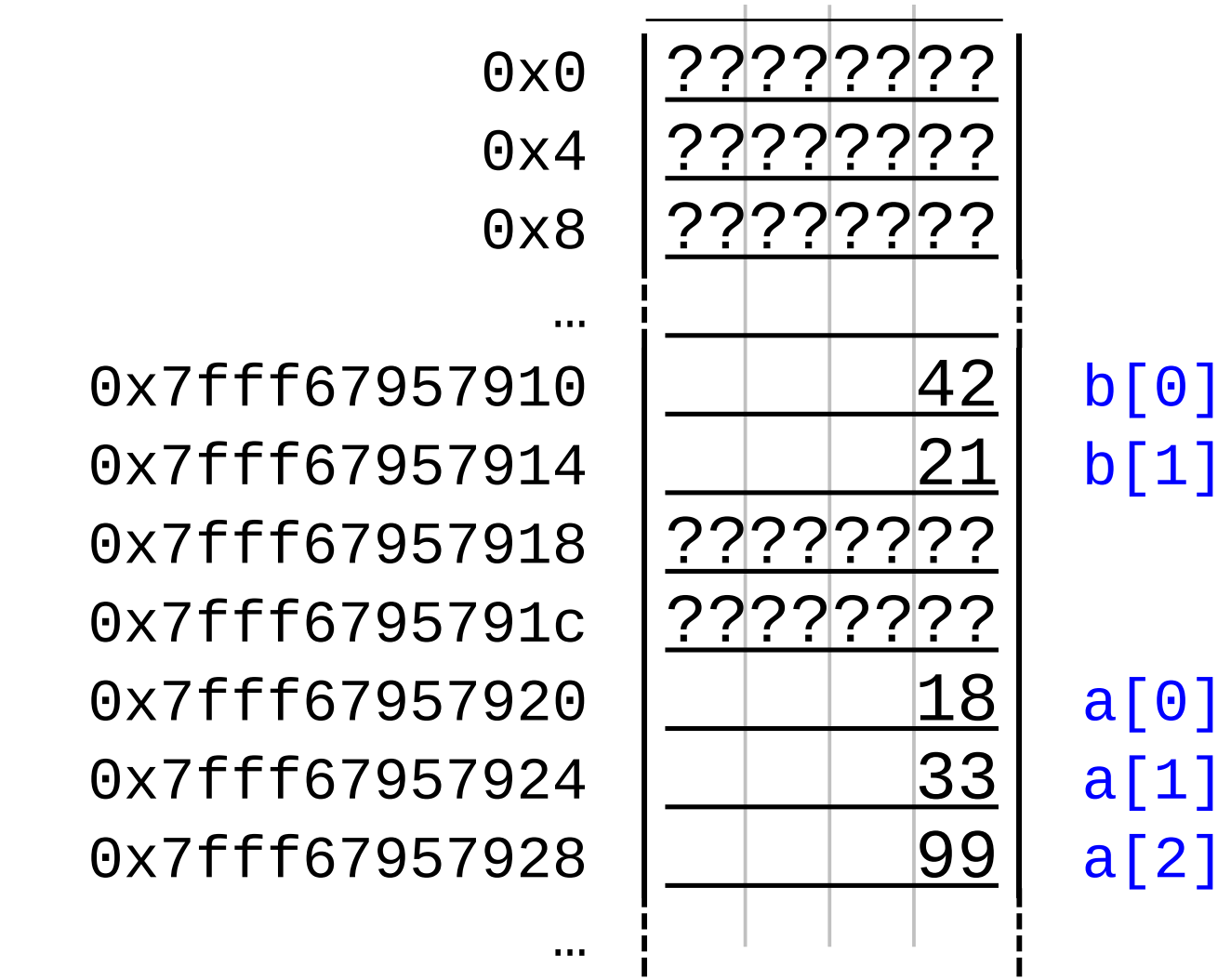
a em 0x7fff67957920 e b em 0x7fff67957910

```
0x7fff67957910: b[0] = 42
0x7fff67957914: b[1] = 21
0x7fff67957918: b[2] = 4195344
0x7fff6795791c: b[3] = 0
0x7fff67957920: b[4] = 18
0x7fff67957924: b[5] = 33
0x7fff67957928: b[6] = 99
```

Obs.: **não posso assumir** que os endereços serão os mesmos em outros computadores, ou com outros compiladores, ou mesmo em outras execuções!

VETORES e ORGANIZAÇÃO DA MEMÓRIA

A memória do computador pode ser vista como um conjunto de “**caixinhas.**” Cada uma pode ser acessada por meio de seu **endereço**.



Obs.: lembre que os endereços são específicos desta máquina, compilador e execução

Se você executar o mesmo programa em outro computador, os resultados podem ser diferentes!

Só é garantido que um vetor ocupe área de memória contígua!

VETORES e PONTEIROS

Vimos que vetor é apenas uma referência para uma posição de memória.

Podemos criar variáveis que, **explicita-mente**, armazenam uma referência para outra posição de memória.

Tais variáveis são chamadas **ponteiros**.

Declaração de um ponteiro:

```
tipo *nome;
```

VETORES e PONTEIROS

Para que uma função retorne um vetor, devemos declarar o tipo de retorno como **ponteiro**.

Exemplo: função para calcular e retornar a soma de dois vetores v, w.

Vamos fazer uma primeira tentativa. Note que o código a seguir compila, mas está **errado**!

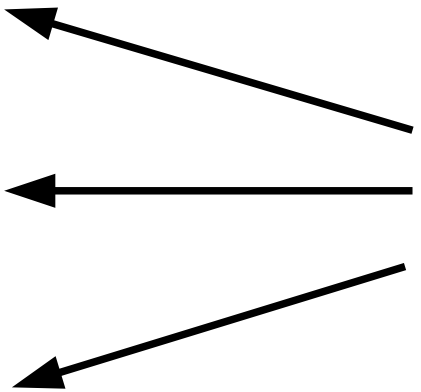
```
// somaVetores.c
// este código compila mas está errado!!

double *somaVetores(double v[],
                    double w[], int n) {
  double soma[n];
  int i;

  for (i = 0; i < n; ++i) {
    soma[i] = v[i] + w[i];
  }

  return soma;
}
```

```
[cling]$ .L somaVetores.c
 warning: address of stack memory associated with
local variable 'ret' returned [-Wreturn-stack-
address]
  return ret;
         ^~~
```

```
[cling]$ double v[] = { 1.5, 2.4, 3.1 };
[cling]$ double w[] = { 4.0, 6.3, 8.2 };
[cling]$ double *z = somaVetores(v,w,3);
[cling]$ z[0]
  (double) (... lixo ...)
[cling]$ z[1]
  (double) (... lixo ...)
[cling]$ z[2]
  (double) (... lixo ...)
```

```
[cling]$ .L somaVetores.c
 warning: address of stack memory associated with
local variable 'ret' returned [-Wreturn-stack-
address]
  return ret;
         ^~~
```

```
[cling]$ double v[] = { 1.5, 2.4, 3.1 };
[cling]$ double w[] = { 4.0, 6.3, 8.2 };
[cling]$ double *z = somaVetores(v,w,3);
[cling]$ z[0]
  (double) (... lixo ...)
[cling]$ z[1]
  (double) (... lixo ...)
[cling]$ z[2]
  (double) (... lixo ...)
```



Não deveriam ser, respectivamente, 5.5, 8.7 e 11.3?

VETORES e PONTEIROS

Relembrando: vetores declarados com

```
tipo nome[tam];
```

são geralmente declarados na área de memória chamada **pilha**.

A parte da pilha alocada a uma função não é mais válida após o término dessa função.

Precisamos alocar espaço para vetores em outra área de memória. Como fazer?

ORGANIZAÇÃO DA MEMÓRIA em C

Para a linguagem C, a memória é organizada em duas áreas principais:

- a **pilha**, região onde, para cada bloco de código, o espaço no “topo” é automaticamente reservado (alocado) para variáveis locais e, ao final do bloco, automaticamente liberado (desalocado).

- o **heap** (amontoado), onde a alocação e desalocação de memória é gerenciada manualmente pelo programador.

GERENCIAMENTO MANUAL DE MEMÓRIA

Em C, o gerenciamento manual de memória é feito por meio de funções encontradas na biblioteca **stdlib.h**

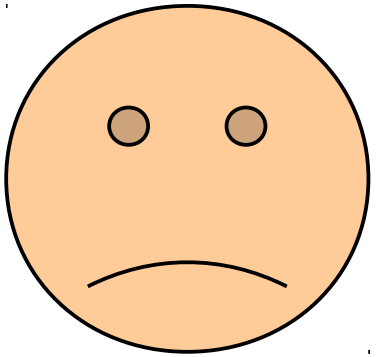
As principais são:

- **malloc**, que recebe o **número de bytes** a serem alocados e retorna o endereço inicial da região de memória alocada Ou 0 (NULL) caso não haja memória disponível.
- **free**, que recebe o endereço inicial de uma região alocada por malloc e marca a região como disponível para futuras Alocações.

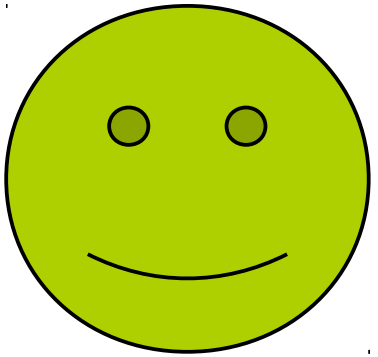
Voltando à implementação **incorreta** de somaVetores:

```
double *somaVetores(double v[],
                    double w[], int n) {
  double soma[n];
  int i;

  for (i = 0; i < n; ++i) {
    soma[i] = v[i] + w[i];
  }

  return soma;
}
```

vetor soma *alocado na* ***pilha****. Esta região de memória é desalocada automaticamente ao final da função.*



```c
#include <stdlib.h>

double *somaVetores(double v[],
                    double w[], int n) {
  double *soma = (double *)
                 malloc(n * sizeof(double));
  int i;
  if (soma == NULL) return NULL;

  for (i = 0; i < n; ++i) {
    soma[i] = v[i] + w[i];
  }

  return soma;
}
```



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

carrega biblioteca com a função `malloc`

```
double *somaVetores(double v[],
                    double w[], int n) {
  double *soma = (double *)
                 malloc(n * sizeof(double));
  int i;
  if (soma == NULL) return NULL;

  for (i = 0; i < n; ++i) {
    soma[i] = v[i] + w[i];
  }

  return soma;
}
```

```
#include <stdlib.h>

double *somaVetores(double v[],
                    double w[], int n) {
  double *soma = (double *)
                 malloc(n * sizeof(double));
  int i;
  if (soma == NULL)
    return NULL;

  for (i = 0; i < n;
    soma[i] = v[i] +
  }

  return soma;
}
```

*Calcula espaço em bytes para um vetor de **n** elementos do tipo **double***

```
#include <stdlib.h>

double *somaVetores(double v[],
                    double w[], int n) {
  double *soma = (double *)
                 malloc(n * sizeof(double));
  if (soma == NULL)
    return NULL;
  for (int i = 0; i < n; i++)
    soma[i] = v[i] + w[i];

  return soma;
}
```

O ponteiro retornado por malloc é do tipo **`void *`** (ponteiro sem tipo).
É preciso convertê-lo para o tipo de ponteiro desejado.

```
#include <stdlib.h>

double *somaVetores(double v[],
                    double w[], int n) {
  double *soma = (double *)
                 malloc(n * sizeof(double));
  int i;
  if (soma == NULL) return NULL;

  for (i = 0; i < n; ++i) {
    soma[i] = v[i] + w[i];
  }

  return s
}
```

Antes de usar, SEMPRE verifique se o ponteiro retornado por malloc é nulo! Se for nulo, trate a situação da melhor maneira possível.

CUIDADOS COM O GERENCIAMENTO DE MEMÓRIA

- para cada malloc deve existir **um, e apenas um**, free correspondente
- esquecer free ⇒ falta de memória no futuro
- “double free” = KABOOM! (a execução do seu programa é abortada imediatamente)
- **jamais** use um ponteiro após executar free nele! Fácil de falar, mas ocorre muitas vezes por descuido. Erros de uso de ponteiro após free costumam ser difíceis de depurar pois podem causar efeitos colaterais aparentemente inexplicáveis e longe da origem.

CUIDADOS COM O GERENCIAMENTO DE MEMÓRIA

- se uma função retorna um ponteiro alocado com malloc, **SEMPRE** informe na documentação da função que você deverá usar free no ponteiro retornado

```
/**
 * Soma dois vetores.
 * IMPORTANTE: o ponteiro retornado
 * deverá ser liberado com free após
 * o uso.
 */
double *somaVetores(double v[],
                    double w[], int n) {
  …
```

CUIDADOS COM O GERENCIAMENTO DE MEMÓRIA

- enquanto estiver escrevendo código que usa malloc, ou código que usa uma função que retorna ponteiro alocado desta forma, escreva **imediatamente** o free após o uso, para não esquecer dele, e depois coloque o resto do código entre a alocação e o free.

EXERCÍCIOS

Crie um arquivo **strfunc.c**, implemente e documente as seguintes funções, com atenção para todos os casos possíveis (inclusive ponteiros nulos)

- **int** comprimento(**const char** *s)
  retorna o número de caracteres da string (sem incluir o caractere nulo)
- **char** *concatena(**const char** *s1,
  const char *s2)
  retorna a concatenação de s1 com s2.
- **char** *maiusculas(**const char** *s)
  retorna a string trocando os caracteres a-z para maiúsculas

OBSERVAÇÃO

Por que o uso de const char * ?

- **int** comprimento(**const char** *s)
- **char** *concatena(**const char** *s1,
  const char *s2)
- **char** *maiusculas(**const char** *s)

Isto indica que as áreas de memória indicadas pelos parâmetros **não serão alteradas** pela função.

BOA PRÁTICA: funções que recebem ponteiros e que não alteram as regiões apontadas por eles devem declará-los como **const**.

Mais exercícios serão colocados no site.

Façam os exercícios e estudem!

Sempre testem suas implementações de funções no **cling**. Pensem em exemplos para testá-las e casos “patológicos” (strings vazias, ponteiros nulos caso sejam permitidos, parâmetros inteiros iguais a zero ou negativos, etc.)

Vou colocar no final de semana uma imagem “live” do ambiente de desenvolvimento (compilador, cling e IDE) que executa **direto de um pendrive sem precisar de instalação**.