

Alocação dinâmica de memória

Programação C

Prof. Renan Augusto Starke

Instituto Federal de Santa Catarina – IFSC
Campus Florianópolis
`renan.starke@ifsc.edu.br`

19 de agosto de 2019



INSTITUTO FEDERAL
SANTA CATARINA

Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
INSTITUTO FEDERAL DE SANTA CATARINA

Tópicos da aula

- 1 Introdução
- 2 Alocação dinâmica de memória
- 3 Exemplos
- 4 Conclusões

1 Introdução

2 Alocação dinâmica de memória

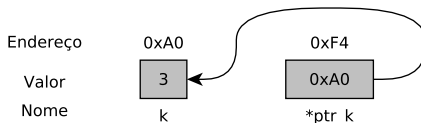
3 Exemplos

4 Conclusões

- ▶ Entender o conceito e aplicações de alocação dinâmica de memória
- ▶ Aprender funções importantes
 - alocação
 - realocação
 - liberação
- ▶ Lidar com alocação, liberação e manipulação de estruturas de dados alocadas dinamicamente

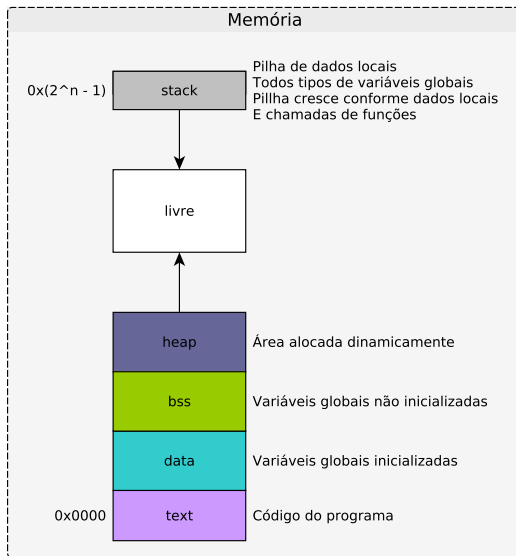
Introdução

- ▶ Declaração de variáveis:
 - um espaço de memória é reservado
 - endereço fixo
- ▶ Através de ponteiros pode-se alterar o endereçamento
 - mas se for necessário mais memória, além daquela já alocada pelo compilador?



```
// variável  
char k = 3;  
  
// ponteiro de k  
char *ptr_k = &k;
```

Introdução



1 Introdução

2 Alocação dinâmica de memória

3 Exemplos

4 Conclusões

Alocação dinâmica

Técnica onde utiliza-se a seção **heap** de dados alocando memória em **tempo de execução** através de funções pré-definidas.

Exemplos:

- ▶ Leitura de dados do disco com tamanho desconhecido e variável
- ▶ Tamanho desconhecido de um *array*
- ▶ Como fazer um programa para ordenar uma quantidade arbitrária de números?

- ▶ Define-se a capacidade máxima permitida ($N = 1000000000000$)

```
#define N 1000000000000  
  
int numeros[N];
```

Limitações?

- ▶ Define-se a capacidade máxima permitida ($N = 1000000000000$)

```
#define N 1000000000000  
  
int numeros[N];
```

Limitações?

- ▶ Limita-se a quantidade de números que pode-se armazenar e ordenar
- ▶ Desperdício de memória
- ▶ É mais interessante alocar a quantidade necessária de memória:
 - Alocação dinâmica

- ▶ O ANSI C define 4 funções para gerenciamento de memória:
 - **malloc**: aloca um quantidade especificada de memória
 - **calloc**: aloca um quantidade especificada de memória zerando todo o seu conteúdo
 - **realloc**: redimensiona um tamanho já alocado de memória
 - **free**: libera espaço alocado
- ▶ Cabeçalhos destas funções: **stdlib.h**

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

malloc

```
void * malloc(size_t size)
```

Entrada:

- ▶ **size**: tamanho, em bytes, que deseja-se alocar

Retorno da função:

- ▶ se sucesso, retorna um ponteiro para o **início da área alocada**
- ▶ se falhar, retorna **NULL**
- ▶ a área de memória alocada é sempre contínua
- ▶ **Sempre deve-se verificar o retorno de malloc**

Exemplo

```
int *numeros;  
int quantidade;  
numeros = (int*) malloc(sizeof(int) * quantidade);
```

- ▶ **(int *)**: mudança do tipo de ponteiro (*cast*)
- ▶ **sizeof(int)**: tamanho em bytes de um **int**

Exemplo

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

int main(int argc, char ** argv) {

    int *numeros, tamanho, i;

    scanf("%d",&tamanho);

    numeros = (int*) malloc(sizeof(int) * tamanho);

    if (numeros == NULL) {
        perror("main:");
        exit(1);
    }

    for (i=0; i<tamanho; i++)
        scanf("%d",&numeros[i]);

    /* ... */
}
```

free

```
void free(void *ptr)
```

Entrada:

- ▶ **ptr**: ponteiro da memória previamente alocada

Observações:

- ▶ se *ptr* for *NULL*, nenhuma operação é realizada
- ▶ se **free(ptr)** for chamada mais de **uma** vez:
 - comportamento imprevisível
 - programa pode ser abortado
- ▶ após **free**, dados não podem mais ser acessados seguramente
- ▶ **sempre libere a memória utilizada**
- ▶ **Não acesse** dados fora da área alocada

Exemplo

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

int main(int argc, char ** argv) {

    int *numeros, tamanho, i;

    scanf("%d",&tamanho);

    numeros = (int*) malloc(sizeof(int) * tamanho);

    if (numeros == NULL) {
        perror("main:");
        exit(1);
    }

    for (i=0; i<tamanho; i++)
        scanf("%d",&numeros[i]);

    /* ... */

    free(numeros);

    return 0;
}
```


calloc

```
void *calloc(size_t count, size_t size);
```

Entrada:

- ▶ **count**: quantidade de elementos de tamanho **size** que se deseja alocar
- ▶ **size**: tamanho, em bytes, de **um** elemento que se deseja alocar

Retorno da função:

- ▶ se sucesso, retorna um ponteiro para o **início da área alocada**
- ▶ se falhar, retorna **NULL**
- ▶ a área de memória alocada é sempre contínua
- ▶ Sempre deve-se verificar o retorno de **calloc**

realloc

```
void *realloc(void *ptr, size_t size);
```

Entrada:

- ▶ **ptr**: ponteiro da memória que se deseja redimensionar
- ▶ **size**: tamanho, em bytes, do tamanho **total** redimensionado

Retorno da função:

- ▶ se sucesso, retorna um ponteiro para o **início da área alocada**
- ▶ se falhar, retorna **NULL**
- ▶ a área de memória alocada é sempre contínua
- ▶ nova área pode começar em endereço diferente do original
- ▶ **Sempre deve-se verificar o retorno de realloc**

1 Introdução

2 Alocação dinâmica de memória

3 Exemplos

4 Conclusões

Encontrar e salvar números ímpares de um vetor

```
int *impares(int *a, int tamanho, int *qtdImpares){
    int i, j = 0;
    int qtdI = 0;
    int *impares;

    for (i = 0; i < tamanho; i++)
        if (a[i] % 2 == 1)
            qtdI++;

    impares=(int*)malloc(sizeof(int)*qtdI);

    if (impares == NULL) {
        perror("impares:");
        exit(-1);
    }

    for (i = 0; i < tamanho; i++)
        if (a[i] % 2 == 1)
            impares[j++]=a[i];

    *qtdImpares = qtdI;

    return impares;
}
```

Encontrar e salvar números ímpares de um vetor

```
int main(int argc, char ** argv) {
    int *numeros,*imp,qtdl, tamanho, i;

    scanf("%d",&tamanho);

    numeros = (int*) malloc(sizeof(int) * tamanho);

    if (numeros == NULL) {
        perror("main:");
        exit(-1);
    }

    for (i=0; i<tamanho; i++)
        scanf("%d",&numeros[i]);

    imp = impares(numeros, tamanho, &qtdl);

    free(numeros);
    free(imp);

    return 0;
}
```

1 Introdução

2 Alocação dinâmica de memória

3 Exemplos

4 Conclusões

- ▶ Alocação dinâmica de memória é uma ferramenta poderosa para desenvolver programas em ANSI C
- ▶ Deve ser usada com cuidado
- ▶ Não perder referências a áreas de memórias alocadas
- ▶ Liberar memória que não é mais usada