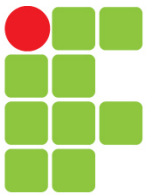
Alocação dinâmica de memória

Programação de computadores II

Prof. Renan Augusto Starke

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

29 de agosto de 2016



INSTITUTO FEDERAL
SANTA CATARINA

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

Tópicos da aula

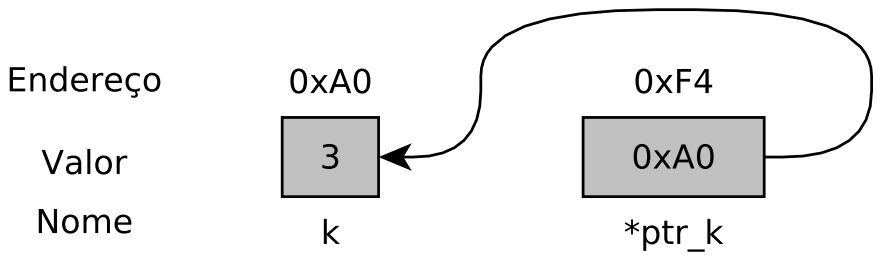
Tópico

Objetivos

- 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

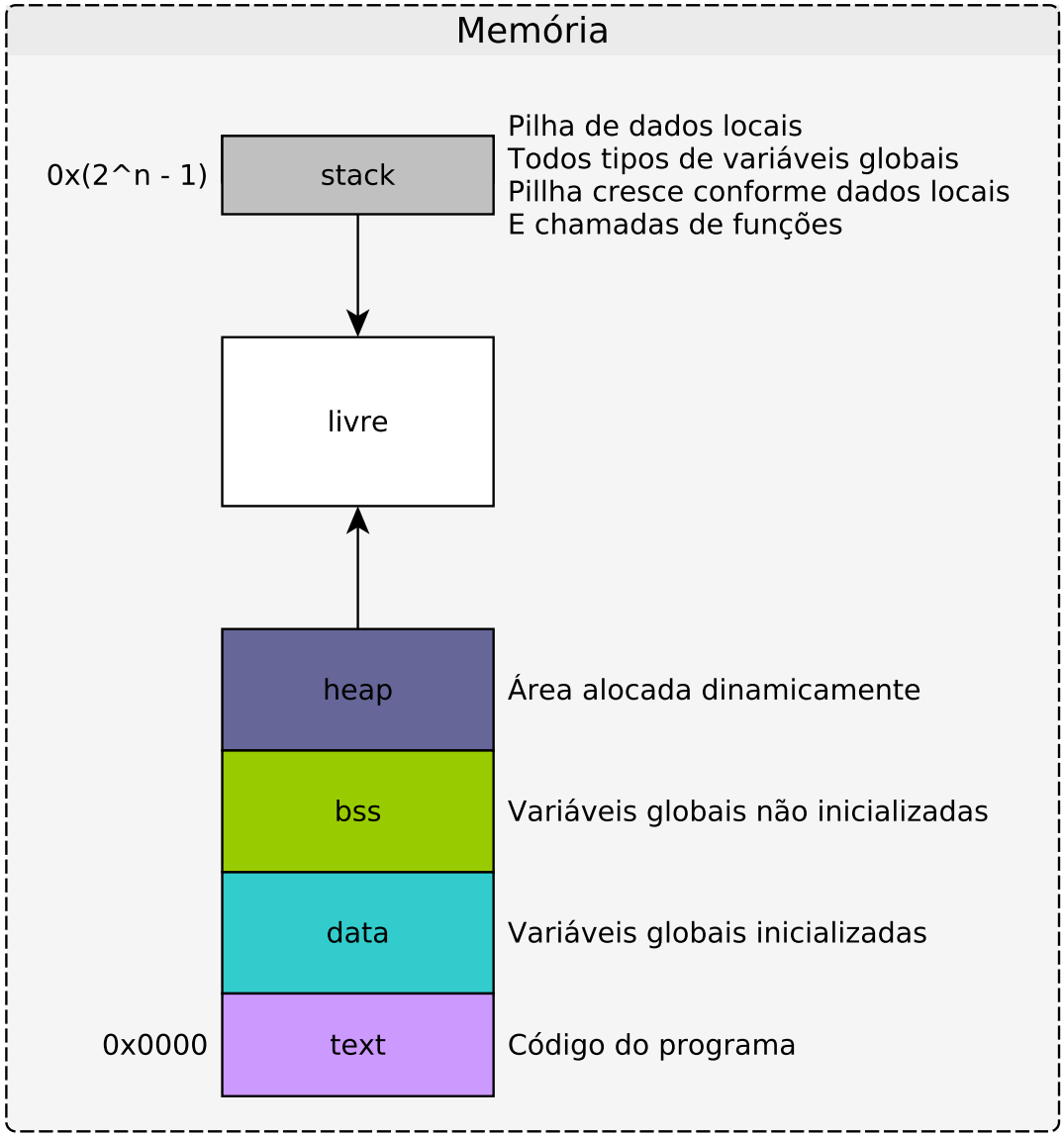
Introdução

- 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?




```
1  // variável
2  char k = 3;
3
4  // ponteiro de k
5  char *ptr_k = &k;
```

Introdução


Memória
0x(2^n - 1)
stack
Pilha de dados locais
Todos tipos de variáveis globais
Pillha cresce conforme dados locais
E chamadas de funções
livre
heap
Área alocada dinamicamente
bss
Variáveis globais não inicializadas
data
Variáveis globais inicializadas
0x0000
text
Código do programa

Tópico

Alocação dinâmica

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?

Sem alocação dinâmica

- Define-se a capacidade máxima permitida ($N = 100000000000$)

```
1 #define N 100000000000
2
3 int numeros[N];
```

Limitações?

Sem alocação dinâmica

- Define-se a capacidade máxima permitida ($N = 100000000000$)

```
1 #define N 100000000000
2
3 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

Gerenciamento de memória

- 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**

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

Alocando

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

```
1 int *numeros;
2 int quantidade;
3 numeros = (int*) malloc(sizeof(int) * quantidade);
```

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

Exemplo

```c
#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]);

  /*...*/
```

Liberando

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

```c
#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;

}
```

Alocando e zerando

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

Realocando

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

Tópico

Encontrar e salvar números ímpares de um vetor

```
1  int *impares(int *a, int tamanho, int *qtdImpares){
2    int i,j = 0;
3    int qtdI = 0;
4    int *impares;
5
6    for (i = 0; i < tamanho; i++)
7      if (a[i] % 2 == 1)
8        qtdI++;
9
10   impares=(int*)malloc(sizeof(int)*qtdI);
11
12   if (impares == NULL) {
13     perror("impares:");
14     exit(-1);
15   }
16
17   for (i = 0; i < tamanho; i++)
18     if (a[i] % 2 == 1)
19       impares[j++]=a[i];
20
21   *qtdImpares = qtdI;
22
23   return impares;
24 }
```

Encontrar e salvar números ímpares de um vetor

```
1  int main(int argc, char ** argv) {
2    int *numeros,*imp,qtdl, tamanho, i;
3
4    scanf("%d",&tamanho);
5
6    numeros = (int*) malloc(sizeof(int) * tamanho);
7
8    if (numeros == NULL)  {
9      perror("main:");
10     exit(-1);
11   }
12
13   for (i=0; i<tamanho; i++)
14     scanf("%d",&numeros[i]);
15
16   imp = impares(numeros, tamanho, &qtdl);
17
18   free(numeros);
19   free(imp);
20
21   return 0;
22 }
```

Tópico

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