

Universidade Federal de Viçosa
Campus de Florestal

Algoritmos e Estruturas de Dados I (CCF 211)

Alocação Dinâmica de Memória

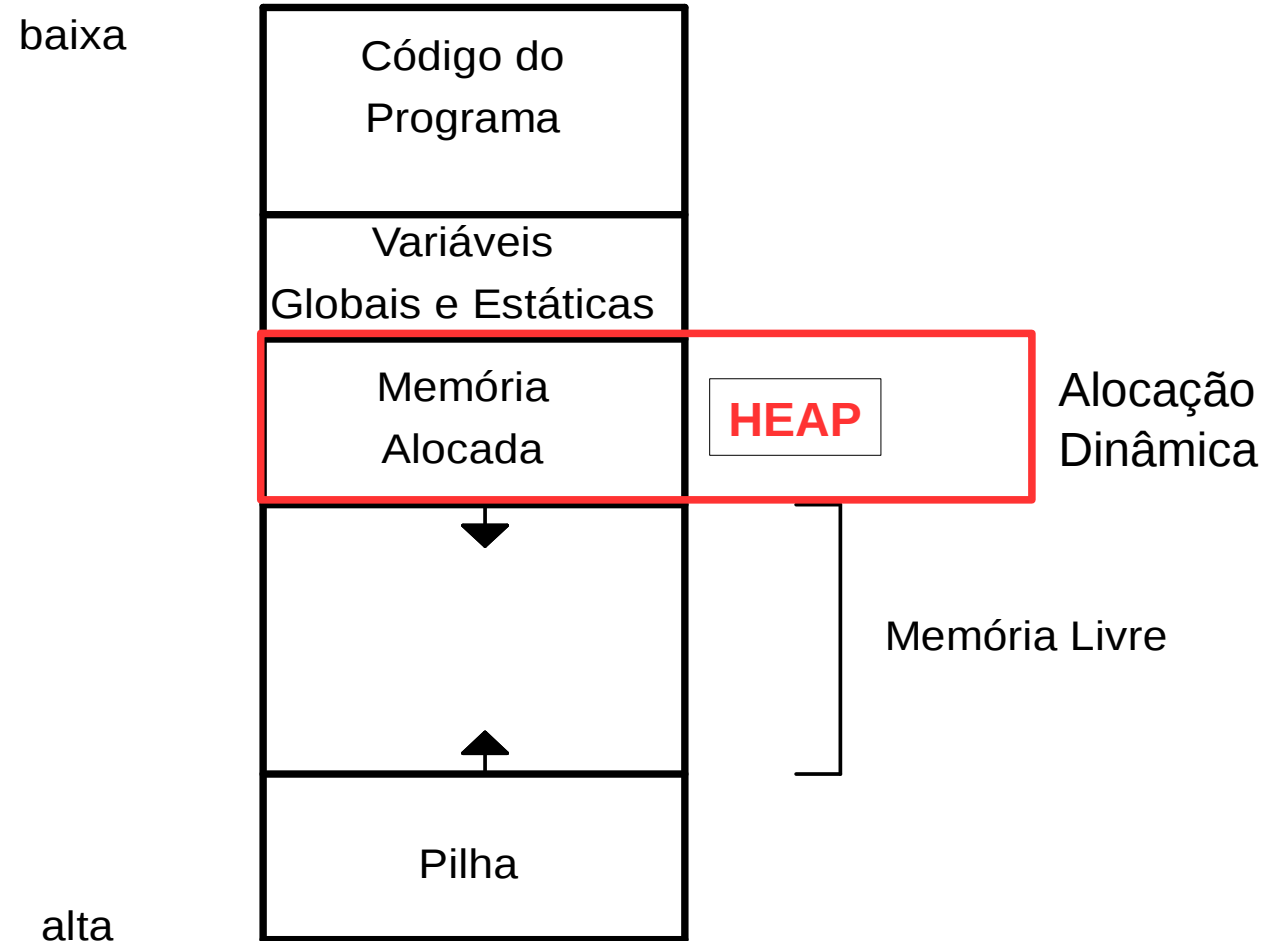
Profa. Thais R. M. Braga Silva
<thais.braga@ufv.br>



Alocação Estática x Dinâmica

- C: dois tipos de alocação de memória: **Estática e Dinâmica**
 - Na alocação estática, o espaço para as variáveis é reservado no início da execução, não podendo ser alterado depois
 - `int a; int b[20];`
 - Na alocação dinâmica, o espaço para as variáveis pode ser alocado dinamicamente durante a execução do programa

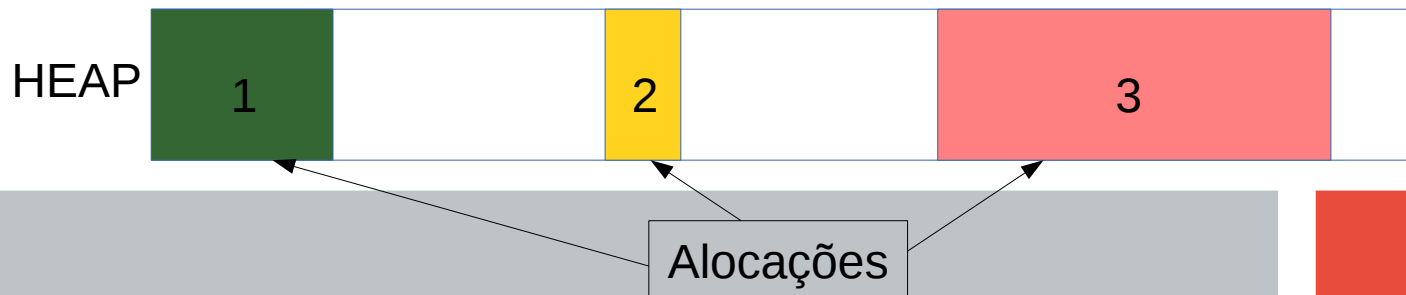
Esquema de Memória



Esquema da memória do sistema

Alocação Dinâmica

- A memória alocada dinamicamente faz parte de uma área de memória chamada **heap**
 - Basicamente, o programa aloca e desaloca porções de memória do heap durante a execução
 - Os bytes de uma mesma alocação estarão contíguos no heap
 - As sucessivas alocações no heap não necessariamente ocorrem de forma contígua no mesmo



Alocação Dinâmica

- O endereço de uma área alocada no heap deve ficar armazenado em uma variável do tipo ponteiro (estaticamente alocada)
 - É através deste ponteiro que a área dinamicamente alocada poderá ser manipulada
 - O ponteiro tem que ser de um tipo de dados específico, que pode ser primitivo ou estruturado (ex: `int*`, `double*`, `TItem*`)

Liberação de Memória

- A memória deve ser liberada após o término de seu uso
- A liberação deve ser feita por quem fez a alocação:
 - ❑ Estática: compilador
 - ❑ Dinâmica: programador

O comando malloc

- A principal função de alocação dinâmica em C é o *void* malloc(int nBytes) <stdlib.h>*
 - Recebe o número de bytes a serem alocados
 - Retorna um **ponteiro genérico (void*)** para a área de memória alocada
 - Não inicializa posições de memória alocadas
 - É necessário fazer sempre uma *conversão de tipos (typecast)* para o tipo de dados a ser armazenado na área de memória reservada
 - Utilizar sempre o operador **sizeof** para calcular o número de bytes a serem reservados

O comando malloc

- A principal função de alocação dinâmica em C é o *void* malloc(int nBytes)*

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

```
int main(){
```

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

```
    *p = 2;
```

```
    free(p);
```

```
}
```

typecast

**Ponteiro para
manipular área
alocada no heap**

nBytes

O comando malloc

- A principal função de alocação dinâmica em C é o *void* malloc(int nBytes)* *<stdlib.h>*
 - É sempre necessário guardar o valor de retorno da função malloc, já que ele é o ponteiro com o qual será acessada a memória alocada (se perdido, área estará inacessível)
 - Conversão explícita de tipos converte um valor de um tipo para outro e é feito através do comando (*<tipo>*)
 - É sempre necessária pois o valor retornado (ponteiro para void) não pode ser derreferenciado
 - *sizeof(<tipo>)* é operador que retorna número de bytes necessários para o *<tipo>*

Os comandos malloc e free

- A principal função de alocação dinâmica em C é o *void* malloc(int nBytes) <stdlib.h>*
 - O comando *free(<ptr>)* desaloca a área do heap apontada por *<ptr>*, tornando este ponteiro nulo
 - Deve sempre ser chamado pelo programador para liberar memória que já não é mais necessária (risco de esgotamento de memória)

O comando calloc

- O *void* calloc(int n, int bytes)* é igual ao *void* malloc(int nBytes)*, mas..
 - Recebe dois parâmetros: número de itens e tamanho de cada item (utilizar também o operador sizeof)
 - Inicializa bits da área de memória com 0
 - É mais “cara” e demorada do que o malloc

O comando realloc

- O *void* realloc(void* ptr, int nBytes)* é utilizado para realocar um conjunto de dados em um espaço maior/menor de memória
 - Recebe ponteiro para a área de memória onde estão os dados e o novo número de Bytes desejado (número total e não quanto a mais)
 - Pode manter a mesma área (maior ou menor) ou alocar nova em local diferente da memória. Os dados serão preservados, até onde couberem na área realocada.
 - Espaço de sobra possui conteúdo indeterminado.

O comando realloc

- *O void* realloc(void* ptr, int nBytes)*
 - Novo endereço de memória é retornado pelo comando e deve ser armazenado em um ponteiro
 - Caso a realocação ocorra com sucesso, o ponteiro passado como parâmetro (ptr) não fica nulo, mas não poderá mais ser utilizado
 - Caso a realocação falhe, o comando retorna NULL mas o ponteiro passado como parâmetro (ptr) é preservado e pode continuar a ser utilizado
 - Cuidado: não funciona se você colocar o resultado da realocação (NULL) no mesmo ponteiro

Os comandos malloc, calloc e realloc

- Operação de alocação (malloc, calloc ou realloc) pode falhar por falta de espaço no heap
 - Função retorna valor NULL
 - Usar condição após alocação para verificar se resultado foi NULL ou não

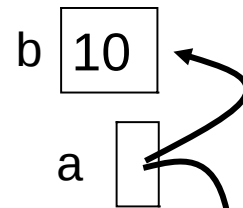
Os comandos malloc, calloc e realloc

- Após alocação dinâmica, espaço de memória pode manipulado:
 - Diretamente pelo ponteiro, usando a sintaxe de ponteiro
 - Como um vetor, usando a sintaxe de indexação de vetores. Índices podem ser utilizados normalmente para acessar os valores armazenados no espaço de memória
 - Ex: seja p um ponteiro com endereço para o heap. É possível acessar a área com o comando *p ou p[0]
 - Caso haja mais de um item *(p+i) ou p[i]

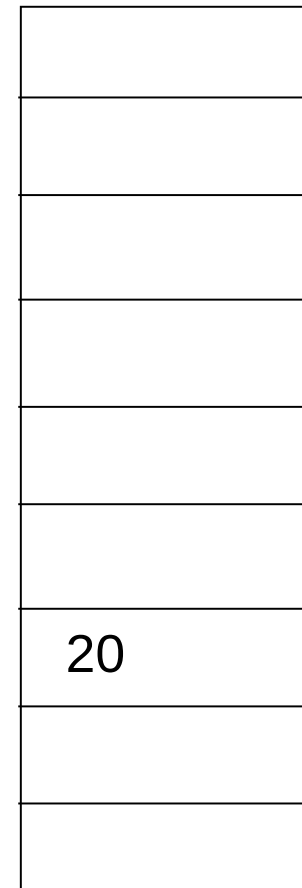
Alocação Estática e Dinâmica

```
int *a, b;  
...  
b = 10;  
a = (int *) malloc(sizeof(int));  
*a = 20;  
a = &b;
```

**Alocação
Estática**



Heap



Erros Comuns

- **Esquecer de alocar memória e tentar acessar o conteúdo da variável**
- Copiar o valor do apontador ao invés do valor da variável apontada
- Esquecer de desalocar memória
- Tentar acessar o conteúdo da variável depois de desalocá-la

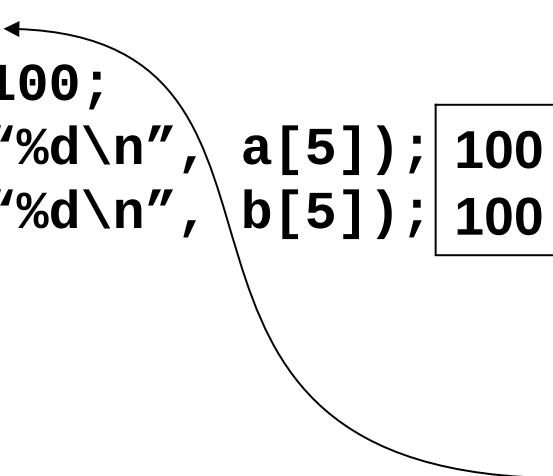
Outro Exemplo

```
double a;  
double *p;  
a = 3.14;  
printf("%lf\n", a);  
p = &a;  
*p = 2.718;  
printf("%lf\n", a);  
a = 5.;  
printf("%lf\n", *p);  
p = NULL;  
p = (double *)malloc(sizeof(double));  
*p = 20.;  
printf("%lf\n", *p);  
printf("%lf\n", a);  
free(p);  
printf("%lf\n", *p);
```

Ponteiros e alocação de memória

- Em C, ponteiros podem ser usados para alocação estática e dinâmica de memória
- Portanto pode-se fazer coisas como:

```
int a[10], *b;  
b = a;  
b[5] = 100;  
printf("%d\n", a[5]); 100  
printf("%d\n", b[5]); 100
```



```
int a[10], *b;  
b = (int *) malloc(10*sizeof(int));  
b[5] = 100;  
printf("%d\n", a[5]); 42657  
printf("%d\n", b[5]); 100
```

Obs. Não se pode fazer $a = b$
no exemplo acima.

Você pode atribuir um vetor a um
ponteiro, mas **não** um ponteiro
a um vetor (como no caso de $a = b$).

Alocação Dinâmica para Tipos Estruturados

- Funciona da mesma forma como usado para tipos primitivos

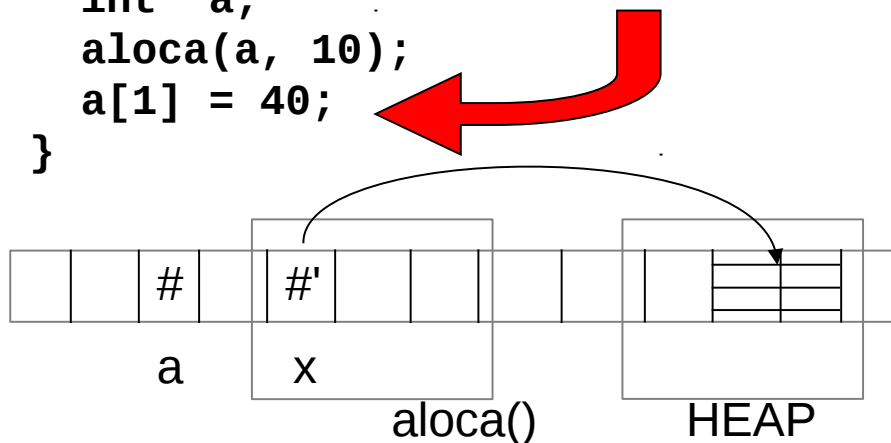
```
Typedef struct {  
    int idade;  
    double salario;  
} TRegistro;  
  
TRegistro *a;  
...  
a = (TRegistro *) malloc(sizeof(TRegistro))  
a->idade = 30;    /* (*a).idade = 30 */  
a->salario = 80;
```

Alocar memória no heap dentro de um procedimento

- Não basta passar a variável (apontador) como referência, pois não há passagem por referência na linguagem C

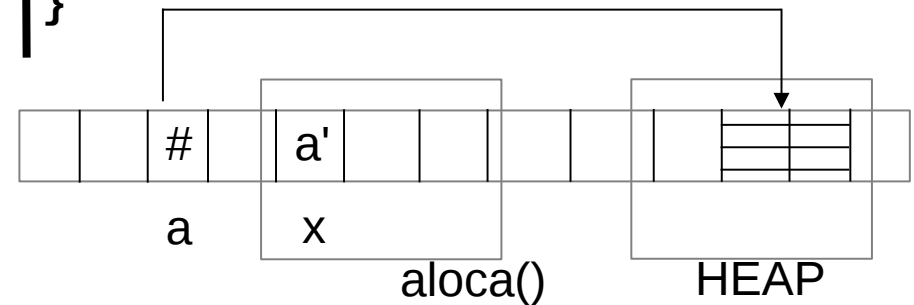
```
void aloca(int *x, int n)
{
    x=(int *)malloc(n*sizeof(int));
    x[0] = 20;
}
int main()
{
    int *a;
    aloca(a, 10);
    a[1] = 40;
}
```

Error!
Access Violation!



```
void aloca(int **x, int n)
{
    (*x)=(int *)malloc(n*sizeof(int));
    (*x)[0] = 20;
}
int main()
{
    int *a;
    aloca(&a, 10);
    a[1] = 40;
}
```

OK



Exercício 1

- Criar um tipo que é uma estrutura que represente uma pessoa, contendo nome, data de nascimento e CPF.
- Criar uma variável que é um ponteiro para esta estrutura (no programa principal)
- Criar uma função que recebe este ponteiro e preenche os dados da estrutura
- Criar uma função que recebe este ponteiro e imprime os dados da estrutura
- Fazer a chamada a estas funções na função principal

Exercício 2

1. Faça um programa que leia um valor n , crie dinamicamente um vetor de n elementos e passe esse vetor para uma função que vai ler os elementos desse vetor.
2. Declare um TipoRegistro, com campos a inteiro e b que é um apontador para char. No seu programa crie dinamicamente uma variável do TipoRegistro e atribua os valores 10 e 'x' aos seus campos.

Respostas (2.1)

```
void LeVetor(int *a, int n){
```

```
    int i;
```

```
    for(i=0; i<n; i++)
```

```
        scanf("%d",&a[i]);
```

```
}
```

Apesar do conteúdo ser modificado
Não é necessário passar por
referência pois todo vetor já
é um apontador...

```
int main(int argc, char *argv[]) {
```

```
    int *v, n, i;
```

```
    scanf("%d",&n);
```

```
    v = (int *) malloc(n*sizeof(int));
```

```
    LeVetor(v,n);
```

```
    for(i=0; i<n; i++)
```

```
        printf("%d\n",v[i]);
```

```
}
```


Respostas (2.2)

```
typedef struct {
    int a;
    char *b;
} TRegistro;
int main(int argc, char *argv[])
{
    TRegistro *reg;
    reg = (TRegistro *) malloc(sizeof(TRegistro));
    reg->a = 10;
    reg->b = (char *) malloc(sizeof(char));
    *(reg->b) = 'x';
    printf("%d %c", reg->a, *(reg->b));
}
```

É necessário alocar espaço para o registro e para o campo b.
*(reg->b) representa o conteúdo da variável apontada por reg->b