

Aula 1: Alocação Dinâmica, Ponteiros, Structs e Introdução a Listas

Professor(a): Virgínia Fernandes Mota

<http://www.dcc.ufmg.br/~virginiaferm>

ALGORITMOS E ESTRUTURAS DE DADOS - SETOR DE INFORMÁTICA



Alocação Dinâmica e Ponteiros

- A Alocação Dinâmica é o processo que aloca memória em tempo de execução
- Ela é utilizada quando não se sabe ao certo quanto de memória será necessário para o armazenamento das informações, podendo ser determinada em tempo de execução conforme a necessidade do programa.
- Dessa forma **evita-se o desperdício de memória**.

- No padrão C ANSI existem 4 funções para alocações dinâmicas pertencentes à biblioteca *stdlib.h*.
- São elas: **malloc()**, **calloc()**, **realloc()** e **free()**.
- Existem ainda outras que não serão abordadas nesta aula, pois não são funções muito utilizadas.

- A alocação dinâmica é muito utilizada em problemas de estrutura de dados, por exemplo, listas encadeadas, pilhas, filas, arvores binárias e grafos.
- **malloc()** e **calloc()**: são responsáveis por alocar memória;
- **realloc()**: responsável por realocar a memória;
- **free()**: responsável por liberar a memória alocada.

A função `malloc()`

```
1 tipo *variavel;  
2 variavel = (tipo *) malloc (tamanho * sizeof(tipo));
```

- Esta função recebe como parâmetro "tamanho" que é o número de bytes de memória que se deseja alocar.
- O interessante é que esta função retorna um **ponteiro** do tipo *void* podendo assim ser atribuído a qualquer tipo de *ponteiro*.
- PONTEIRO?!?!?!?!?

- O ponteiro nada mais é do que uma variável que guarda o endereço de uma outra variável.
- Declaração:
tipo *nome;
- Como atribuir valor ao ponteiro declarado?
***nome = valor;**
- Lembrando: Quando usamos uma passagem de parâmetros por referência estamos utilizando ponteiros!!
 - Usamos o operador & para retornar o endereço de memória que está localizado o valor da variável contida no ponteiro.
 - Usamos o operador * para retornar o valor da variável (conteúdo) que está localizada no ponteiro.
- Podem existir ainda ponteiros de ponteiros!
tipo **nome;

Motivação

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int num_elementos, vet[99] , i;
5     printf("Digite a quantidade de elementos que deseja: ");
6     scanf("%d", &num_elementos);
7     for ( i = 0; i < num_elementos; i++){
8         printf("\n Digite o elemento da posição %d: ", i);
9         scanf("%d", &vet[i]);
10    }
11    return 0;
12 }
```

- E se o usuário colocar apenas 3 elementos no vetor?
Desperdício!
- E se o usuário colocar mais de 100 elementos no vetor?
Estouro de memória!


```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int num_elementos, i;
6     int *vet; //utilizando um ponteiro
7
8     printf("Digite a quantidade de elementos que deseja: ");
9     scanf("%d", &num_elementos);
10
11     //Alocando apenas o espaço necessário
12     vet = (int *) malloc (num_elementos * sizeof(int));
13     for ( i = 0; i < num_elementos; i++){
14         printf("\n Digite o elemento da posição %d: ", i);
15         scanf("%d", &vet[i]);
16     }
17     return 0;
18 }
```

- Agora o usuário pode digitar o tamanho do vetor que desejar e não haverá desperdício de memória!
- Mas ainda temos um problema: Sempre que alocamos dinamicamente a memória precisamos obrigatoriamente liberar essa memória.
 - Na alocação estática isso é feito automaticamente.

A função `free()`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int num_elementos, i;
6     int *vet; //utilizando um ponteiro
7
8     printf("Digite a quantidade de elementos que deseja: ");
9     scanf("%d", &num_elementos);
10
11     //Alocando apenas o espaço necessário
12     vet = (int *) malloc (num_elementos * sizeof(int));
13     for ( i = 0; i < num_elementos; i++){
14         printf("\n Digite o elemento da posição %d: ", i);
15         scanf("%d", &vet[i]);
16     }
17
18     free(vet);
19
20     return 0;
21 }
22 }
```

- Libera a memória alocada.
- Deve ser chamada quando o ponteiro não for mais utilizado.

A função `calloc()`

```
1 tipo *variavel;  
2 variavel = (tipo *) calloc (tamanho, sizeof(tipo));
```

- Esta função inicia o espaço alocado com 0.
- No exemplo anterior, a alocação ficaria:
vet = (int *) calloc (num_elementos, sizeof(int));

A função **realloc()**

```
1 tipo *variavel;  
2 variavel = (tipo *) realloc (variavel, tamanho);
```

- Esta função altera o tamanho da memória anteriormente alocado.
- No exemplo anterior, a realocação para um novo_tamanho ficaria:

```
vet = (int *) realloc (vet, novo_tamanho);
```

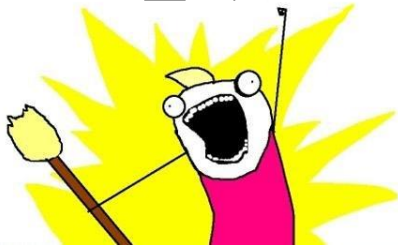
Alocação dinâmica e Subrotinas

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float* alocaVetor(int n){
5     float *v;
6     v = (float *) malloc (n * sizeof(float));
7     return v;
8 }
9 void leVetor(float *v, int n){
10     int i;
11     for (i = 0; i < n; i++)
12         scanf("%f", &v[i]);
13 }
14
15 int main(){
16     int n;
17     float *v;
18     printf("\n Digite o numero de elementos do vetor \n");
19     scanf("%d", &n);
20     v = alocaVetor(n);
21     printf("\n Digite seu vetor \n");
22     leVetor(v, n);
23     free(v);
24     return 0;
25 }
```

- Importante: Não se pode retornar vetores em função, mas pode-se retornar ponteiros!

- A Alocação Dinâmica reduz o desperdício de memória e torna o programa mais portátil.
- As funções mais utilizadas: `malloc()`, `calloc()`, `realloc()` e `free()`.

Free ALL pointers





Estruturas (Structs)

- Sintaxe para definir uma estrutura com **n** campos em C:

```
1 struct nome_estrutura {  
2     tipo1 identificador1;  
3     tipo2 identificador2;  
4     ...  
5     tipon identificadorn;  
6 };
```

- Em C, pode-se criar **m** variáveis de uma dada estrutura de duas formas:

```
struct nome_estrutura VAR_1, VAR_2,...,VAR_M;
```

ou

```
typedef struct nome_estrutura novo_nome_estrutura;  
novo_nome_estrutura VAR_1, VAR_2,...,VAR_M;
```

- Comando **typedef** é usado para definir um novo nome para a estrutura.

- Exemplo de estrutura que armazenaria a matrícula e o nome de um funcionário.

```
1 //Definição
2 struct funcionario {
3     int matricula;
4     char nome[30];
5 };
6
7 //Declaração
8 struct funcionario f1;
```

Estruturas: Definição e Declaração

- Em geral, a definição de um tipo estrutura deve ficar **fora** do programa (principal) e de qualquer sub-rotina.
- A declaração de uma variável do tipo estrutura deve ficar **dentro** do programa (principal) e/ou **dentro** de qualquer sub-rotina.

- Campos ou membros de uma estrutura podem ser usados da mesma forma como as variáveis.
- Campos são acessados usando o operador de acesso ponto (.) entre o **nome da estrutura** e o **nome do campo**.
- Para modificar um campo de uma estrutura, basta usarmos novamente o operador (.).
`scanf("%d", &f1.matricula);`

Exemplo 1: Leitura dos dados da estrutura funcionário

```
1 #include <stdio.h>
2 struct funcionario {
3     int matricula;
4     char nome[30];
5 };
6
7 int main(){
8     struct funcionario f1;
9     scanf("%d", &f1.matricula);
10    gets(f1.nome);
11    puts("Informacoes armazenadas: \n");
12    printf("%d", f1.matricula);
13    puts(f1.nome);
14    return 0;
15 }
```

Exemplo 2: Estrutura para armazenar endereço

```
1 struct est_endereco {  
2     char rua[50];  
3     int numero;  
4     char bairro[20];  
5     char cidade[30];  
6     char sigla_estado[3];  
7     int cep;  
8 };  
9  
10 struct est_endereco end1;
```

Exemplo 3: Estrutura para armazenar endereço (outra maneira)

```
1 typedef struct est_endereco {  
2     char rua[50];  
3     int numero;  
4     char bairro[20];  
5     char cidade[30];  
6     char sigla_estado[3];  
7     int cep;  
8 }endereco;  
9  
10 endereco end1;
```

- typedef: Pode-se utilizar o modificador de tipo na criação da estrutura.
- A estrutura passará a ser referenciada pelo nome que aparece no final da definição.
- A criação de variáveis fica bastante facilitada dessa forma.

Exemplo 4: Definição de uma estrutura onde um de seus campos é outra estrutura (endereço):

```
1 typedef struct est_ficha_pessoal {  
2     char nome[50];  
3     int telefone;  
4     endereco end;  
5 } ficha_pessoal;  
6  
7 ficha_pessoal ficha1, ficha2;
```

- Para acessarmos o campo telefone da variável ficha1 do tipo ficha_pessoal (tipo estrutura), devemos usar a seguinte sintaxe:
ficha1.telefone = 1234567;
- Para acessar os campos da estrutura interna (end), podemos fazer da seguinte forma:
ficha1.end.rua = "Rua das Flores";

Exemplo 5: Tipo estrutura possui vetores como um dos seus campos.

```
1 typedef struct est_ficha_pessoal {  
2     char nome[50];  
3     int telefone;  
4     endereco end;  
5 } ficha_pessoal;  
6  
7 ficha_pessoal ficha1, ficha2;
```

- O acesso a estes campos é feito da mesma maneira como acesso direto a um vetor.
ficha1.nome[1] = 'A';

Exemplo 6: Inicializando uma estrutura

```
1 typedef struct est_ficha_pessoal {  
2     char nome[50];  
3     int telefone;  
4 }ficha_pessoal = {"João das Neves", 1234567};  
5  
6 ficha_pessoal ficha1;
```

- Desta forma, a variável ficha1 inicializará com os campos preenchidos com "João das Neves" e 1234567.

- Uma das vantagens ao utilizarmos estruturas é a possibilidade de copiarmos toda a informação de uma estrutura para outra do mesmo tipo com uma atribuição simples:

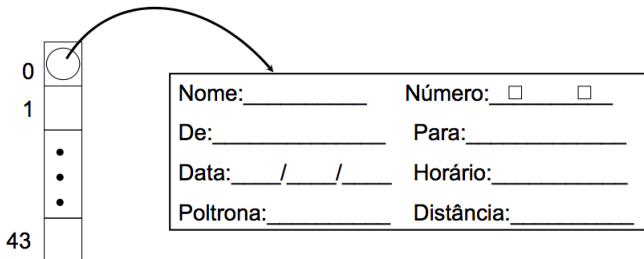
```
1 typedef struct coordenadas {  
2     int x;  
3     int y;  
4 }coordenadas;  
5  
6 coordenadas coord1, coord2;  
7 coord1.x = 10;  
8 coord1.y = 20;  
9 coord2 = coord1;
```

- Como qualquer outra variável, uma variável do tipo estrutura pode ser usada como parâmetro.
- Também, uma variável do tipo estrutura pode ser passada para uma subrotina por referência ou por valor.

- Pode-se criar vetores de estruturas como se criam vetores de tipos primitivos.
- Até o momento só fizemos menção a uma única instância de estrutura.
- É necessário possuir uma definição da estrutura antes de declarar um vetor de estrutura.

- Suponha que deseja-se manter um registro de informações relativas a passagens rodoviárias de todos lugares (poltronas) de um ônibus.
- Pode-se utilizar uma estrutura referente a cada poltrona (passagem) e para agrupar todas elas utiliza-se um vetor de estruturas.

- Um ônibus possui 44 lugares numerados de 0 a 43:



Vetores de estruturas

```
1 typedef struct reg_passagem{
2     char NOME[50];
3     int NUMERO;
4     char ORIGEM[20];
5     char DESTINO[20];
6     char DATA[8];
7     char HORARIO[5];
8     int POLTRONA;
9     float DISTANCIA;
10 }passagem;
11
12 passagem VET_PASSAGEM[44];
```

- Para acessar: `VET_PASSAGEM[3].NUMERO = 42;` // O campo *número da passagem* da posição 3 recebe 42

Faça um programa que leia o nome e as 4 notas escolares de 8 alunos. Imprima a listagem dos alunos, com suas notas e a média das mesmas.

- Crie um procedimento para leitura, um procedimento para impressão e uma função para o cálculo da média.

Exemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef struct ref_aluno{
4     char nome[40];
5     float nota[4];
6 }aluno;
7 float calculaMedia(float notas[], int n){
8     int i;
9     float media = 0.0;
10    for (i = 0; i < n; i++)
11        media += notas[i];
12    return media/(float)n;
13 }
14 void leVetorAlunos(aluno a[], int n){
15     int i, j;
16     for (i = 0; i < n; i++){
17         printf("Informe o nome do aluno \n");
18         gets(a[i].nome);
19         printf("Informe as quatro notas \n");
20         for (j = 0; j < 4; j++)
21             scanf("%f %c", &a[i].nota[j]);
22     }
23 }
```

Exemplo

```
1 void imprimeVetorAlunos(aluno a[], int n){
2     int i,j;
3     printf("\n***Alunos, Notas e Media***\n");
4     for (i = 0; i < n; i++){
5         printf("\n %20s ", a[i].nome);
6         for (j = 0; j < 4; j++)
7             printf("%.2f ", a[i].nota[j]);
8         printf("%.2f", calculaMedia(a[i].nota, 4));
9     }
10 }
11
12 int main(){
13     aluno alunos[8];
14     leVetorAlunos(alunos, 8);
15     imprimeVetorAlunos(alunos, 8);
16     return 0;
17 }
```

Alocação dinâmica e Estruturas

- Para acessar os elementos de um registro através de um ponteiro, devemos primeiro acessar o registro e depois acessar o campo desejado.
- Os parênteses são necessários pois o operador ***** tem prioridade menor que o operador **.**

```
1 typedef struct ref_coordenada{
2     float x;
3     float y;
4 }coordenada;
5
6 coordenada *ponto;
7 ponto = (coordenada*) malloc (sizeof(coordenada));
8 (*ponto).x = 4.0;
9 (*ponto).y = 3.4;
10
11 free(ponto);
```

Alocação dinâmica e Estruturas

- Para simplificar o acesso aos campos de um registro através de ponteiros, foi criado o operador \rightarrow
- Usando este operador acessamos os campos de um registro diretamente através do ponteiro.

```
1 typedef struct ref_coordenada{  
2     float x;  
3     float y;  
4 }coordenada;  
5  
6 coordenada *ponto;  
7 ponto = (coordenada*) malloc (sizeof(coordenada));  
8 ponto->x = 4.0;  
9 ponto->y = 3.4;  
10  
11 free(ponto);
```

Para não confundir: Aqui temos UM ponteiro de estrutura

Alocação dinâmica e Estruturas

- Podemos também usar alocação dinâmica de vetores de estruturas.
- O acesso a cada elemento irá ocorrer pelo uso do operador `.`

```
1 typedef struct ref_coordenada{
2     float x;
3     float y;
4 }coordenada;
5
6 coordenada *vet_pontos;
7 vet_pontos = (coordenada*) malloc (4*sizeof(coordenada));
8 vet_pontos[1].x = 4.0;
9 vet_pontos[1].y = 3.4;
10
11 free(vet_pontos);
```

A variável `x` da posição 1 do vetor recebe 4.0 e a variável `y` da mesma posição recebe 3.4.

Alocação dinâmica e Estruturas

- E se elementos da estrutura forem do tipo ponteiro?
- Deve-se alocar e desalocar cada elemento!

```
1 typedef struct ref_coordenadas{
2     float *conjunto;
3 }coordenadas;
4
5 int i;
6 coordenadas *vet_ponto;
7 vet_ponto = (coordenadas*) malloc (4*sizeof(coordenadas));
8
9 for (i = 0; i < 4; i++)
10     vet_ponto[i].conjunto = (float *) malloc (2*sizeof(float));
11
12 for (i = 0; i < 4; i++){
13     vet_ponto[i].conjunto[0] = i;
14     vet_ponto[i].conjunto[1] = 2*i;
15 }
16
17 for (i = 0; i < 4; i++)
18     free(vet_ponto[i].conjunto);
19 free(vet_ponto);
```


Listas

- Uma lista encadeada é uma representação de uma sequência de objetos na memória do computador.
- Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda e assim por diante.
- Listas Encadeadas x Vetores: listas são mais flexíveis e mais baratas (computacionalmente).

Listas Encadeadas

Uma **lista encadeada** (*linked list* ou lista ligada) é uma sequência de células; cada célula contém um objeto de algum tipo e o endereço da célula seguinte.

```
1
2 struct cel {
3     int  conteudo;
4     struct cel *prox;
5 };
6
7 typedef struct cel lista;
```



Na próxima aula...

Continuação Listas