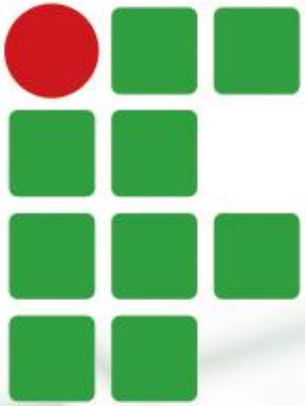

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Estruturas de Dados I

- Alocação Dinâmica -

Alocação Estática x Dinâmica

- Observe o trecho de código a seguir...
  - Qual a capacidade máxima de alunos que esta aplicação conseguirá gerir?
  - Mas, e se o usuário quiser **aumentar a turma** depois que o programa foi compilado???

```
typedef struct{
     int matricula;
     char nome[100];
}Aluno;

int main{
   Aluno turma[30];
}
```

Alocação Estática x Dinâmica

- A alocação das variáveis (vetor de alunos) da aplicação anterior é realizada de forma **estática**, pois acontece **uma única vez**, e não é possível alterar ***durante*** a execução da aplicação.

- Entretanto, existem situações em que a quantidade exata de variáveis só é conhecida ***durante*** a execução da aplicação.

Alocação Estática x Dinâmica

- **Possível solução?**

```
typedef struct{
     int matricula;
     char nome[100];
}Aluno;

int main{
   int n;
   printf("Digite a Qtde. de Alunos na Turma: ");
   scanf(" %d", &n);
   Aluno turma[n];
}
```

Alocação Estática x Dinâmica

- **Possível solução?**

```
typedef struct{
```

Esta solução somente "*mascara*" o mesmo problema...

- **E se o valor "N" informado não for suficiente?**
- **E se o valor "N" informado for muito exagerado?**

Alocação Estática x Dinâmica

- **Possível solução?**

```
typedef struct{
```

Esta "solução" não é satisfatória em termos de desempenho e performance!

- **E se o valor "N" informado for muito exagerado?**

Alocação Estática x Dinâmica

- **Alocação Dinâmica** é a técnica que aloca (reserva) a memória em ***tempo de execução***.
- Isso significa que o espaço de memória para armazenamento de dados é reservado **sob demanda**, **durante a execução da aplicação**.
- Útil nas situações onde não se sabe exatamente quantas variáveis serão necessárias para o armazenamento das informações.

Alocação Estática x Dinâmica

- **Alocação Dinâmica** é a técnica que aloca (reserva) a memória em ***tempo de execução***.

Fica evidente a **melhor utilização e economia** de um dos recursos computacionais mais importantes:

MEMÓRIA PRINCIPAL

armazenamento das informações.

Alocação Dinâmica

- **Alocação Dinâmica** é muito utilizada em problemas de estrutura de dados:
  - Listas encadeadas e generalizações.
  - Estruturas de filas e pilhas.
  - Árvores binárias e grafos.

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Alocação Dinâmica

- A **alocação dinâmica** é gerenciada através de duas funções principais:
  - **`malloc`**
    - *Memory ALLOCation*
  - **`free`**
    - Libera o espaço alocado para uma variável.

- Ambas funções, pertencem à biblioteca:

`<stdlib.h>`

Função de Alocação

```
void* malloc(int tamanho)
```

- *Memory Allocation*
  - A função recebe como parâmetro o número de *bytes* que deseja-se alocar na memória (tamanho).
  - O retorno da função é um **ponteiro do tipo `void`**.

Função de Alocação

```
void* malloc(int tamanho)
```

- *Memory Allocation*
  - A função recebe como parâmetro o número de *bytes* que deseja-se alocar na memória (tamanho).

- ***Ponteiro do tipo `void` ???***
  - A vantagem do **ponteiro `void`** é que ele pode ser **convertido** para qualquer outro tipo de ponteiro, através da técnica de ***typecast***.

Função de Alocação

```
#include “stdlib.h”

int main(){
  char *str;
  str = (char*)malloc(150);
  scanf(“ %[^\n]s”, str);
  printf(“%s”,str);
  getch();
}
```

Função de Alocação

```
#include “stdlib.h”

int main(){
  char *str;
  str = (char*)malloc(150);
  scanf(“ %[^\n]s”, str);
  printf(“%s”,str);
  getch();
}
```

typecast = conversão de tipos

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Função de Alocação

```
void* malloc(int tamanho)
```

Função de Alocação

```
#include “stdlib.h”

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a;
  a = (Aluno*)malloc(¿¿¿???);
}
```

Função `sizeof`

```
int sizeof(type);
```

- A função **`sizeof`** recebe como parâmetro um *tipo de dados* e retorna a quantidade de bytes que esta estrutura de dados ocupa na memória.

Função de Alocação

```
#include “stdlib.h”

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a;
  a = (Aluno*)malloc(sizeof(Aluno));
}
```

Atenção!

- Cuidado ao trabalhar com ***ponteiros de structs***...

```
aluno *a = NULL;
a = (aluno*)malloc(sizeof(aluno));

*a.matricula;       //é equivalente a...
*(a.matricula);     // mas é diferente de...
(*a).matricula;
```

- O operador **->** é uma abreviatura **muito útil.**

dt->dia equivale à **(*dt).dia**

Exemplo de Código

```
#include “stdio.h”
#include “stdlib.h”

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a;
  a = (Aluno*)malloc(sizeof(Aluno));
  scanf(“ %d”, &a->matricula);
  scanf(“ $[^\n]s”, a->nome);
}
```

Função de Liberação

```
void free(void *p)
```

- A função **`free`** é utilizada para liberar o espaço de memória alocado para um ponteiro *p* qualquer.
- É recomendável a utilização da função **`free`** ao término da execução do programa, ou sempre que o espaço de memória de uma variável não for mais útil, para evitar erros inesperados, e para economia de memória do sistema.

Exemplo de Código

```
#include “stdlib.h”

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a;
  a = (Aluno*)malloc(sizeof(Aluno));
  scanf(“ %d”, &a->matricula);
  scanf(“ $[^\n]s”, a->nome);
  free(a);
}
```

Exercício A

- Defina uma *struct* **empregado** para armazenar os dados de **um** funcionário (nome, sobrenome, RG, salário, número da matrícula).
- Declare um ponteiro (*não uma variável*) do tipo **empregado**.
- Faça a alocação dinâmica em memória e realize a leitura e impressão de todas as informações do empregado em **funções** isoladas:
  - ***empregado* setEmpregado()***
  - ***void getEmpregado(empregado* e)***

Vetor x Alocação Dinâmica

- Até então, quando precisamos armazenar uma coleção de dados, de um mesmo tipo abstrato, sempre recorremos a uma estrutura do tipo ***Array***.
- Entretanto, como vimos, um *array* representa a forma mais primitiva de representar diversos elementos agrupados.
  - Isto porque uma estrutura *Array* **não é flexível**.
- Um *array* **sempre** é alocado de maneira estática, portanto:
  - Se o número de elementos exceder a dimensão do vetor, teremos **problemas** de execução.
  - Se o número de elementos é abaixo do limite do vetor, teremos **problemas** de desempenho.

Estruturas de Dados Dinâmicas

- A **solução ótima** para este tipo de situação é a utilização de estruturas que **possam crescer na medida em que precisarmos armazenar novos elementos** (*e diminuam na medida que elementos não forem mais necessários*).

- Tais estruturas são chamadas **dinâmicas** e armazenam cada um dos seus elementos através da técnica de **Alocação Dinâmica**.

Listas Encadeadas

- A **Lista Encadeada** é um exemplo de estrutura de dados **dinâmica**.
- Para cada novo elemento inserido na lista, alocamos um espaço de memória para armazená-lo.
- Desta forma, o espaço total de memória ocupado pela estrutura é proporcional ao número de elementos armazenados na lista.

Entretanto... Como nem tudo são flores, existe uma complexidade maior do que em estruturas do tipo array.

Array x Listas Encadeadas

- Quando declaramos um vetor, alocamos um **espaço contíguo de memória** para armazenar os elementos.

Vetor V

6003	6004	6005	6006	6007	6008	6009	6010	6011	6012
9	5	1	4	6	8	4	7	0	6

- Isso facilita (e *muito*) o acesso a qualquer elemento do vetor, pois basta conhecer o local onde está armazenado o primeiro elemento (que é o endereço da própria variável v) e incrementar o endereço de acordo com a posição desejada.

Array x Listas Encadeadas

- Em uma estrutura do tipo **Lista Encadeada** não é possível obter essa mesma vantagem.
  - Isto porque os elementos são alocados de forma dinâmica (e em posições aleatórias).
  - **Não há como garantir que os mesmos estejam contíguos**.
- Uma estrutura de ***Lista*** portanto, consiste numa seqüência encadeada de elementos, em geral chamados de **nós**, e o encadeamento destes nós é realizado por meio de **ponteiros**.

Listas Encadeadas

- Arranjo de memória de uma lista encadeada.

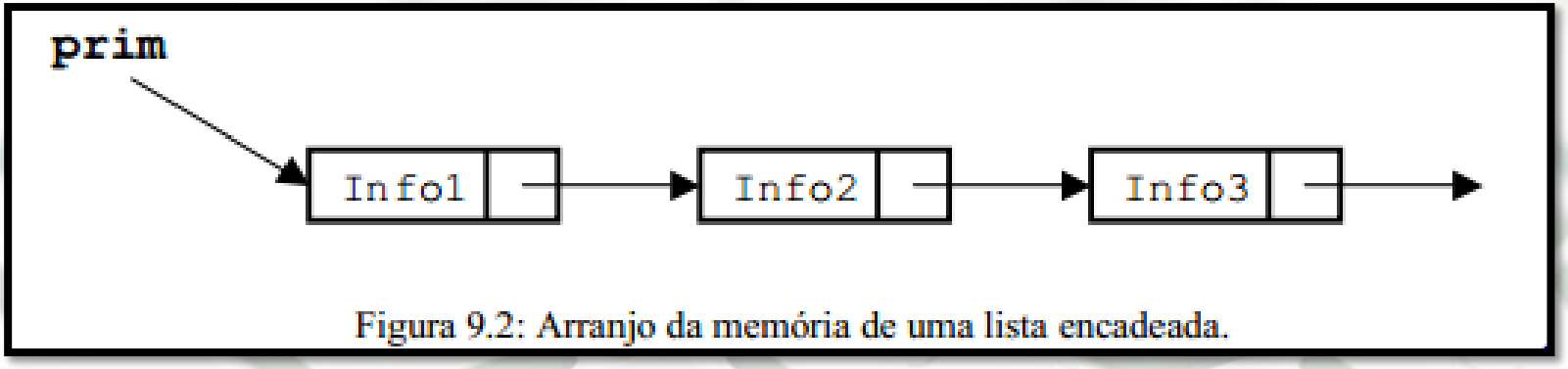

Figura 9.2: Arranjo da memória de uma lista encadeada.

  - Do primeiro elemento, acessamos o segundo.
  - Do segundo o terceiro, e assim por diante...
  - O último elemento da lista, aponta para ***NULL***, sinalizando que não existe um próximo registro.

Lista Encadeada

- Como podemos observar, cada elemento (nó) da lista, deve apontar para o nó subseqüente.
- Este apontamento é realizado através de variáveis do tipo **ponteiro**.
- Portanto, cada elemento (nó) deve possuir, em sua estrutura, uma variável ponteiro para o seu próprio tipo de dados.

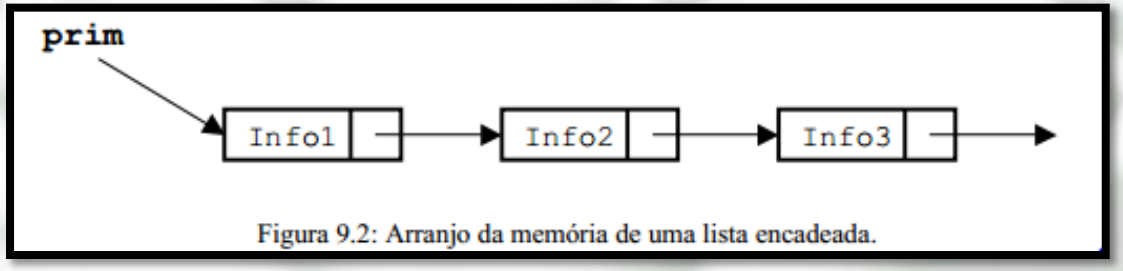

Figura 9.2: Arranjo da memória de uma lista encadeada.

Lista Encadeada

- Traduzindo isso em linguagem de programação...

```
typedef struct no{
    int informacao;
    struct no *prox;
}no;
```

Lista Encadeada

- Uma boa estratégia para referenciarmos uma lista encadeada, é sempre manter armazenado (e atualizado), o ponteiro para **o primeiro nó da lista**.
- A partir do primeiro nó da lista, podemos percorrer todos os encadeamentos subseqüentes.

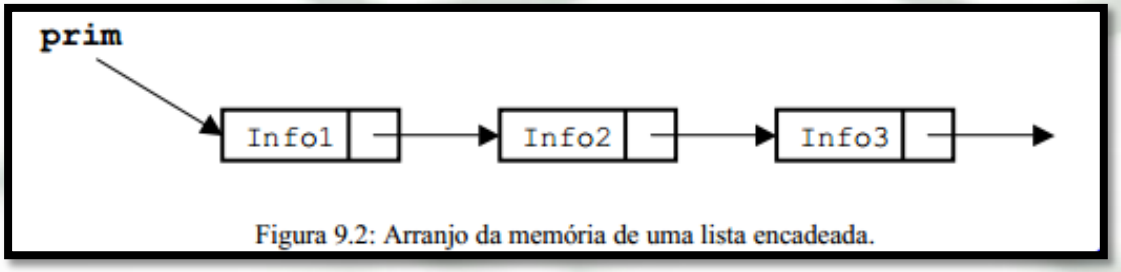

Figura 9.2: Arranjo da memória de uma lista encadeada.

Função para Inserção de Nós

- Uma vez criada a lista, iremos inserir novos elementos...
- Para cada elemento (nó) inserido, é necessário:
  - Alocar o espaço de memória através da função **`malloc()`**.
  - Ler as informações úteis do nó.
  - Encadear este novo nó à lista.
- O método mais simples de inserção insere novos nós no início da lista.
  - **A lista é tratada, portanto, como uma pilha (LIFO).**

Função para Inserção de Nós

```
no* setNo(no *inicio){
   no *novo;
   novo=(no*)malloc(sizeof(no));
   scanf(“ %d”,&novo->informacao);
   novo->prox = inicio;
   return novo;
}

int main(){
   no *lista = NULL;
   for (int i=0; i<10; i++)
       lista = setNo(lista);
}
```

Função para Acesso aos Nós

- Versão **Iterativa**...

```
void getNos(no *pont){
  while(pont){
      printf("\n%d",pont->informacao);
      pont = pont->prox;
  }
}
```

Função para Acesso aos Nós

- Versão **Recursiva**...

```
void getNos(no* pont){
  if(pont)
      printf("\n%d", pont->informacao);
  getNos(pont->prox);
}
```

Exercício B

- Faça um programa modular, que realize o cadastro dinâmico de estruturas do tipo carro (ano, modelo, valor, placa e proprietário).
  - Utilize uma estrutura do tipo Lista Encadeada.
  - Implemente uma função para o cadastro de nós.
  - Implemente uma função para listagem dos carros.
  - Implemente uma função que, através da placa, imprima todos os dados de um carro.
  - Implemente uma função que retorna o valor médio dos carros cadastrados no sistema.

Exercício C

- Outra abordagem para **Lista Encadeadas** é o tratamento de Filas (**FIFO**), onde os nós são inseridos ao final da lista, e não no início.
- Para facilitar esta rotina, além do ponteiro indicando o início da lista, também é armazenado um ponteiro que sempre aponta para o **último elemento da lista**.
- As leituras se baseiam no ponteiro início, enquanto as inclusões são baseadas no ponteiro fim.

- ***Altere o exercício anterior, fazendo os cadastros como uma estrutura do tipo FIFO.***