


GOVERNO DO ESTADO DO RIO DE JANEIRO

UERJ – UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO - ZO

CTC – CENTRO DE TECNOLOGIAS E CIÊNCIAS

FCEE – FACULDADE DE CIÊNCIAS EXATAS E ENGENHARIAS

DEPCOMP – DEPARTAMENTO DE COMPUTAÇÃO

CURSOS DE COMPUTAÇÃO

Estrutura de Dados I

Parte 4 – Estruturas e Alocação Dinâmica de Memória.

Prof. Raul Queirós

- Estruturas: Structs
- Alocação Dinâmica de Memória

Estrutura de dados I

Estruturas – ***struct***s – são usadas quando há necessidade de se combinar dados de um mesmo tipo ou de tipos diferentes em um único objeto ou registro.

Forma de definição:

struct <nome>

{

<tipo 1> < campo 1 , campo 2 , . . . , campo n>;

<tipo 2> < campo 1 , campo 2 , . . . , campo n>;

. . . . .

. . . . .

<tipo m> < campo 1 , campo 2 , . . . , campo n>;

};

Exemplos:

struct s1 // Define estrutura chamada **s1** contendo um conjunto campos

Estrutura de dados I

Estruturas são usadas, para que possamos agrupar dados de um determinado registro. Uma vez que uma estrutura é definida, ela pode ser utilizada na declaração de variáveis ou novas estruturas.

1 - Definição de estrutura:

```
struct funcionario {
        char nome [40];
        char endereco [40];
        char conjuge [40];
        int num_dep;
        float salario;
        char data_adimis [10];
};
```

A estrutura definida se torna um tipo de dado e pode ser usada na declaração das variáveis analista e programador:

```
struct funcionario analista, programador;
```

ou de forma direta:

```
struct funcionario{
        char nome [40];
        char endereco [40];
        char conjuge [40];
        int num_dep;
        float salario;
        char data_adimis [10];
} analista, programador;
```

Obs.: O uso de sizeof(analista); resultará na quantidade de bytes de toda a estrutura. Serão somados os bytes necessários para o armazenamento de cada membro da estrutura.

Estrutura de dados I

Como criar um vetor de estruturas (struct):

```
struct funcionario {
        char nome [40];
        char endereco [40];
        char conjuge [40];
        int num_dep;
        float salario;
        char data_admis [10];
}  vetFunc[50];
```

Estrutura de dados I

2 - Inicialização de estrutura:

```
struct agenda
{
        char nome [40];
        char endereco [40];
        char telefone [10];
        char data_nasc [10];
} amigos =  {
        “João da Silva”,
        “Rua do Arroz, 10/ 102”,
        “2223-4545”,
        “01/01/1910“
};
```

Estrutura de dados I

Inicialização de estruturas compostas:

```
struct data {       int dia;
                    int mes;
                    int ano;
};
struct agenda {
            char nome[40];
            char end[40];
            char tel[9];
            struct data datanasc;
} amigos = {
            "João da Silva",
            "Rua do Arroz, 10/ 102",
            "2223-4545",
            {       15,
                    5,
                    1950
            }
};
```

Estrutura de dados I

Obs.1: A estrutura data foi utilizada dentro da estrutura agenda, ficando responsável pela data de nascimento no exemplo e se fosse necessário colocar outro campo do tipo data na estrutura, como por exemplo data de admissão, poderíamos definir outra vez a estrutura data com o nome data_admin, por exemplo:

```
struct agenda {
        char nome[40];
        char end[40];
        char tel[9];
        struct data datanasc;
        struct data data_admin;
} amigos;
```

Estrutura de dados I

Obs.2: na prática podemos omitir as chaves da estrutura data ao inicializarmos a estrutura amigos.

Ex.:

```
{       "João da Silva",
        "Rua do Arroz, 10/ 102",
        "2223-4545",
                15,
                5,
                1950
};
```

Estrutura de dados I

3 - Vetores de Estruturas:

Uma estrutura é um tipo de dado, assim como as variáveis. Deste modo, podemos criarmos ponteiros para estruturas.

Ex.: struct data *pt_data;

Criamos então um ponteiro pt_data para a estrutura data. Deste modo, pt_data aponta para o primeiro byte da estrutura data.

Podemos nos referir aos campos da estrutura de duas formas:

- (* ponteiro).campo , onde ponteiro é o ponteiro para a estrutura e campo, o nome do campo desejado dentro da estrutura. Os parênteses são obrigatórios em razão de o operador . ter precedência sobre o operador *;
- ponteiro->campo , da mesma forma que o exemplo anterior, ponteiro é o ponteiro para a estrutura e campo, o campo desejado na estrutura. Esta é a forma mais comum de programação, encontrada nos sistemas.

Estrutura de dados I

Para acessarmos o mês da estrutura data pelo ponteiro pt_data, como no exemplo anterior, faríamos:

```
(*pt_data).mes;
```

ou

```
pt_data->mes;
```

4 - *Pointers* para Estruturas

Como vimos anteriormente, devemos passar as estruturas através de ***pointers*** (ponteiros) e não por valor. Isto deve ser feito mesmo que não alteremos nenhum dos campos da estrutura.

```
/* Exemplo - Pointer para estruturas */
#include <stdio.h>
struct s_data
{
                    int dia, mes, ano;
};
typedef struct s_data sdata;

void prtdata (s_data *);
void prtdata(s_data *data)
{
                    printf("   %d/% d/ %d \n", (*data).dia, (*data).mes, (*data). ano);
}
```

Estrutura de dados I

```
void main ( )
{
        sdata data;
        data.dia        = 15;
        data.mes        = 3;
        data.ano        = 1988;
        prtdata ( &data);  /* passa o endereço da estrutura data */
   }
```

Estrutura de dados I

Lembrando que existem duas formas de se acessar os valores de uma estrutura através de pointers:

(* < pointer >**).**< componente > ou < pointer > –> < componente >;

Exemplo:

```
sdata data, *pdata;
pdata = &data;
(*pdata).dia    = 31;        // ou   pdata–>dia = 31;
(*pdata).mes    = 12;        // ou   pdata–>mes = 12;
(*pdata).ano    = 1989;      // ou   pdata–>ano = 1989;
```

Estrutura de dados I

```
#include <stdio.h>

struct s_data {
          int dia, mes, ano;
};
typedef struct s_data sdata;



void main ( ) {
   sdata data, *pdata;
          pdata = &data;
          (*pdata).dia        = 31;
          (*pdata).mes        = 12;
          (*pdata).ano        = 1989;
          printf("\n %d/%d/%d\n", (*pdata).dia, (*pdata).mes, (*pdata).ano);


          pdata->dia = 07;
          pdata->mes = 04;
          pdata->ano = 2020;
          printf("\n %d/%d/%d\n", pdata->dia, pdata->mes, pdata->ano);

}
```

Alocação dinâmica de memória.

Estrutura de dados I

O tipo de armazenamento de uma estrutura na memória do computador pode ser dividida em:

- **Alocação Estática (sequencial):** os dados tem um tamanho fixo e estão organizados seqüencialmente na memória do computador. Ex: variáveis globais, e os vetores (estáticos). Em geral, o armazenamento sequencial é empregado quando as estruturas sofrem poucas remoções e inserções.
- **Alocação Dinâmica (encadeada):** a quantidade de memória necessária é alocada e desalocada no momento de execução do programa. Sendo assim os blocos de memória não precisam estar dispostos sequencialmente. Para poder "achar" os blocos esparsos na memória usamos as variáveis do tipo *Ponteiro* (indicadores de endereços de memória). Ex: variáveis locais e os parâmetros passados por referência.

A escolha do tipo de alocação depende das operações que serão executas na estrutura de dados e suas características.

Alocação Sequencial (Estática)

- não usa a memória de forma eficiente, pois aloca um espaço finito e pré-determinado de memória;
- para inserções e remoções é necessário um grande número de movimentações

Na **alocação dinâmica** (encadeada) as posições de memória são alocadas (ou desalocadas) na medida em que são necessárias.

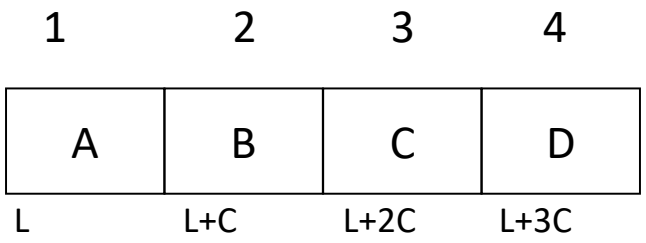
Os nós de uma estrutura dinâmica de dados encontram-se aleatoriamente dispostos na memória sendo interligados por ponteiros, que indicam a posição do próximo elemento da estrutura.

Para isso, é necessário um **campo adicional** a cada elemento para indicar o endereço do próximo nó, que chamaremos de *prox*. O último elemento da estrutura não possui "próximo nó" portanto apontará para NULL(ninguém).

Estrutura de dados I

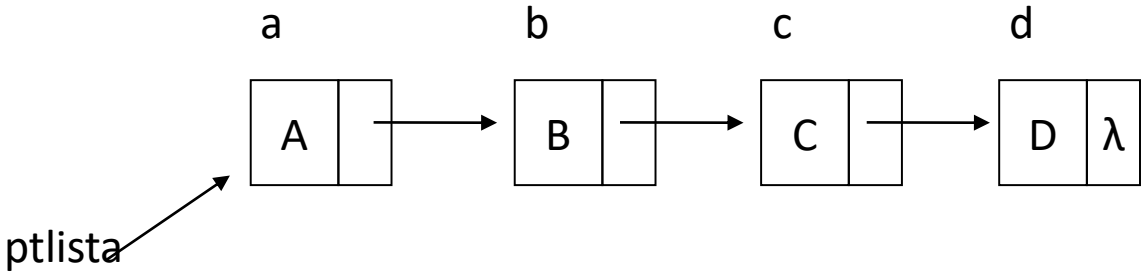
******** Representação estática (vetor)

vet: vetor[1..4] de char



******* Representação dinâmica (ponteiros)

*ptlista: char



LED - Lista de Espaços Disponíveis

É uma lista especial que contém posições livres de memória, utilizada por um algoritmo de gerenciamento de memória para encontrar espaço para armazenamento de novos nós nas estruturas dinâmicas de memória.

Pode-se construir algoritmos que implementem uma LED com um único vetor de nós que simule a memória total disponível. Neste caso, cada nó deste vetor representa um endereço de memória disponível para uso.

No entanto, as linguagens de programação geralmente fornecem um módulo de gerência de memória, bastando apenas utilizar rotinas internas de alocação e liberação de memória quando da inclusão e remoção de nós de uma estrutura dinâmica.

Em nossa sintaxe de estudo utilizaremos as rotinas:

- Procedimento **aloca(point**) e // reserva um espaço de memória para o novo nó;
- Procedimento **desaloca(point**), // libera a memória ocupada pelo nó removido onde *point* é o ponteiro que aponta para o nó a ser inserido ou removido.

Estrutura de dados I

Conclusão

Alocação Dinâmica (Encadeada)

- nós ocupam mais espaço em virtude do novo campo ponteiro *prox*;
- acesso a qualquer elemento da lista obriga ao percurso na lista até o elemento desejado;
- pode-se usar memória fragmentada;
- as posições de memória são alocadas (ou desalocadas) na medida em que são necessárias.

Alocação Estática (Sequencial)

- acesso a qualquer elemento da lista é imediato;
- não usa a memória de forma eficiente, pois aloca um espaço finito e pré-determinado de memória;
- para inserções e remoções em listas é necessário um grande número de movimentações de nós;

Alocação Dinâmica

Na alocação dinâmica podemos alocar espaços durante a execução de um programa, ou seja, a alocação dinâmica é feita em tempo de execução. Isto é bem interessante do ponto de vista do programador, pois permite que o espaço em memória seja alocado apenas quando necessário. Além disso, a alocação dinâmica permite aumentar ou até diminuir a quantidade de memória alocada.

- **sizeof -** A função sizeof determina o número de bytes para um determinado tipo de dados.

É interessante notar que o número de bytes reservados pode variar de acordo com o compilador utilizado.

Exemplo: x = sizeof(int); //retorna 4 no gcc

- **malloc -** A função malloc aloca um espaço de memória e retorna um ponteiro do tipo void para o início do espaço de memória alocado.
- **free -** A função free libera o espaço de memória alocado.

Alocação Dinâmica

- **calloc() -** também é utilizada para alocar memória, mas além do protótipo um pouco diferente, a principal diferença entre ela e sua similar (malloc) é que calloc, ao reservar o endereçamento solicitado, automaticamente preencher esses endereço com o valor de partida 0. Enquanto malloc mantém o valor da mémoria reservada, o chamado lixo de buffer.

  Sintaxe: (tipo*)(calloc(quantidade_de_memoria, sizeof(tipo)));

- **realloc() -** é utilizada para realocar memória.

  Sintaxe: void *realloc (void *ptr, unsigned int num);

Estrutura de dados I

- Operador sizeof()
  - Retorna o número de bytes de um dado tipo de dado.

Ex.: int, float, char, struct...

```c
struct ponto{
    int x,y;
};

int main(){

    printf("char: %d\n", sizeof(char));// 1
    printf("int: %d\n", sizeof(int));// 4
    printf("float: %d\n", sizeof(float));// 4
    printf("ponto: %d\n", sizeof(struct ponto));// 8

    return 0;
}
```

Estrutura de dados I

- Operador sizeof()
  - No exemplo anterior,

    p = (int *) malloc(50*sizeof(int));
- sizeof(int) retorna 4
- número de bytes do tipo int na memória
- Portanto, são alocados 200 bytes (50 * 4)
- 200 bytes = 50 posições do tipo int na memória

Estrutura de dados I

Se não houver memória suficiente para alocar a memória requisitada, a função malloc() retorna um ponteiro nulo.

```
int main(){
    int *p;
    p = (int *) malloc(5*sizeof(int));
    if(p == NULL){
        printf("Erro: Memoria Insuficiente!\n");
        system("pause");
        exit(1);
    }
    int i;
    for (i=0; i<5; i++){
        printf("Digite o valor da posicao %d: ",i);
        scanf("%d",&p[i]);
    }

    return 0;
}
```

Estrutura de dados I

- calloc

A função calloc() também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num, unsigned int size);
```

- Funcionalidade
  - Basicamente, a função calloc() faz o mesmo que a função malloc(). A diferença é que agora passamos a quantidade de posições a serem alocadas e o tamanho do tipo de dado alocado como parâmetros distintos da função.

Estrutura de dados I

```c
int main(){
    //alocação com malloc
    int *p;
    p = (int *) malloc(50*sizeof(int));
    if(p == NULL){
        printf("Erro: Memoria Insuficiente!\n");
    }
    //alocação com calloc
    int *p1;
    p1 = (int *) calloc(50,sizeof(int));
    if(p1 == NULL){
        printf("Erro: Memoria Insuficiente!\n");
    }

    return 0;
}
```

- **realloc**

A função realloc() serve para realocar memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

- Funcionalidade
  - A função modifica o tamanho da memória previamente alocada e apontada por *ptr para aquele especificado por num.
  - O valor de num pode ser maior ou menor que o original.
  - Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco para aumentar seu tamanho.
  - Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco, e nenhuma informação é perdida.

Estrutura de dados I

```c
int main(){
    int i;
    int *p = malloc(5*sizeof(int));
    for (i = 0; i < 5; i++){
        p[i] = i+1;
    }
    for (i = 0; i < 5; i++){
        printf("%d\n",p[i]);
    }
    printf("\n");
    //Diminui o tamanho do array
    p = realloc(p,3*sizeof(int));
    for (i = 0; i < 3; i++){
        printf("%d\n",p[i]);
    }
    printf("\n");
    //Aumenta o tamanho do array
    p = realloc(p,10*sizeof(int));
    for (i = 0; i < 10; i++){
        printf("%d\n",p[i]);
    }

    return 0;
}
```

Estrutura de dados I

- **free**
  - Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa.
  - Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária.
  - Para isto existe a função free() cujo protótipo é:

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

Estrutura de dados I

```c
int main(){
    int *p,i;
    p = (int *) malloc(50*sizeof(int));
    if(p == NULL){
        printf("Erro: Memoria Insuficiente!\n");
        system("pause");
        exit(1);
    }
    for (i = 0; i < 50; i++){
        p[i] = i+1;
    }
    for (i = 0; i < 50; i++){
        printf("%d\n",p[i]);
    }
    //libera a memória alocada
    free(p);

    return 0;
}
```

Exercite o seu conhecimento
com os seguintes exemplos práticos:

Estrutura de dados I

```
#include <stdio.h>
#include <stdlib.h>
int main (){
    int *p;
    int num;
    printf("\nDigite o tamanho do vetor-->");
    scanf("%d", &num); // grava na variável (num) o tamanho do
                       // vetor que será alocado
    p=(int *)malloc(num*sizeof(int));
    if (!p){
        printf ("** \n\nErro: Memoria Insuficiente\n\n **");
        exit;
    }else{
        printf ("** \n\nMemoria Alocada com Sucesso\n\n **");
    }
    return (0);
}
```

Estrutura de dados I

```
#include <stdio.h>
#include <stdlib.h> //necessário para usar as funções malloc() e free()
#include <conio.h>
int main(void) {
    float *v; //definindo o ponteiro v
    int i, num_componentes;
    printf("Informe o numero de componentes do vetor\n");
    scanf("%d", &num_componentes);
    v = (float *) malloc(num_componentes * sizeof(float));
    //Armazenando os dados em um vetor
    for (i = 0; i < num_componentes; i++) {
         printf("\nDigite o valor para a posicao %d do vetor: ", i+1);
         scanf("%f",&v[i]);
    }
    // ------ Percorrendo o vetor e imprimindo os valores ----------
    printf("\n********** Valores do vetor dinamico ***********\n\n");
    for (i = 0;i < num_componentes; i++) {
         printf("%.2f\n",v[i]);
    }
    //liberando o espaço de memória alocado
    free(v);
    getch();
    return 0;
}
```

Estrutura de dados I

```
Informe o numero de componentes do vetor
5

Digite o valor para a posicao 1 do vetor: 24

Digite o valor para a posicao 2 do vetor: 56

Digite o valor para a posicao 3 do vetor: 45

Digite o valor para a posicao 4 do vetor: 76

Digite o valor para a posicao 5 do vetor: 56

********** Valores do vetor dinamico **********

24.00
56.00
45.00
76.00
56.00
```

Estrutura de dados I

Utilização de realloc

```
#include <stdio.h>
#include <stdlib.h>
int main(void)   {
    int *p;
    int i,k,n;
    printf ("\nDigite a quantidade de números :");
    fflush(stdin);
    scanf ("%d",&i);

/* a função malloc reserva espaço suficiente para um vetor de
   inteiros. Caso sejam digitados 5 elementos serão reservados 20
   bytes, pois cada inteiro ocupa 4 bytes na memória */

    p=(int*)malloc(i*sizeof(int));
    if (p==NULL)
```

Estrutura de dados I

```
for (k=0;k<i;k++) {
    printf ("\nDigite o %do valor do vetor: ",k+1);
    fflush(stdin);
    scanf ("%d",&p[k]);
}
printf ("\n\nVETOR.\n");
for (k=0;k<i;k++) {
        printf ("%d\t",p[k]); }
printf ("\n\nSeu vetor possui %d elementos.",i);
printf ("\nDigite um valor positivo para aumentar ao vetor.");
printf ("\nDigite um valor negativo para diminuir do vetor.");
printf ("\n\n-->");
fflush(stdin);
scanf ("%d",&n);
```

Estrutura de dados I

```
 if (!(i+n)) {
          printf ("\nSeu vetor possui 0 elementos.\n\n");
          free(p);
          return 0;
          system("pause");
      }
      else if ((i+n)<0)
      {
          printf ("\nSeu vetor possui qtd negativa de
elemento.\n\nIMPOSSIVEL ALOCAR MEMORIA.\n\n");
          free(p);
          return 0;
          system("pause");
      }
```

Estrutura de dados I

```
 /* a função realloc aumenta (numero positivo) ou diminui (numero
negativo), o tamanho do vetor dinamicamente. ela recebe o ponteiro
para o vetor anterior e retorna o novo espaço alocado */
      p=(int*)(realloc(p,(i+n)*sizeof(int)));
      if (p==NULL) {
          printf ("\nERRO DE RE-ALOCACAO.MEMORIA INSUFICIENTE");
          exit(1);
      }
      for (k=0;k<(n+i);k++) {
          printf ("\nDigite o %do valor do vetor: ",k+1);
          fflush(stdin);
          scanf ("%d",&p[k]);
      }
      printf ("\n\nVETOR.\n");
      for (k=0;k<(n+i);k++)
      printf ("%d\t",p[k]);
      free(p); // libera a memória alocada
      system("pause");
      return 0;
}
```

Alocação Dinâmica de Memória

- Estruturas Alocadas Dinamicamente
  - estruturas também podem ser alocadas dinamicamente.

Exemplo de uso de Estruturas com alocação dinâmica de memória:

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

struct ST_DADOS
{
  char nome[40];
  float salario;

  // estrutura dentro de uma estrutura
  struct nascimento
  {
     int ano;
     int mes;
     int dia;
   } dt_nascimento;
};

int main(void)
{
   //ponteiro para a estrutura
   struct ST_DADOS *p;

   p = (struct ST_DADOS *)malloc(sizeof(struct
          ST_DADOS));

   // p->nome é um ponteiro, não precisa do &
   printf("\nEntre com o nome ->");
   scanf("%s", p->nome);

   printf("Entre com o salario ->");
   scanf("%f", &p->salario);

   printf("Entre com o nascimento ->");
   scanf("%d%d%d", &p->dt_nascimento.dia,
   &p->dt_nascimento.mes,
   &p->dt_nascimento.ano);

   printf("\n===== Dados digitados ====");
   printf("\nNome = %s", p->nome);
   printf("\nSalario = %f", p->salario);
   printf("\nNascimento = %d/%d/%d\n",
        p->dt_nascimento.dia,
        p->dt_nascimento.mes,
        p->dt_nascimento.ano);

   free(p);
   return 0;
}
```

Exercícios Práticos 1 (Parte 4):

- Reproduza os exemplos todos os exemplos práticos apresentados e capture as telas de execução;
- Monte um documento do Word com todos os exemplos e suas execuções ordenando os exemplos e entregue como exercícios da parte 4.

Estrutura de dados I

Exercícios práticos 2 (Parte 4):

Uso de structs com vetores:

1. Escreva um programa em linguagem C, que preencha, a partir do teclado, duas estruturas distintas do tipo vetor com os nomes e as notas (as notas têm de estar contidas no intervalo ***0 ≤ nota ≤ 10***) dos alunos, respectivamente, de uma turma de 10 (dez) alunos.

   Após, exteriorize somente os nomes dos alunos que obtiveram notas iguais ou maiores que 5 (cinco).

1. Escreva um programa em linguagem C, que preencha, a partir do teclado, duas estruturas distintas do tipo vetor com as idades de 10 (dez) pessoas.

   A primeira estrutura do tipo vetor deverá receber somente as idades das pessoas do sexo masculino, enquanto a segunda deverá armazenar as idades das pessoas do sexo feminino.

   Após, o programa deverá exteriorizar os nomes, o sexo e as idades das pessoas que possuem idade compreendida entre 20 (vinte) e 40 (quarenta) anos, inclusive.

3. Escreva um programa em linguagem C, que preencha 5 (cinco) estruturas do tipo registro com os nomes e as idades de 5 (cinco) pessoas – cada conjunto de informações sobre uma pessoa deverá ser conteúdo de um registro – e, após, visualize o nome da(s) pessoa(s) de mais idade.

Uso de structs com alocação dinâmica de memória:

4. Escreva um programa em linguagem C, que preencha uma estrutura do tipo **vetor** de **registros**, de dimensão igual a 10 (dez), onde cada registro deve conter o ***nome*** e a ***idade*** de uma pessoa, informados através do teclado e que, **após**, visualize o nome da pessoa de menor idade e o nome da pessoa de mais idade.

5. Escreva um programa em linguagem C, que preencha uma estrutura do tipo **vetor** de registros, onde cada elemento deverá armazenar o nome, a idade e os 5 graus (notas) obtidos em 5 (cinco) verificações de aprendizagem, de uma turma de 10 (dez) alunos.

   Após o programa deverá visualizar os nomes de todos os alunos da turma acompanhados de sua situação acadêmica – ***aprovado*** ou ***reprovado***.

   A situação de ***aprovado*** somente pode ser computada se o aluno obtiver todos os seus graus maiores ou iguais a 5 (cinco). Caso contrário, ele deverá ser assinalado como ***reprovado***.

6. Escreva um programa em linguagem C, que preencha uma estrutura do tipo vetor de registros, de dimensão 10 (dez), onde cada registro deve armazenar o nome e a idade de uma pessoa e que, após o preenchimento do vetor, apresentar os nomes dos alunos de menor e maior idades, respectivamente.

Obrigado pela atenção!

Fim da parte 4.