# Aula 31: Revisão para Prova 03

## Introdução a Programação

**Túlio Toffolo & Puca Huachi**
http://www.toffolo.com.br

BCC202 — 2019/2
Departamento de Computação — UFOP

# Avaliação

- 3 Provas (60% da nota):
  - Prova 01: 15% da nota
  - Prova 02: 20% da nota
  - **Prova 03: 25% da nota**
- Exercícios em aula práticas (10% da nota):
  - Atividades em todas as aulas serão entregues via *moodle*.
- Trabalho(s) prático(s) (30% da nota):
  - **Entrega final na próxima semana**
  - Código e documentação serão entregues via *moodle*.
  - Apresentação para o(s) professor(es) da disciplina no final do semestre.
- Ponto extra: frequência e exercícios nas aulas teóricas

# Revisão

- A seguir alguns slides foram selecionados das aulas anteriores para uma breve revisão do conteúdo.
- A revisão está organizada em 3 partes:

  1. Estruturas de dados heterogêneas
  2. Alocação dinâmica
  3. Arquivos de texto e binários

# 1

# Estruturas de dados heterogêneas

# Struct

- `struct`: palavra reservada que cria um novo tipo de dados.
- Tipos conhecidos: `char`, `int`, `float`, `double` e `void`.
- Estrutura: é um tipo de estrutura de dados heterogênea; agrupa itens de dados de diferentes tipos.
- Cada item de dado é denominado membro (ou campo);
- `struct` define um tipo de dados (estrutura): informa ao compilador o nome, o tamanho em bytes e a maneira como ela deve ser armazenada e recuperada da memória.
- Ao ser definido, o tipo passa a existir e pode ser utilizado para criar variáveis.

# Exemplo: armazenando dados de um aluno

```c
#include <stdio.h>

struct Aluno {
    int nMat;       // número de matrícula
    float nota[3]; // três notas
    float media;   // média aritmética
};                 // fim da definição da estrutura (com ;)

int main()
{
    struct Aluno bart;   // declara a variável do tipo 'struct Aluno'
    bart.nMat = 1521001;
    bart.nota[0] = 8.5;
    bart.nota[1] = 9.5;
    bart.nota[2] = 10.0;
    bart.media = ( bart.nota[0]+ bart.nota[1] + bart.nota[2] ) / 3.0;
    printf("Matrícula: %d\n", bart.nMat);
    printf("Média    : %.1f\n", bart.media);
    return 0;
}
```

# O comando *typedef*

- O comando `typedef` define um apelido (*alias*) para um tipo.
- Em geral, apelidos simplificam o uso de estruturas em C.
- Exemplo:

```
1  typedef struct { // não precisamos definir o nome aqui
2      int  dia;
3      char mes[10];
4      int  ano;
5  } Data;           // 'apelido' (novo nome) para a estrutura: Data
```

- Uso simplificado (omitimos a palavra *struct* ao declarar variáveis):

```
1  Data natal = { 25, "Dezembro", 2016 };
2
3  Data natalDesteAno;
4  natalDesteAno = natal;
```

## Exemplo de uso e operações em structs:

```c
typedef struct {
    int   pecas;
    float preco;
} Venda;

int main()
{
    Venda A = {20, 110.0};
    Venda B = {3, 258.0};
    Venda total;

    // soma membro a membro
    total.pecas = A.pecas + B.pecas;
    total.preco = A.preco + B.preco;
}
```

## Erro comum

```c
// ERRO!
total = A + B;
```

# Estruturas aninhadas

```
1  typedef struct {
2      int  dia;
3      char mes[10];
4      int  ano;
5  } Data;
6
7  typedef struct {
8      int   pecas;
9      float preco;
10     Data  diaVenda;
11 } Venda;
12
13 int main()
14 {
15     // exemplo de declaração
16     Venda v = {20, 110.0, {7, "Novembro", 2015} };
17
18     // exemplo de uso:
19     printf("Ano da venda: %d", v.diaVenda.ano);
20
21     return 0;
22 }
```

# Estruturas em funções

As estruturas podem ser passadas como argumentos de funções da mesma maneira que as variáveis simples.

- O nome de uma estrutura em C não é um endereço, portanto ela pode ser passada por **valor**.
- Exemplo: função que recebe duas estruturas como argumento e imprime os valores da soma de seus membros.

```
1  typedef struct {
2      int   pecas;
3      float preco;
4  } Venda;
5
6  // protótipo (com passagem por valor)
7  void imprimeTotal(Venda v1, Venda v2);
```

# Estruturas em funções

Exemplo utilizando passagem por **valor**:

```
1  typedef struct {
2      int   pecas;
3      float preco;
4  } Venda;
5
6  void imprimeTotal(Venda v1, Venda v2)
7  {
8      Venda total = {0, 0.0};
9      total.pecas = v1.pecas + v2.pecas;
10     total.preco = v1.preco + v2.preco;
11     printf("Nro peças:   %d\n", total.pecas);
12     printf("Preço total: %.2f\n", total.preco);
13 }
14
15 int main()
16 {
17     Venda v1 = {1, 20}, v2 = {3, 10};
18     imprimeTotal(v1, v2);
19     return 0;
20 }
```

# Estruturas em funções

Exemplo utilizando **ponteiros**:

```
1  typedef struct {
2      int   pecas;
3      float preco;
4  } Venda;
5
6  void imprimeTotal(Venda *v1, Venda *v2)
7  {
8      Venda total = {0, 0.0};
9      total.pecas = (*v1).pecas + (*v2).pecas;
10     total.preco = (*v1).preco + (*v2).preco;
11     printf("Nro peças:   %d\n", total.pecas);
12     printf("Preço total: %.2f\n", total.preco);
13 }
14
15 int main()
16 {
17     Venda v1 = {1, 20}, v2 = {3, 10};
18     imprimeTotal(&v1, &v2);
19     return 0;
20 }
```

# Estruturas em funções

Exemplo utilizando **ponteiros** (alternativa):

```
1  typedef struct {
2      int   pecas;
3      float preco;
4  } Venda;
5
6  void imprimeTotal(Venda *v1, Venda *v2)
7  {
8      Venda total = {0, 0.0};
9      total.pecas = v1->pecas + v2->pecas;
10     total.preco = v1->preco + v2->preco;
11     printf("Nro peças:   %d\n", total.pecas);
12     printf("Preço total: %.2f\n", total.preco);
13 }
14
15 int main()
16 {
17     Venda v1 = {1, 20}, v2 = {3, 10};
18     imprimeTotal(&v1, &v2);
19     return 0;
20 }
```

v1->pecas ou (*v1).pecas

2

# Alocação dinâmica

# Alocação dinâmica

Uso do método `malloc` para criar um bloco com 3 doubles:

```
1  // aloca memória de forma dinâmica e inicializa com valor 10.5
2  double *nro = malloc(3 * sizeof(double));
3
4  // alterando valores
5  nro[0] = 1.1;
6  nro[1] = 1.5;
7  nro[2] = 2.2;
8
9  for (int i = 0; i < 3; i++)
10     printf("%.1lf ", nro[i]);
```

- Este código imprimirá:

```
1  1.1 1.5 2.2
```

# Muito importante: liberar a memória

- A memória alocada de forma estática pelo compilador é liberada automaticamente.
- Quando fazemos alocação dinâmica, **liberar a memória se torna nossa responsabilidade**.
- Em C usamos o procedimento `free`
- Exemplo (1):

```
1  int *a = malloc(sizeof(int));
2  ...
3  free(a);
```

# Muito importante: liberar a memória

- A memória alocada de forma estática pelo compilador é liberada automaticamente.
- Quando fazemos alocação dinâmica, **liberar a memória se torna nossa responsabilidade**.
- Em C usamos o operador `free`
- Exemplo (2):

```
1  int *a = malloc(100 * sizeof(int));
2  ...
3  free(a);
```

# Exemplo:

| Endereço | Conteúdo | Nome |
|---|---|---|
| | | |
| 0x1000 | | |
| 0x1004 | | |
| 0x1008 | | |
| 0x1012 | | |
| 0x1016 | | |
| 0x1020 | | |
| 0x1024 | | |
| 0x1028 | | |
| 0x1032 | | |
| 0x1036 | | |
| 0x1040 | | |
| 0x1044 | | |
| 0x1048 | | |
| 0x1052 | | |
| 0x1056 | | |
| 0x1060 | | |
| | | |

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

void leVetor(int *v, int n) {
    for (int i = 0; i < n; i++)
        scanf("%d", &v[i]);
}

int *maior(int *v, int n) {
    int *maior = v;
    for (int i = 1; i < n; i++)
        if (v[i] > *maior)
            maior = v + i;
    return maior;
}

int main() {
    int n, *v;
    scanf("%d", &n);
    v = malloc(n * sizeof(int));
    leVetor(v, n);
    int *valor = maior(v, n);
    printf("Maior = %d\n", *valor);
    free(v);
    return 0;
}
```

# Alocação dinâmica: erros comuns

## Qual o erro no código abaixo?

```c
int main()
{
    int n;
    printf("Qual o tamanho do vetor? ");
    scanf("%d", &n);

    int *vetor = malloc(n * sizeof(int));

    usaVetor(vetor, n); // função que faz algum uso do vetor...

    printf("Qual o tamanho do segundo vetor? ");
    scanf("%d", &n);
    vetor = malloc(n * sizeof(int));

    usaVetor(vetor, n); // função que faz o segundo uso do vetor...

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

## Qual o erro no código abaixo?

```c
int *criaPreencheVetor(int tamanho)
{
    int *vetor = malloc(tamanho * sizeof(int));
    for (int i = 0; i < tamanho; i++) {
        vetor[i] = i;
    }

    return vetor;
}

int main()
{
    int n;
    printf("Qual o tamanho do vetor? ");
    scanf("%d", &n);

    int *vetor = malloc(n * sizeof(int));
    vetor = criaPreencheVetor(n);

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

# Matrizes dinâmicas

Matrizes dinâmicas são, na verdade, vetores de vetores!

- Criamos um vetor de ponteiros
- Criamos, para cada ponteiro, um vetor!

# Matrizes dinâmicas

Exemplo:

```
1  // função que cria uma matriz de tamanho n x m
2  int ** criaMatriz(int n, int m)
3  {
4      int **matriz;
5      matriz = malloc(n * sizeof(int*));
6      for (int i = 0; i < n; i++) {
7          matriz[i] = malloc(m * sizeof(int));
8      }
9      return matriz;
10 }
```

Mas... como liberar a memória alocada para a matriz?

# Matrizes dinâmicas

Como liberar a memória alocada por uma matriz?

- Lembre-se: um `free` para cada `mallloc`!

Exemplo:

```
1  void liberaMatriz(int **matriz, int n, int m)
2  {
3      for (int i = 0; i < n; i++)
4          free(matriz[i]);
5      free(matriz);
6  }
```

- Obs: note que a variável `m` não é necessária!

# Erros comuns

## O que está errado no código abaixo?

```
1  void liberaMatriz(int **matriz, int n, int m)
2   {
3       free(matriz);
4       for (int i = 0; i < n; i++)
5           free(matriz[i]);
6   }
```

Note que neste contexto a **a ordem** faz toda a diferença!

- Após liberar a memória apontada por `matriz`, não podemos mais acessar `matriz[i]`.

# Referência de ponteiros

Em alguns casos, pode ser útil passar a referência de um ponteiro para uma função.

Exemplo:

- Alocar memória para dois vetores
- Armazenar o endereço em ponteiros passados por parâmetro

```
1  /* Função fictícia que aloca memória para dois vetores de tamanho n */
2  void alocaVetores(int **vetor1, int **vetor2, int n);
```

- Note o tipo `int**`: trata-se de um ponteiro para ponteiro
- Como implementar a função acima?

```c
/* Função fictícia que aloca memória para dois vetores de tamanho n */
void alocaVetores(int **vetor1, int **vetor2, int n) {
    *vetor1 = malloc(n * sizeof(int));
    *vetor2 = malloc(n * sizeof(int));
}
```

Observações:

- Note que o conteúdo de um ponteiro para ponteiro é um **ponteiro**!
- Note o exemplo abaixo, de como chamar a função acima:

```c
int main() {
    int n = 100;
    ...
    int *vetor1, *vetor2;
 → alocaVetores(&vetor1, &vetor2, n); // passagem por referência
    ...
    free(vetor1);
    free(vetor2);
    return 0;
}
```

# Referência de ponteiros

**Atenção**: cuidado para não confundir com matrizes:

- A referência de um ponteiro `int*` é do tipo `int**`
- A referência de uma matriz `int**` é do tipo `int***`

```c
#include <stdlib.h>

void criaVetor(int **vetor, int n)
{
    *vetor = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        (*vetor)[i] = 0;

    // poderíamos fazer tb:
    // *vetor = calloc(n, sizeof(int));
}

int main()
{
    printf("Digite o tamanho do vetor: ");
    int n;
    scanf("%d", &n);

    int *vetor;
    criaVetor(&vetor, n); // passagem por referência

    ... // faz algo com o vetor

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

# Exercícios

## Exercício 1

Crie um procedimento `alocaMatriz` que recebe por parâmetro:

1. referência de um ponteiro para alocar e retornar uma matriz de inteiros;
2. número de linhas (`n`);
3. número de colunas (`m`).

A função deve alocar uma matriz `n`$\times$`m` dinamicamente e preencher todos os campos com valor zero.

**Importante:** implemente a função `main` para mostrar um exemplo de uso da função criada anteriormente.

# Exercício 1 - Solução

```c
#include <stdlib.h>

void alocaMatriz(int ***matriz, int n, int m) {
    *matriz = malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++)
        (*matriz)[i] = calloc(m, sizeof(int));
}

// exemplo (simples) de uso da função acima
int main() {
    int **mat;
    alocaMatriz(&mat, 10, 20);

    // ...
    // faz algo com a matriz
    // ...

    for (int i = 0; i < 10; i++)
        free(mat[i]);
    free(mat);

    return 0;
}
```

3

# Arquivos de texto e binários

# Arquivos

## Arquivo texto

- Armazena caracteres seguindo uma codificação (utf-8, por exemplo).
- Exemplo:

```
1 Este é um arquivo de texto, composto por caracteres...
2  - abc
3  - def...
```

## Arquivo binário

- Sequência de bits sujeita às convenções do programa que o gerou.
- Exemplos: arquivos executáveis, compactados, de registros, etc.

# Biblioteca <stdio.h>

C fornece o tipo FILE para representar um arquivo.
Na prática, usamos um ponteiro do tipo FILE.

Exemplo de declaração:

```
1  // arquivo para leitura
2  FILE *entrada;
3
4  // arquivo para gravação
5  FILE *saida;
```

# Biblioteca <stdio.h>

A função `fopen` é usada para abrir um arquivo e tem o seguinte protótipo:

```
1  FILE * fopen(const char *filename, const char *mode);
```

Note que a função tem 2 parâmetros:

1. `filename`: nome do arquivo a ser aberto
2. `mode`: modo de abertura do arquivo
   - `"r"` (read): **leitura**
   - `"w"` (write): **gravação** (sobrescreve o arquivo, se existir)
   - `"r+"` (read/update): **leitura** e **gravação** (arquivo tem que existir)
   - `"w+"` (write/read): **leitura** e **gravação**
   - `"a+"` (append/update): **acrescenta** dados no arquivo

# Biblioteca <stdio.h>

A função `fopen` é usada para abrir um arquivo e tem o seguinte protótipo:

```
1  FILE * fopen(const char *filename, const char *mode);
```

Note que a função tem 2 parâmetros:

1. `filename`: nome do arquivo a ser aberto
2. `mode`: modo de abertura (note que o `b` indica "modo" binário)
   - `"rb"` (read): **leitura**
   - `"wb"` (write): **gravação** (sobrescreve o arquivo, se existir)
   - `"rb+"` (read/update): **leitura** e **gravação** (arquivo tem que existir)
   - `"wb+"` (write/read): **leitura** e **gravação**
   - `"ab+"` (append/update): **acrescenta** dados no arquivo

# Biblioteca <stdio.h>

Após abrir um arquivo, **temos que fechá-lo** com a função `fclose`.

```
1  int fclose(FILE *stream);
```

A função retorna 0 em caso de sucesso e `EOF` (-1) caso contrário.

# Biblioteca <stdio.h>

Para impressão (gravar no arquivo), podemos utilizar a função `fprintf`, cujo funcionamento é muito parecido com a função `printf`.

```
1  int fprintf(FILE *stream, const char *format, ... );
```

Exemplo:

```
1  FILE *arquivo = fopen("texto.txt", "w");
2
3  // escrevendo texto e um número inteiro no arquivo
4  int n = 10;
5  fprintf(arquivo, "O valor de n = %d\n", n);
6
7  fclose(arquivo);
```

# Biblioteca <stdio.h>

Para leitura, podemos utilizar a função `fscanf`, cujo funcionamento é muito parecido com a função `scanf`.

```
1  int fscanf(FILE *stream, const char *format, ... );
```

- A função retona o número de argumentos preenchidos ou `EOF` se o fim do arquivo for atingido.

Exemplo de uso:

```
1  FILE *arquivo = fopen("file.txt", "r");
2
3  // lendo um inteiro e um caractere separados por um espaço
4  int inteiro;
5  char caractere;
6  fscanf(arquivo, "%d %c", &inteiro, &caractere);
7
8  fclose(arquivo);
```

# Outra funções

A biblioteca <stdio.h> fornece outras funções úteis para **ler** dados de um arquivo texto:

```
// lê uma linha, incluindo o '\n' de um arquivo (lembram dela?)
char *fgets (char *str, int num, FILE *stream);

// lê um caractere e retorna (sim, retorna como um inteiro)
int fgetc(FILE *stream);

// retorna 0 se a posição atual não for o fim do arquivo
// e um valor diferente de 0 caso contrário
int feof(FILE *stream);
```

# Outra funções

A biblioteca <stdio.h> também fornece outras funções úteis para **gravar** dados em um arquivo texto:

```c
// escreve uma string no arquivo
// a função retorna EOF em caso de erro
int fputs(const char *str, FILE *stream);

// escreve um caractere no arquivo (sim, como um inteiro);
// a função retorna EOF em caso de erro
int fputc(int character, FILE *stream);

// retorna 0 se a posição atual não for o fim do arquivo
// e um valor diferente de 0 caso contrário
int feof(FILE *stream);

// atualiza o arquivo (grava todo o conteúdo que ainda não foi
// gravado); retorna 0 em caso de sucesso e EOF caso contrário
int fflush(FILE *stream);
```

# Exemplo 5

Crie uma estrutura `Pessoa` contendo *primeiro nome*, *CPF* e *salário* de um funcionário. Em seguida, crie um programa que lê os dados de $n$ funcionários e escreve os dados lidos em um arquivo texto como o apresentado a seguir:

```
1  Nome                 CPF                  Salario
2  ---------------------------------------------
3  Renata               123.456.789-10       2000.00
4  Rodrigo              123.456.789-10       1000.00
5  Tulio                123.456.789-10       1000.00
```

Crie duas funções:

```
1  // le os dados de n funcionarios e retorna um vetor
2  Pessoa *lePessoas(int n);
3
4  // salva no arquivo "saida" os dados de n funcionarios
5  void gravaPessoas(char saida[], Pessoa *pessoas, int n);
```

# Exemplo 5

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

typedef struct {
    char nome[15]; // primero nome (de até 15 caracteres)
    char cpf[15];  // armazenado como uma string
    float salario;
} Pessoa;

Pessoa *lePessoas(int);
void gravaPessoas(char[], Pessoa*, int);

int main() {
    int n;
    printf("Digite o nro de funcionarios: ");
    scanf("%d", &n);

    Pessoa *pessoas = lePessoas(n);
    gravaPessoas("funcionarios.txt", pessoas, n);
    free(pessoas);
    return 0;
}
```

```c
Pessoa *lePessoas(int n) {
    Pessoa *pessoas = malloc(n * sizeof(Pessoa));
    for (int i = 0; i < n; i++) {
        printf("Digite nome, CPF e salário do funcionário %d:\n", i+1);
        scanf("%s", pessoas[i].nome);
        scanf("%s", pessoas[i].cpf);
        scanf("%f", &pessoas[i].salario);
    }
    return pessoas;
}

void gravaPessoas(char saida[], Pessoa *pessoas, int n) {
    FILE *file = fopen(saida, "w");

    // imprimindo cabecalho
    fprintf(file, "%-15s %-15s %10s\n", "Nome", "CPF", "Salario");
    fprintf(file, "-----------------------------------------------\n");

    // imprimindo dados dos funcionários
    for (int i = 0; i < n; i++)
        fprintf(file, "%-15s %-15s %10.2lf\n",
                pessoas[i].nome, pessoas[i].cpf, pessoas[i].salario);

    fclose(file);
}
```

# Arquivos binários

- Sequência de bits sujeita às convenções do programa que o gerou.
- Muitos úteis para salvar informação de forma compacta.
- Permitem, por exemplo, armazenar registros (como `structs`) em arquivos.
- Exemplos de arquivos binários:
  - arquivos executáveis,
  - arquivos compactados,
  - arquivos de registros, etc.

# Biblioteca <stdio.h>

Para gravar *bytes* no arquivo, usamos a função `fwrite`.

```
1  size_t fwrite(const void *ptr, size_t size, size_t count, FILE *file);
```

- A função retona o número de elementos gravados com sucesso.

Exemplo:

```
1  FILE *arquivo = fopen("texto.dat", "wb");
2
3  // escrevendo sizeof(int) bytes no arquivo
4  int n = 10;
5  fwrite(&n, sizeof(int), 1, arquivo);
6
7  fclose(arquivo);
```

# Biblioteca <stdio.h>

Para ler *bytes* do arquivo, usamos a função `fread`:

```
1  size_t fread(void *ptr, size_t size, size_t count, FILE *file);
```

- A função retona o número de elementos lidos.

Exemplo de uso:

```
1  FILE *arquivo = fopen("file.dat", "rb");
2
3  // lendo um inteiro e um caractere
4  int inteiro;
5  char caractere;
6  fread(&inteiro, sizeof(int), 1, arquivo);
7  fread(&caractere, sizeof(char), 1, arquivo);
8
9  fclose(arquivo);
```

# A função `fseek`

A função `fseek` reposiciona o indicador de **posição** em um arquivo.

```
1  int fseek(FILE *file, long int offset, int whence);
```

- A função retorna 0 em caso de sucesso e outro valor caso contrário.

Note que a função tem 3 parâmetros:

1. `file`: ponteiro para o arquivo considerado;
2. `offset`: quantidade de *bytes* de deslocamento (podemos utilizar números negativos);
3. `whence`: indica de onde o deslocamento é feito:
   - `SEEK_SET`: **início** do arquivo;
   - `SEEK_CUR`: posição **atual** no arquivo;
   - `SEEK_END`: **final** do arquivo.

# A função `ftell`

A função `ftell` retorna a **posição** atual em um arquivo (em bytes):

```
1  long int ftell(FILE *file);
```

- A função retorna a **posição** atual no arquivo (em *bytes*).

# Exemplo de uso:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE* arquivo = fopen(argv[1], "rb"); // b indica modo binário

    // move para o indicador de posição para o final do arquivo
    fseek(arquivo, 0, SEEK_END);

    // captura a posição atual (em bytes)
    long int tamanho = ftell(arquivo);

    printf("O arquivo %s tem %ld bytes\n", argv[1], tamanho);

    fclose(arquivo);
    return 0;
}
```

`fseek` e `ftell` também podem ser utilizados em arquivos de texto:

```c
#include <stdio.h>

int main() {
    FILE* arquivo = fopen("arquivo.txt", "w");

    fprintf(arquivo, "Imprimindo um texto no arquivo arquivo.txt\n");

    // movendo indicador de posição em -5 bytes
    fseek(arquivo, -5, SEEK_CUR);
    fprintf(arquivo, ".TXT");

    // movendo indicador de posição para 14 bytes a partir do início
    fseek(arquivo, 14, SEEK_SET);
    fprintf(arquivo, "TEXTO");

    // movendo indicador de posição para o final do arquivo
    fseek(arquivo, 0, SEEK_END);
    printf("Tamanho do arquivo: %ld bytes\n", ftell(arquivo));

    fclose(arquivo);
    return 0;
}
```

# Exemplo

Escreva um programa que lê $n$ inteiros da entrada e a escreve:

- No arquivo texto `"vetor.txt"`
- No arquivo binário `"vetor.dat"`

Em seguida, compare o conteúdo (e tamanho) dos arquivos.

Exemplo de entrada: número de inteiros seguido pelos valores inteiros

```
1 8
2 1000000 2000000 3000000 4000000 5000000 6000000 7000000 8000000
```

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

int main()
{
    int n, *v;
    FILE *txt, *bin;

    // lendo vetor da entrada
    scanf("%d", &n);
    v = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        scanf("%d", &v[i]);

    // escrevendo vetor em arquivo texto
    txt = fopen("vetor.txt", "w");
    fprintf(txt, "%d\n", n);
    for (int i = 0; i < n; i++)
        fprintf(txt, "%d ", v[i]);   // escrevendo vetor (elemento por elemento)
    fclose(txt);

    // escrevendo vetor em arquivo binário
    bin = fopen("vetor.dat", "wb");
    fwrite(&n, sizeof(int), 1, bin);
    fwrite(v, sizeof(int), n, bin);  // escrevendo bloco de memória do vetor
    fclose(bin);

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

# Perguntas?