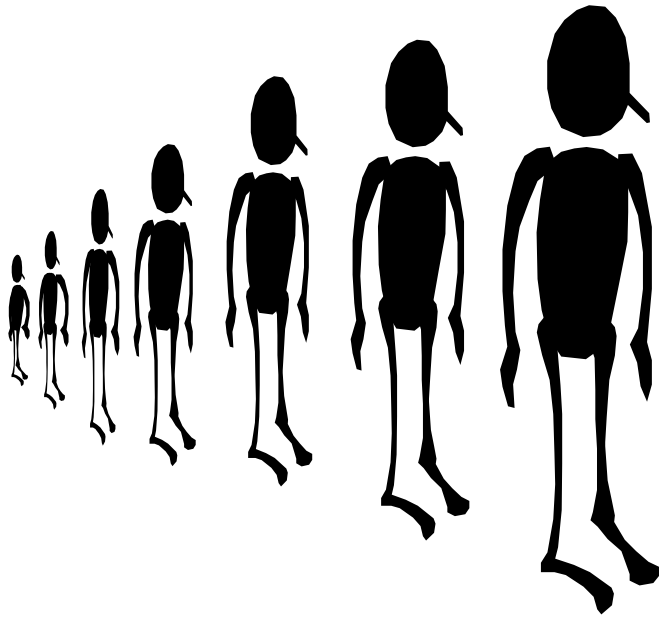
Paradigmas de Programação

Josino Rodrigues

Programando em C

- **Comandos de Repetição**
  - A estrutura de repetição permite que um bloco de instruções seja executado repetidamente uma quantidade controlada de vezes.
    - **For**
    - **While**
    - **Do/while**



Programando em C

- **Comandos de Repetição**
  - **For (Para)**

```
for (var-controle = valor-inicial; condição; incremento)
        Comando;
```

```
for (var-controle = valor-inicial; condição; incremento)
{       Comando;

        Comando;
}
```

```
for (var-controle = valor-inicial; condição; decremento)
        Comando;
```

Programando em C

- **Comandos de Repetição**
  - For (Para)
    - Exemplos

```
#include <conio.h>
#include <stdio.h>
main ()
{ int i;
  for (i=1; i<=10; i++)
      printf("\n %d", i);
  getche();
}
```

```
#include <conio.h>
#include <stdio.h>
main ()
{ int i;
  for (i=10; i>=1; i--)
      printf("\n %d", i);
  getche();
}
```

Programando em C

- **Comandos de Repetição**
  - **While (Enquanto – teste no início)**
    - enquanto a condição for verdadeira, o comando será executado repetidamente. Se a condição for falsa, então a repetição será interrompida.
    - A sintaxe do While é:

```
while (condição)
    comando;
```

```
while (condição)
{    comando;
     ...;
     comando;
}
```

Programando em C

- **Comandos de Repetição - While (Enquanto – teste no início)**

```
#include <stdio.h>
#include <conio.h>
// Calcula bonus do Cliente
main (){
        float valor ;
        int cod, cont;
        printf("Loja Compre Tudo\n\n");
        cont = 1;
        while (cont <= 5){
                printf("Codigo do Cliente %d: ", cont);
                scanf("%d", &cod);
                printf("Valor das Compras: ");
                scanf("%f", &valor);
                if (valor> 5000)
                                printf("\n\n%Valor do bonus R$  %2.f\n", valor*0.1);
                else
                                printf("\n\n%Valor do bonus R$  %2.f\n", valor*0.15);
                printf("\n\ntecle entre para continuar...\n");
                getche();
                cont++;
        }
}
```

Programando em C

- **Comandos de Repetição**
  - **Do/While (Faça...enquanto - teste no final)**
    - Repete um bloco de instruções enquanto a condição é verdadeira.
    - A diferença é que a condição só é testada no final.
    - Sua sintaxe é:

```
do
   comando;
while (condição);
```

```
do
{   comando;
      ....;
   comando;
} while (condição);
```

Programando em C

- **Comandos de Repetição - Do/While**

```
#include <stdio.h>
#include <conio.h>
// Calcula bonus do Cliente
main (){
float valor ;
int cod, op;
printf("Loja Compre Tudo\n\n");
        do{
                printf("Codigo do Cliente ");
                scanf("%d", &cod);
                printf("Valor das Compras: ");
                scanf("%f", &valor);
                if (valor< 5000)
                                printf("\n\nValor do bonus R$  %2.f\n", valor*0.1);
                else
                                printf("\n\nValor do bonus R$  %2.f\n", valor*0.15);
                printf("\n\n Deseja continuar (1-sim/2-nao)?");
                scanf("%d",&op);
                } while (op ==1 );
}
```

Programando em C

- **Comandos Desestruturadores**
  - **Break**
    - Este comando força o encerramento de uma repetição. Sintaxe do break:
      - `break;`
  - **Continue**
    - O comando “continue” funciona de maneira análoga ao “break”, contudo ao invés de forçar o encerramento da repetição, força nova iteração saltando o código entre seu uso e a marca de término da repetição. Sintaxe do continue:
      - `continue;`

Modularização

Programando em C

- **Modularização – Funções e Procedimentos**
  - Modularização serve para dividir um grande programa em diversas partes menores.
  - Ao dividir um programa em módulos, diversas vantagens são encontradas:
    - impedir que o programador tenha que repetir o código diversas vezes
    - facilitar o trabalho de encontrar erros no código.
  - Um módulo deverá possuir um nome e a lista de argumentos que receberá e o tipo de retorno.

Programando em C

- **Modularização – Funções e Procedimentos**
  - **Declarando um módulo**

```
tipo nome(lista de parametros)
{ declaração de variáveis locais;
  comando ou bloco de comandos;
}
```

  - **Variáveis Locais**
    - **A declaração das variáveis, em C, deve vir no início do módulo, antes de qualquer comando.**

Programando em C

- **Modularização – Funções e Procedimentos**

  - **Chamando Módulo**
    - `nome(argumentos);`
    - `nome();`
    - `variavel = nome(argumentos);`

  - **Os módulos só podem ser chamados depois de terem sido declarados.**

Programando em C

- **Modularização – Funções e Procedimentos**
  - **Comando return**
    - **Serve para retornarmos um valor calculado dentro de um módulo quando chamado de alguma parte do programa.**
    - **Exemplo :**

```
float calc_sin(float arg)
{ float val;
  val = sin(arg);
  return(val);
}
void main( )
{ float valor;
  valor = calc_sin(50);
  printf("%f",valor);
}
```

Programando em C

- **Modularização – Funções e Procedimentos**
  - **Argumentos**
    - **Argumentos são utilizados para transmitir informações para o módulo.**
    - **O quinto tipo existente em C, *void* (vazio, em inglês), é um tipo utilizado para representar o nada. Nenhuma variável pode ser declarada como sendo do tipo void. A função main() é um exemplo de função sem argumentos.**
  - **Exemplo:**

```
void teste(int x, int y)
void teste(int x, int y, float z)
int teste(int x, int y)
```

Armazenamento em Vetores

Programando em C

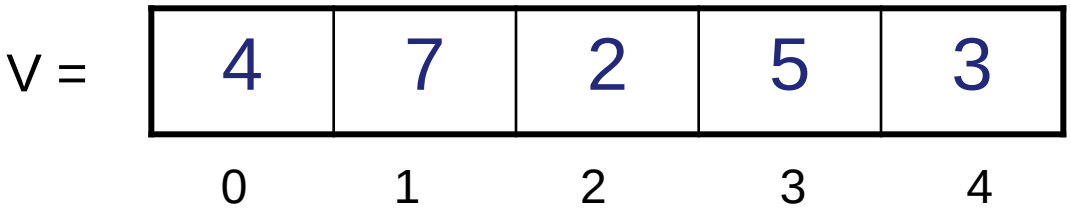
- **Tipos de Dados Estruturados**
  - **Vetores**
    - **Declaração**
      - `Tipo-da-variavel  Nome_Vetor[quant_elementos];`
      - Exemplos:

```
int v[30];
float n[10];
```

    - **Atribuição**

```
v[i] = 10;
v[2] = 12 + x;
v[i+2] = 14;
```

- **Tipos de DadosEstruturados**
  - **Vetores**
    - As *variáveis compostas homogêneas*, mais conhecidas como ***arrays***, correspondem a conjuntos de elementos de um mesmo tipo, representados por um único nome.
    - **Vetor**



    - Cada elemento do array pode ser referenciado através de índices. Exemplos:

V[0] = 4
V[1] = 7
V[4] = 3

Programando em C

- **Tipos de Dados Estruturados**
  - **Vetores**
    - **Leitura**

```
for (i=0; i<30; i++)
{  printf("\n\nElemento %d: ",i+1);
   scanf("%d",&v[i]);
}
```

    - **Impressão**

```
printf("\n\nVetor: ");
for (i=0; i<30; i++)
     printf(" %d ",v[i]);
```

Programando em C

- **Exercícios**

1. Faça um programa que leia e armazene em um vetor uma seqüência de n números e imprima na tela os dados lidos na ordem inversa à da leitura.

2. Faça um programa que leia e armazene em um vetor de 10 elementos e imprima em tela quais desses elementos são números primos e qual a posição ocupada por ele no vetor.

3. Faça um programa que leia e armazene em um vetor 10 valores e inverta a ordem dos elementos no vetor.

4. Faça um programa que leia e armazene em um vetor 10 valores ordene o vetor em ordem crescente imprima-o em seguida.

Armazenamento em Registro

- **Tipos de Dados Estruturados**
  - **Registros**
    - São conjuntos de dados logicamente relacionados, mas de tipos diferentes (inteiro, real, char, etc.).
    - Exemplo: Numa dada aplicação, podem-se ter os seguintes dados sobre os alunos de uma turma:
      - Matricula
      - Nome
      - Nota1
      - Nota2
      - Media
    - Cada conjunto de informações do aluno pode ser referenciado por um mesmo nome, por exemplo, ALUNO. Tais estruturas são conhecidas como registros, e aos elementos do registro dá-se o nome de campos.

Programando em C

- **Tipos de Dados Estruturados**
  - **Registros**
    - **Declaração**

```
typedef struct {  variaveis
                }nome_do_tipo;
```

    - **Exemplo:**

```
typedef struct { int mat;
                 char nome[20];
                 float n1, n2, med;
                }TAlunos;

TAlunos aluno;
```

aluno

mat
nome
n1
n2
med

Programando em C

- **Tipos de Dados Estruturados**
  - **Registros**
    - **Atribuição**

```
Nome-do-registro.campo = valor;
```

    - **Exemplos**

```
aluno.n1 = 5.0;
scanf("%d", &aluno.mat);
fflush(stdin);
gets(aluno.nome);
```

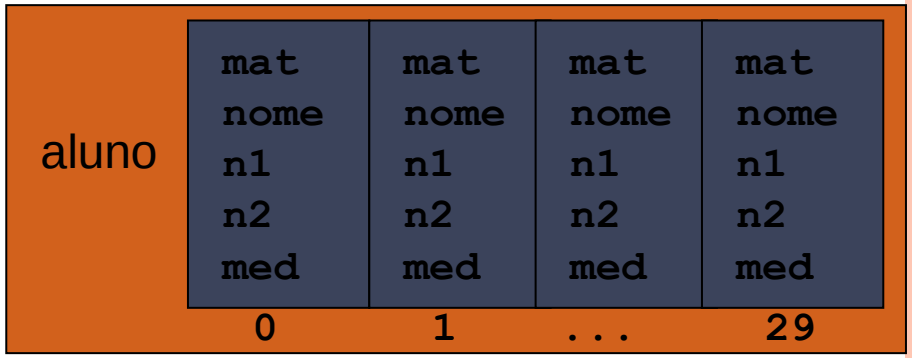
Tipos de Dados Estruturados

Vetor de Registro

- Podemos ter um conjunto de registros referenciados por um mesmo nome e individualizados por índices, através da utilização de um vetor de registros.
- **Exemplo:**

```
typedef struct { int mat;
                 float n1, n2, med;
               }TAlunos;

TAlunos aluno[30];
```



Programando em C

- **Tipos de Dados Estruturados**
  - **Vetor de Registro**
    - **Atribuição**
      - `nome-do-vetor[indice].campo = valor;`
    - **Exemplo**
      - `aluno[3].n1 = 5.0;`
      - `scanf("%d", &aluno[i].mat);`
      - `fflush(stdin);`
      - `gets(aluno[2].nome);`

Arquivos

- **O que são arquivos?**
  - são estruturas de dados manipuladas fora do ambiente do programa.

- Armazenamento de dados de forma “definitiva”.

Arquivos

- **Ponteiros**
  - é um tipo de variável que armazena um endereço de memória.
  - Ao trabalhar com arquivos, precisamos saber em qual endereço de memória o arquivo está armazenado.
  - O endereço de memória onde o arquivo está armazenado, será colocado em ponteiro.

Arquivos

- **Ponteiros**
  - **Declaração**
    - `tipo  *nome_do_ponteiro;`
  - **Ponteiro para Arquivo**
    - `FILE  *nome_do_ponteiro;`
    - **Exemplos:**
      - `FILE *paluno;`
      - `FILE *pproduto;`

Arquivos

- **Comandos**
  - funções reunidas em stdio.
  - Todas estas funções trabalham com o conceito de "ponteiro de arquivo".

Arquivos

- **Declaração de um arquivo**
  - FILE *p;
    - p será então um ponteiro para um arquivo.

Arquivos

- **Abrindo um arquivo**
  - **fopen**

Esta é a função de abertura de arquivos.

Sintaxe:

```
pont_arquivo =  fopen("nome_arquivo",
"modo_abertura");
```

Arquivos

- **Abrindo um arquivo**
  - **fopen**

Modo	Significado
"r+b"	Abre um arquivo binário para leitura e escrita. O arquivo deve existir antes de ser aberto.
"w+b"	Cria um arquivo binário para leitura e escrita. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
"a+b"	Acrescenta dados ou cria um arquivo binário para leitura e escrita. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.

Arquivos

- **Abrindo um arquivo**
  - **fopen**

```
FILE *paluno, *pprofessor;
paluno = fopen("alunos.bin", "w+b");
pprofessor = fopen("professores.bin", "r+b");
if (pprofessor == NULL)
  printf(”Arquivo de professores não existe”);
```

Arquivos

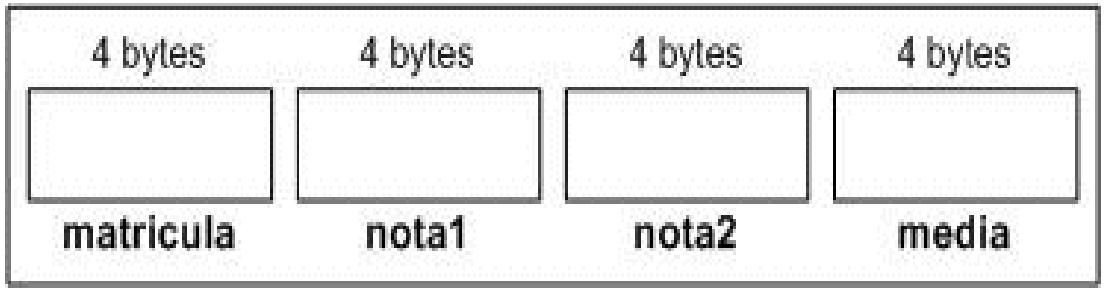
- **Fechando um arquivo**
  - **fclose**
    - Quando acabamos de usar um arquivo que abrimos, devemos fechá-lo.
    - **Sintaxe:**
      - `fclose (pont_arquivo);`
    - **Exemplo:**
      - `fclose(fp);`

Arquivos

- **Ler registro do arquivo**
  - A leitura é feita a partir do ponto onde o leitor se encontra.
  - Ao abrir o arquivo, o leitor é posicionado no início do primeiro registro, À medida que vamos executando leituras, o leitor vai se deslocando. Dessa forma, precisamos ter noção da posição onde o leitor se encontra.
  - Nos arquivos binários, SEMPRE fazemos a leitura de um registro completo, independente de quantos campos ele tenha.
  - Um arquivo deve armazenar registro do mesmo tipo.

Arquivos

- **Ler registro do arquivo**
  - fread
    - Faz a leitura de um registro, no ponto onde o leitor se encontra.
    - Sintaxe:
      - `fread (&registro, numero_bytes, quant, pont_arquivo);`



Arquivos

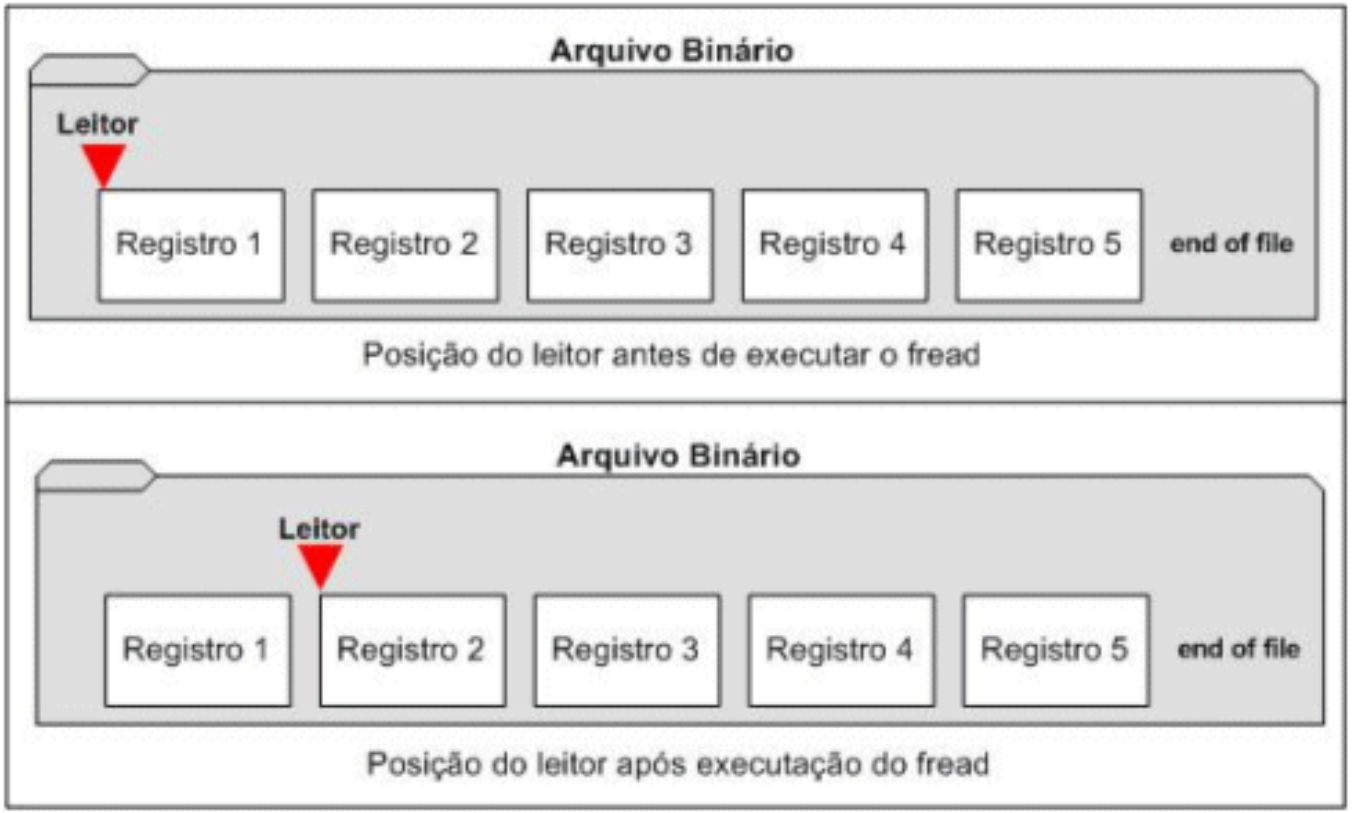
- **Ler registro do arquivo**

```
typedef struct{ int matricula;
          float nota1, nota2, media;
        }TAluno
TAluno aluno;
FILE *paluno;

fread(&aluno, sizeof(TAluno), 1, paluno);
```

Arquivos

- **Ler registro do arquivo**



Arquivos

- **Gravar registro no arquivo**
  - fwrite
    - Faz a gravação de um registro, no ponto onde o leitor se encontra.
    - Sintaxe:
      - `fwrite(&registro, numero_bytes, quant, pont_arquivo);`

Arquivos

- **Gravar registro no arquivo**

```
typedef struct{ int matricula;
          float nota1, nota2, media;
        }TAluno
TAluno aluno;
FILE *paluno;

fwrite(&aluno, sizeof(TAluno), 1, paluno);
```

Arquivos

- **Posicionando o leitor**
  - As operações de leitura e gravação são feitas na posição onde o leitor se encontra no momento.
  - Podemos mudar a posição do leitor, colocando-o em um ponto específico do arquivo.

Arquivos

- **Posicionando o leitor**
  - fseek
    - Sintaxe:
      - `fseek(ponteiro_arquivo, numero_de_bytes, origem);`

Origem	Significado
SEEK_SET	O deslocamento do leitor será contado a partir do início do arquivo.
SEEK_CUR	O deslocamento do leitor será contado a partir da sua posição corrente.
SEEK_END	O deslocamento do leitor será contado a partir do final do arquivo.

Arquivos

- **Posicionando o leitor**
  - fseek
    - `fseek(paluno, 0,SEEK_END);`

    - `fseek(paluno, 2*sizeof(TAluno), SEEK_SET);`

Arquivos

- **Posicionando o leitor no início do arquivo**
  - fseek
    - `fseek(paluno, 0,SEEK_SET);`
  - rewind
    - **Sintaxe:**
      - `rewind(pont_arquivo);`
    - **Exemplo:**
      - `rewind(paluno);`

Arquivos

- **Verificando Fim de Arquivo**
  - **feof**
    - EOF ("End of file") indica o fim de um arquivo.
    - A função retorna um valor diferente de zero se o arquivo chegou ao fim, caso contrário retorna zero.
    - **Sintaxe**
      - `int feof(ponteiro_arquivo);`
    - **Exemplo:**
      - `while (feof(paluno)==0)`

Arquivos

- **Remover um arquivo**
  - **Sintaxe:**
    - `remove("nome_do_arquivo");`
  - **Exemplo:**
    - `remove("alunos.bin");`

Arquivos

- **Renomear um arquivo**
  - **Sintaxe:**
    - `rename("nome_arquivo", "novo_nome_arquivo");`
  - **Exemplo:**
    - `rename("alunos.bin", "alunos_temp.bin");`

Acabou!!!!