

RESUMO DAS PARTES IMPORTANTES DA MATERIA

BÁSICO DA LINGUAGEM

ESTRUTURA

```
1  /*
2   AUTOR: FRITZ
3   DATA: 31/08/2021
4   */
5
6 // comments
7
8 #include <stdio.h>
9
10 int main(void){
11     return 0;
12 }
13
14 }
```

TIPOS DE VARIÁVEIS

```
int i; //i é uma variável do tipo inteiro
char chl, novo; //chl e novo são vars do tipo char;
float pi, raio;
double total;
```

char: tipos usados para guardar caracteres, mas que também são usados para guardar números de 8 bits

int: números inteiros de 16 ou 32 bits, dependendo da implementação.

float: números reais IEEE de 32 bits.

double: números reais IEEE de 64 bits.

void: usada para criar procedimentos ou ponteiros genéricos. É o tipo nulo.

ESPECIFICADORES DE TIPO

(PRINTF)

```
7 #include <stdio.h>
8
9 int main(void) {
10     int idade = 25;
11     char nome[20] = "Fulano";
12     printf("0 %s tem %d anos\n", nome, idade);
13     return 0;
14 }
```

PARA CASAS DECIMAS

```
printf("0 preço é %.2f \n", preco);
```

especificador	tipo
%d	Número inteiro (int)
%c	Caractere (char)
%f	Números decimais (float e double)
%s	Cadeia de caracteres (char[])

SCANF

```
7 #include <stdio.h>
8
9 int main(void) {
10     float preco;
11     scanf("%f ", &preco);
12     printf("0 preço é %.2f \n", preco);
13     return 0;
14 }
```

& ("E" comercial)

ESTRUTURAS DE DECISÃO

operador	operação
<code>==</code>	Igualdade
<code>></code>	Maior que
<code><</code>	Menor que
<code>!=</code>	Diferente
<code>>=</code>	Maior ou igual
<code><=</code>	Menor ou igual

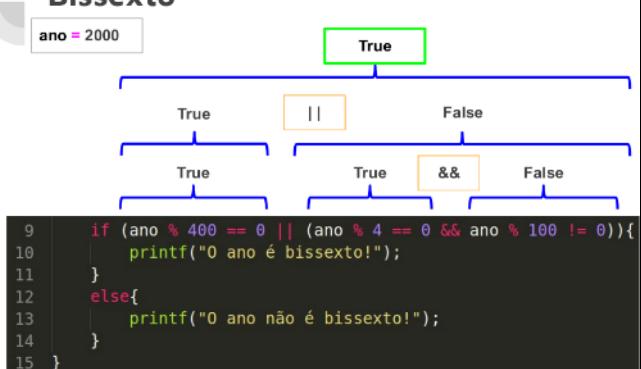
OPERADORES LÓGICOS

`&&` (E)

`||` (OU)

```
if (ano % 400 == 0 || (ano % 4 == 0 && ano % 100 != 0)){
    printf("O ano é bissexto!");
}
else{
    printf("O ano não é bissexto!");
}
```

Operadores Lógicos - Exemplo Ano Bissexto



IF

```
if (condição)
```

```
{ //instrução ou instruções para condição verdadeira;
}
```

IF - ELSE

```
if(soma > 10)
{
    printf("O valor da soma é maior que 10\n");
}
else
{
    printf("Valor menor ou igual a 10");
}
```

IF - ELSE IF - ELSE

```
int N1 = 2;
int N2 = 5;

if (N1 == N2)
    printf("Os numeros sao iguais!");
else if (N1 > N2)
    printf("O maior valor e = %d", N1);
else
    printf("O maior valor e = %d", N2);
```

SWITCH - CASE

```
switch(variavel){
    case valor1:
        // entra aqui caso variável = valor1
        break;
    case valor2:
        // entra aqui caso variável = valor2
        break;
    ...
    default:
        // entra aqui caso não tenha entrado em
        // nenhum case
}
```

`switch` testa a variável e executa a declaração cujo `case` corresponda ao valor atual da variável

`break` - faz a interrupção assim que uma declaração `for` executada

`default` - Opcional: executa se nenhum `case` for a resposta da variável

ESTRUTURAS DE REPETIÇÃO

WHILE

```
while(condição)
{
    //instruções dentro do loop;
}
```

```
1 #include <stdio.h>
2
3 int main(void){
4     int i = 1;
5     while(i <= 10){
6         printf("%d\n", i);
7         i++;
8     }
9 }
```

WHILE INFINITO

```
while(1){
    // código que vai executar infinitamente
}
```

USA BREAK PARA PARAR

FOR

```
for (valor_inicial; condição_final; valor_incremento)
{
    // instruções dentro do loop;
}
```

```
#include <stdio.h>

int main(void)
{
    int contador;
    for(contador = 1; contador <= 10; contador++)
    {
        printf("%d ", contador);
    }
    return(0);
}
```

PARA NÚMEROS ALEATÓRIOS

GERA 100 NÚMEROS ALEATÓRIOS - SEJA DE 1 A 100 OU DE 0 A 99

```
#include <stdlib.h> → rand() % 100;
```

O '% 100' É A CHAVE DE ANÁLISE. SE MUDAR, SERÁ ALTERADO A QUANTIDADE DE NÚMEROS GERADOS

ESSE COMANDO GERA VARIAS 'SEED', BASEADO NO TEMPO

```
#include <time.h> → srand(time(NULL));
```

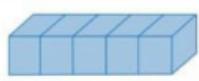
PODE SER USADO srand(seed), ESSA SEED SENDO UM VALOR QQ

DO - WHILE

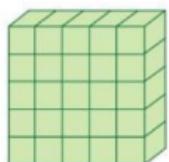
```
do{
    // instruções dentro do loop
}while(condição)
```

BLOCO DE INSTRUÇÃO É EXECUTADO PELO MENOS 1X

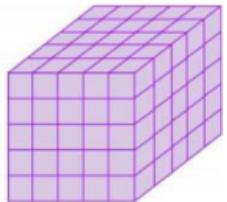
ARRAYS



vector



matrix



3-D Tensor

CONJUNTO FINITO DE ELEMENTOS HOMOGÊNEOS

VETORES

ARRANJO DE UMA DIMENSÃO

SÃO INDEXADOS

valores	2	5	1	2	9	45	3	2	1	1
índices	0	1	2	3	4	5	6	7	8	9

SINTAX

<tipo> nome_vetor [<tamanho>];

int v[100]; // vetor de inteiros com 100 elementos

float f[8]; // vetor de floats com 8 elementos

int idades[1000]; // vetor de int com 1000 elementos

FORMAS DE INICIALIZAR UM VETOR

```
void main(){
    int v[] = {0,0,0,0,0,0,0,0,0,0};
```

```
void main() {
    int v[10];
    //inicializando...
    int i;
    for(i=0;i<10;i++)
        v[i] = 0;
}
```

RESUMÃO PARA UTILIZAR

Declaração de vetores:

```
int vet[5]; // declara um vetor de inteiros de 5 posições
```

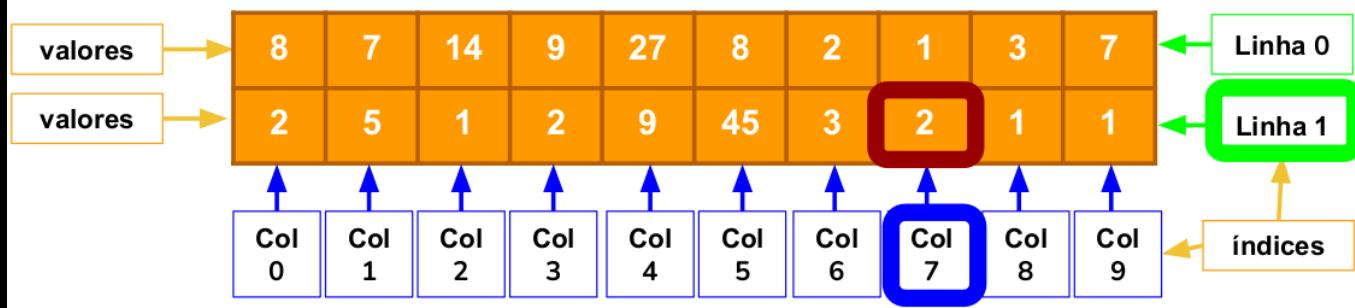
Atribuindo valor a uma posição do vetor “vet”:

```
vet[0] = 2; // atribui 2 na posição 0 (primeira) de "vet"
vet[4] = 9; // atribui 9 na posição 4 (última) de "vet"
```

Preenchimento de “vet” com números de 1 a 5:

```
for(int i=0; i < 5; i++)
    vet[i] = i+1;
```

- Exemplo: Qual é o valor no índice - linha 1 e coluna 7?



SINTAX

<tipo> nome_matriz [<tam_linha>][<tam_coluna>];

int v[100][100]; // matriz de int com 100 linhas e 100 colunas

float f[8][10]; // matriz de floats com 8 linhas e 10 colunas

ATRIBUIÇÃO DE VALORES

m[1][9] = -11; **int i = m[1][9];**

FORMAS DE INICIALIZAR UMA MATRIZ

```
int main(void){
    int mat[][][3] = {{1,2,3},{4,5,6},{7,8,9}};
}
```

```
int main(void){
    int m[3][3];
    int i,j;
    for(i=0;i<3;i++){
        for(j=0;j<3;j++) {
            m[i][j] = 0;
        }
    }
}
```

Preenchimento de “mat” com número 1 em todas as posições:

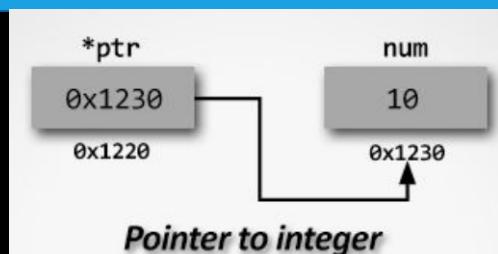
```
for(i=0; i < 5; i++)
    for(j=0; j < 3; j++)
        mat[i][j] = 1;
```

PONTEIROS

SABER ONDE ESTÁ LOCALIZADA UMA VARIÁVEL NA MEMÓRIA RAM

USADO PARA ARMAZENAR ESSE VALOR DE ENDEREÇO DE OUTRAS VARIÁVEIS

EXEMPLO:
PONTEIRO PTR TEM O ENDEREÇO DE NUM



```

1 #include <stdio.h>
2
3 int main(void) {
4     int age = 10;
5     printf("%p", &age);
6 }

```

./main
0x7ffd9aeb0e5c

%p = especificador de conversão para endereço

DECLARAÇÃO:
O TIPO DO PONTEIRO É O MESMO DA VARIÁVEL APONTADA

int *x_ponteiro;
 ↑
 Tipo da variável apontada Símbolo para declaração de ponteiro Nome da variável ponteiro

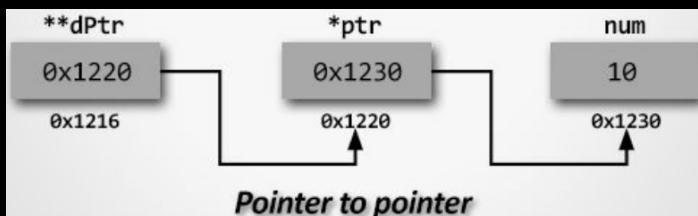
EXEMPLO: ATRIBUIR O PONTEIRO aPtr AO ENDEREÇO DE a

```

int a = 10; // declarando e inicializando "a"
int *aPtr; // declarando a variável ponteiro "aPtr"
aPtr = &a; // atribuindo o endereço de "a" para o ponteiro "aPtr"

```

PONTEIRO DE PONTEIRO:
APONTA PARA O ENDEREÇO DE OUTRO PONTEIRO



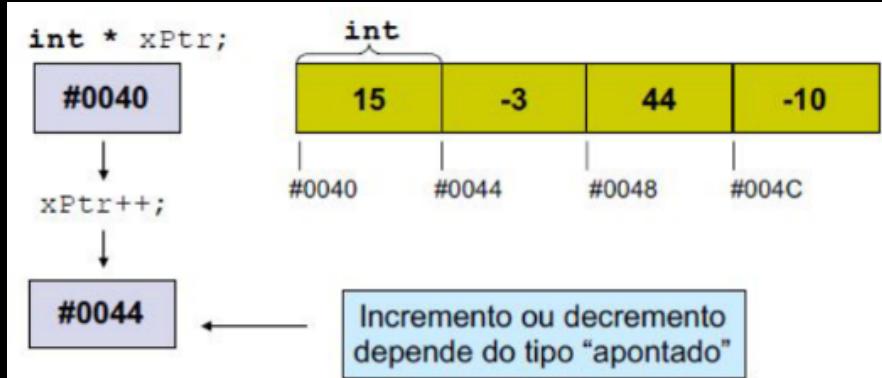
```

double x = 10.00;
double *xPtr;
xPtr = &x;
double **ptrXPtr;
ptrXPtr = &xPtr;

```

ARITMÉTICA DE PONTEIRO

PODE SER INCREMENTADO (++)
OU DECREMENTADO (--)



BOA PRÁTICA INICIALIZAR O PONTEIRO PARA O NADA (NULL)

int x, y, *p = NULL;

p++; // incrementa o ponteiro, ou seja o endereço

(*p)++; // incrementa o conteúdo apontado por p

***(++p);** // incrementa primeiro o ponteiro; depois acessa o valor da nova posição

TROCA INDIRETA DOS VALORES POR MEIO DE PONTEIROS

```
// declaração de ponteiros
int *pa, a = 10;
float *pb, b = 20.20;
char *pc, c = 'a';

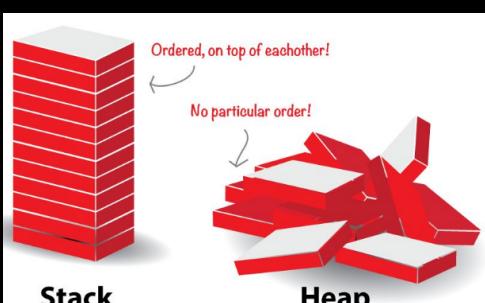
// pra onde os ponteiros apontam
pa = &a;
pb = &b;
pc = &c;

// troca indireta
*pa = 11;
*pb = 21.21;
*pc = 'c';
```

GERENCIAMENTO DE MEMÓRIAS

Stack (Alocação Estática):

- Gerenciamento de memória automática
- Funciona como uma pilha de pratos
- Acesso muito rápido
- Ocorre em tempo de compilação
- No momento em que se define uma variável é necessário que se definam seu tipo e tamanho



Heap (Alocação Dinâmica):

- Gerenciamento por conta do programador
- Variáveis podem ser acessadas globalmente
- Não tem limite de tamanho (permite redimensionamento)
- Acesso mais lento
- Ocorre em tempo de execução
- Variáveis são declaradas sem a necessidade de se definir seu tamanho, pois nenhuma memória será reservada ao colocar o programa em execução

STACK É A DECLARAÇÃO NORMAL QUE ESTAMOS FAZENDO, ELA ENTRA DO FINAL A MEMÓRIA PRA O INÍCIO. E NO FINAL DO PROGRAMA É ZERADA

O POSICIONAMENTO RECEBIDO É O DO PRIMEIRO SLOT DE MEMÓRIA

NA HEAP: PRECISA DE OUTRA BIBLIOTECA E DE FUNÇÕES PARA CHAMAR A ALOCAÇÃO E DESALOCAÇÃO

#include <stdlib.h>

A alocação e desalocação ocorrem através das chamadas: **malloc** e **free**

malloc - sizeof

Se não conseguir alocar, retorna **NULL**

```
int *ptr = malloc(sizeof(int));
```

sizeof - permite saber o número de bytes ocupado por um determinado tipo de variável.

Assim, para liberar a memória ocupada por essas variáveis, é preciso recorrer à função **free**

```
free(j);
```

```
int *j;  
j=malloc(4);  
*j = 4;
```

Exemplo: aloca o tamanho de 5 **int** (cria um **vetor** de ponteiros com 5 inteiros)

```
int *ptr = (int*)malloc(5*sizeof(int));
```

- **calloc**

- Podemos usar também a função **calloc**
- Faz a mesma coisa que **malloc**, mas a chamada é diferente

```
int *ptr = (int*)calloc(5, sizeof(int));
```

- **realloc**

- Altera dinamicamente a alocação de memória de uma memória previamente alocada

```
int *ptr = (int*)calloc(5, sizeof(int));
```

```
ptr = realloc(ptr, 10 * sizeof(int));
```

FUNÇÕES

Funções são conjuntos de instruções planejadas para cumprir uma tarefa específica

USADO PARA MODULARIZAR PROGRAMA

SINTAXE

```
tipoRetorno nomeFuncao(tipo1 param1, ..., tipoN paramN) {  
    //Corpo da função  
    return valor;  
} //Fim da função
```

O valor de retorno de uma função é dada pela instrução **return**

FUNÇÕES ANTES DO MAIN

```
1 #include <stdio.h>  
2  
3 int soma(int a, int b){  
4     int c = a+b;  
5     return c;  
6 }  
7  
8 void main(){  
9     int num = soma(12,20);  
10    printf("%i", num);  
11 }
```

FUNÇÕES DEPOIS DO MAIN

PRECISA DECLARAR UM PROTÓTIPO

```
1 #include <stdio.h>  
2  
3 int soma(int, int);  
4  
5 void main(){  
6     int num = soma(12,20);  
7     printf("%i", num);  
8 }  
9  
10 int soma(int a, int b){  
11     int c = a+b;  
12     return c;  
13 }
```

POR VALOR

PARÂMETROS RECEBEM CÓPIA DOS VALORES

```

1 int soma(int a, int b){
2     int c = a+b;
3     return c;
4 }
5 void main(){
6     int num = soma(12,20);
7     printf("%i", num);
8 }
```

POR REFERÊNCIA

PARÂMETROS RECEBEM UM ENDEREÇO DE UMA VARIÁVEL

O C não suporta passagem de valor por referência, porém podemos passar endereços!

```

1 void troca(int *a, int *b){
2     int c = *a;
3     *a = *b;
4     *b = c;
5 }
6 void main(){
7     int a=2, b=6;
8     int num = troca(&a,&b);
9     printf("%i", num);
10 }
```

Qualquer alteração no valor da variável vai ser percebida globalmente

Boa estratégia quando a função deveria retornar mais do que um valor (como visto, o return só permite o retorno de uma valor)

STRINGS

TABELA ASCII EM CÓDIGO

```
int i = 97;  
printf("%d: %c\n", i, i);
```

97: a

STRING É UM VETOR DE 'CHAR'

```
// declarando uma string de tamanho 10  
char minha_string[10];
```

Último caractere: '\0' → indica o final de uma **string**

Pode ser chamado de **terminador** da **string**

CHAR VS STRING

as **strings** são representadas por **aspas duplas** “ ”

enquanto **char** usam **aspas simples** ‘ ’

‘A’ != “A”

IMPRESSÃO DE STRINGS

PRINTF() -- ARGUMENTO %S

```
// imprime "minha string" na tela  
printf("%s", "minha string"); // ou  
printf("minha string");  
  
// imprime o conteúdo da variável string nome  
printf("%s", nome);  
  
// imprime o conteúdo das variáveis strings dia, mes e ano  
printf("%s de %s de %s", dia, mes, ano);
```

PUTS()
ADICIONA LINHA AUTOMÁTICO
SÓ UMA STRING POR VEZ

```
// imprime "minha string" na tela  
puts("minha string");  
// imprime o conteúdo da variável string nome  
puts(nome)
```

LEITURA DE STRINGS

SCANF()
TERMINA DE ARMAZENAR NO PRIMEIRO
ESPAÇO
NÃO PRECISA DO '&'

```
// leitura de valor para a variável nome  
scanf("%s", nome);
```

GETS()
PERIGOSO PQ PODE PEGAR MEMÓRIAS
QUE NÃO DEVERIA

```
// leitura de valor para a variável nome  
gets(nome);
```

FGETS()
MELHOR USAR ELE PARA FRASES

```
// leitura de valor para a variável nome  
fgets(nome, 10, stdin);
```

EXEMPLO DE UTILIZAÇÃO

MELHOR DECLARAR COMO PONTEIRO

```
#include <stdio.h>

int main(void){
    char *str = "declaracao como ponteiro para char";
    printf("%s", str);
    return 0;
}
```

PS C:\Users\daniel\Documents> .\string2
declaracao como ponteiro para char
PS C:\Users\daniel\Documents>

BIBLIOTECA - STRING.H

FUNÇÕES

`int strlen (char*);`

SEM TERMINADOR

- Retorna o comprimento de uma string sem contar seu terminador ('\0')

`char * strcat (char *, char*);`

CONCATENA

- Concatena a segunda string na primeira

`int strcmp (char *, char *);`

COMPARA

- Compara strings. Retorna 0, negativo, ou positivo se forem iguais, se a primeira for menor (alfabeticamente) que a segunda ou a primeira for maior (alfabeticamente) que a segunda, respectivamente.

`char *strupr (char*);`

MAIUSCULA

- Converte e retorna a string recebida em maiúsculos

`char *strlwr (char *);`

MINUSCULA

- Converte e retorna a string recebida em minúsculos.

`int sprintf (char * str, const char * format, ...);`

QUANTIDADE

- Imprime o mesmo texto que seria impresso com printf porém na string str. Retorna o número total de caracteres escritos em str.

`int sscanf (const char * str, const char * format, ...);`

LEITURA

- Lê a string str assim como o scanf faria com a entrada padrão. Retorna o número de items que foram lidos.

COPIA

`char * strcpy (char *, char *);`

- Copia a segunda string na primeira

VETOR DE STRINGS

`Vetores de ponteiros para char`

```
char *strings[12];
strings[0] = "Janeiro";
strings[1] = "Fevereiro";
...
strings[11] = "Dezembro";
```

REGISTROS - STRUCT

COMO DECLARAR

```
struct nome_struct {  
    tipo nome_da_variável;  
    tipo nome_da_variável;  
    tipo nome_da_variável;  
};
```

EXEMPLO

```
struct sPonto {  
    int x;  
    int y;  
};
```

DECLARAÇÃO DE UMA VARIÁVEL COM ESSE TIPO STRUCT

```
int main(){  
    struct sPonto meuPonto;  
}
```

ARMAZENA DADOS DE TIPOS DIFERENTES

```
struct pessoa {  
    char nome[50];  
    int idade;  
    char sexo;  
};
```

PODE SER FEITO UM VETOR DE REGISTROS, TENDO ASSIM UM VETOR COM AQUELES TIPOS DECLARADOS

```
struct pessoa agenda[100];
```

USAR FUNÇÕES E ATRIBUIR VALORES

```
3 struct pessoa {  
4     char nome[50];  
5     int idade;  
6     char sexo;  
7 };  
8  
9 void leiaDados(struct pessoa *p){  
10    printf("Digite o nome: ");  
11    scanf("%s", (*p).nome);  
12    printf("Digite a idade: ");  
13    scanf("%d", &(*p).idade);  
14    printf("Digite o sexo: ");  
15    scanf(" %c", &(*p).sexo);  
16 }  
17  
18 int main(void){  
19     struct pessoa agenda[100];  
20     leiaDados(&agenda[0]);  
21     printf("Dados: %s %d %c", agenda[0].nome, agenda[0].idade, agenda[0].sexo);  
22     return 0;  
23 }
```

structs e funções

- Podemos passar estruturas inteiras por **referência para funções**

Função `leiaDados` tem 1 parâmetro: um ponteiro de `struct pessoa`.

Chamada da função - o endereço de uma pessoa é enviado

strcpy - copy string ←
Depois de inicializados não podemos alterar o valor de um vetor inteiro

strcpy atribui a string para o vetor de char

Precisa da biblioteca:
`#include <string.h>`

PARA ACESSAR OS VALORES DESSAS VARIÁVEIS

```
strcpy(p1.nome, "Fulano");  
strcpy(p2.nome, "Sicrana");  
strcpy(p3.nome, "Beltrano");  
  
p1.idade = 26;  
p2.idade = 30;  
p3.idade = 18;  
  
p1.sexo = 'M';  
p2.sexo = 'F';  
p3.sexo = 'M';
```

USAR NO PRINT

```
printf("Nome de p1: %s\n", p1.nome);  
printf("Nome de p2: %s\n", p2.nome);  
printf("Nome de p3: %s\n", p3.nome);
```

TEMOS O OPERADOR SETA TBM QUE FACILITA A ESCRITA DESSA PASSAGEM POR REFERÊNCIA

```
3 struct pessoa {  
4     char nome[50];  
5     int idade;  
6     char sexo;  
7 };  
8  
9 void leiaDados(struct pessoa *p){  
10    printf("Digite o nome: ");  
11    scanf("%s", p->nome);  
12    printf("Digite a idade: ");  
13    scanf("%d", &p->idade);  
14    printf("Digite o sexo: ");  
15    scanf(" %c", &p->sexo);  
16 }  
17  
18 int main(void){  
19     struct pessoa agenda[100];  
20     leiaDados(&agenda[0]);  
21     printf("Dados: %s %d %c", agenda[0].nome, agenda[0].idade, agenda[0].sexo);  
22     return 0;  
23 }
```

structs e funções

Usando o operador `->`, no lugar de `*` e `.`