Prova 1 → Arquivos

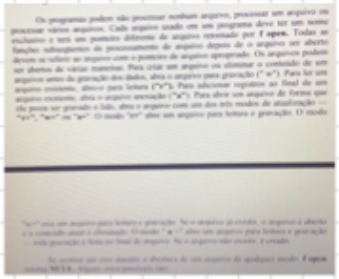
→ Dados permanentes

→ Bit → Byte → Campo (field) → Registro (word) → Arquivo (file)

Nome	Matrícula	IRA
Thales	0024885	10
Victor	222 0029084	10

→ registro } arquivo

* Campo-chave (ex: nome)
  ↳ identifica o registro



→ Ponteiros de arquivos: stdin, stout, stderr

r+	w+	a+
↓	↓	↓
null	new	new
read	write	add → Final do arquivo
↑	↑	↑
r	w	a

→ fopen
  ↳ Abre o arquivo
  ↳ Argumento: nome do arquivo e parâmetro

→ fclose
  ↳ fecha o arquivo
  ↳ argumento: porteiro do arquivo

→ fgetc
  ↳ lê um caractere de um arquivo
  ↳ Argumento: Porteiro FILE do arquivo

* fgets(s, 79, FP): lê string
  ↳ lê 79 bytes do arquivo portado pelo pont FP e escreve no endereço de memória 's'

→ fputc/fputs
  ↳ escreve caractere/string num arquivo
  ↳ argumento: caractere/string a ser escrito e um porteiro para o arquivo

→ EOF: caractere especial do ASC II de valor 26 usado para identificar o fim do arquivo

→ fprint
  grava dados no arquivo
  argumento: porteiro do arquivo, tipo de leitura (% ?) e variáveis

→ fscanf
  ↳ lê o arquivo
  ↳ Argumento: Porteiro do arquivo, tipo de leitura (% ?) e variáveis

→ tipos de variáveis
  ↳ % d - int
  % f - float
  % Lf - long float
  % p - int hexadecimal - porteiros
  % c - char
  % s - string
  % u - unsigned char (0)
  % 2f - duas casas decimais

```c
#include <stdio.h>
main( )
{
  FILE *fp ;          // arquivo usado
  char ch ;
  fp = fopen ( "prog1.c", "r" ) ;
  while ( 1 )
  {
    ch = fgetc ( fp ) ;
    if ( ch == EOF )
      break ;
    printf ( "%c", ch ) ;
  }
  fclose ( fp ) ;
}
```

↳ 4: afirma que FP é' ponteiro para uma estrutura FILE
↳ 5: declara a variável ch do tipo char
↳ 6: Abre o arquivo prog1.c e lê com o modulo r (lê, e se não existe retorna null)
↳ 9: Variavel ch vai pegar os caracteres que fgetc esta lendo
↳ 10: quando fgetc ler o EOF o programa para
↳ 14: Fecha o programa

Ponteiros

→ Variáveis que contém endereços de memória como valores
↳ Ponteiro - endereço de memória → variavel - valor
↳ Variável: referência direta ao valor
Ponteiro: referência indireta

→ int * contPtr, cont;
↳ contPth: ponteiro para um int
↳ cont: variável tipo int

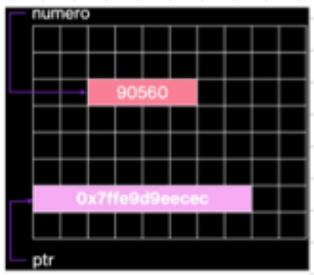
→ Devem ser inicializados

→ Ponteiro com valor NULL não aponta para lugar nenhum

→ Operador de ponteiro &
↳ int y = 5;
int *yPtr;

yPtr = &y; → Atribuição do endereço de y à yPtr → yPtr "aponta" para y

```c
#include <stdio.h>
int main() {
    int numero, *ptr;
    numero = 79417;
    ptr = &numero;
    *ptr = 90560
    return 0;
}
```



Área de memória do programa.

→ Operador de desreferenciamento *
- printf ("%d", *yPtr) → imprime o valor da variável que o ponteiro aponta (5). (desreferencia um ponteiro)

↳ se printf("%d", yptr) → imprime o endereço de memória yptr

→ void *: ponteiro generico que pode representar qualquer tipo de ponteiro aponta (5)
↳ não pode ser desreferenciado

→ Array e ponteiros:
int b[5], bPtr;
bPtr = b → nome do array = pont. para o 1º elemento
bPtr = &b[0]

→ elemento b[3]: *(bPtr + 3)
↓
offset (deslocamento)

* (b+3)
''

Alocação Dinâmica

→ Controle de uso de memória: uso sob demanda

→ # include <stdlib.h>: biblioteca de malloc, calloc, Free

newPtr = malloc(sizeof(struct node));

processa **sizeof (struct node)** para determinar o tamanho em bytes **de** uma estrutura do tipo **struct node,** aloca uma nova área **de sizeof (struct node)** na memória e armazena na variável **newPtr** um ponteiro para a memória alocada. Se não houver memória disponível, **malloc** retorna um ponteiro **NULL.**

Alocação de memória

→ Malloc
↳ Argumento: nº de bytes a serem alocados
↳ retorna: ponteiro do tipo void*
↳ tipo *nome = (tipo*) malloc (sizeof (tipo) * tamanho);
- int * vet1;
vet1 = (int *) malloc (100 * sizeof (int));
↳ Aloca uma quantidade específica de memória, mas não inicializa.

→ Calloc
↳ aloca uma quantidade e inicializa toda a memória alocada com zero
↳ tipo *nome = (tipo *) calloc (tamanho, sizeof (tipo));
- int * vet2;
vet2 = (int *) calloc (100, sizeof (int)).

Alocação

→ Free
↳ libera memória alocada
↳ void free (void * ptr);
- free (vet1)

Deslocação

→ New
↳ tipo * nome = new tipo [tamanho];
- float * pvetor; int ptam;
pvetor = new float [ptam];

→ Delete
↳ delete [ ] vetor;
- delete [ ] p vetor.

Estruturas

→ Estrutura de dados composta que define fisicamente uma lista de variáveis agrupadas sob um nome em um bloco de memória

→

```
struct Livros {
  char titulo[50];
  char autor[50];
  char assunto[100];
  int id_livro;
};
/* Declarando Livro1 e Livro2 do tipo Livros */
struct Livros Livro1, Livro2;
```

```
// inicializando
struct Livros Livro1 = { "Titulo genérico", "Blog Trybe",
"Um livro bem genérico", 6495407 };
```

```
/* Declarando Livro1 do tipo Livro */
struct Livros Livro1;
strcpy( Livro1.titulo, "Titulo genérico"); // inicializando os valore
para cada variável separada
strcpy( Livro1.autor, "Blog Trybe");
strcpy( Livro1.assunto, "Um livro bem genérico");
Livro1.id_livro = 6495407;
```

```
/* mostrando as informações do livro 1 */
  printf( "Livro 1 titulo : %s\n", Livro1.titulo);
  printf( "Livro 1 autor : %s\n", Livro1.autor);
  printf( "Livro 1 assunto : %s\n", Livro1.assunto);
  printf( "Livro 1 id_livro : %d\n", Livro1.id_livro);

  /* mostrando as informações do livro 2 */
  printf( "Livro 2 titulo : %s\n", Livro2.titulo);
  printf( "Livro 2 autor : %s\n", Livro2.autor);
  printf( "Livro 2 assunto : %s\n", Livro2.assunto);
  printf( "Livro 2 id_livro : %d\n", Livro2.id_livro);
```

→ Typedef: usado para renomear um tipo de dado.

Para acessar os campos de uma struct, usa-se a sintaxe **NomeDaVariavel.NomeDoCampo**, conforme o exemplo a seguir.

```
Joao.Idade = 15;
Joao.Peso = 60.5;
Joao.Altura = 1.75;

Povo[4].Idade = 23;
Povo[4].Peso = 75.3;
Povo[4].Altura = 1.89;
```

* Outra vantagem de utilizar struct é a possibilidade de atribuir os dados de uma struct para outra, com apenas um comando de atribuição, como neste exemplo:

```
P2 = Povo[4];
```

```
typedef struct
{
  float Peso;  // define o campo Peso
  int Idade;   // define o campo Idade
  float Altura; // define o campo Altura
} Pessoa; // Define o nome do novo tipo criado
Após a criação do tipo, é possível declarar variáveis do
tipo Pessoa, desta forma:

Pessoa Joao, P1, P2;
Pessoa Povo[10]; // cria um vetor de 10 pessoas.
```

→ Struct Pessoa * p;
p -> nome, d -> idade;

(=p. nome mas para ponteiro)

Recursividade

→

- Uma função recursiva é uma função que resolve uma parte pequena de um problema, e que, para resolver o resto do problema, chama ela mesma.
- O conceito é matemático mas se aplica muito bem à programação. Um exemplo vindo da matemática é a definição dos números Naturais (inteiros positivos):
  - 0 é um inteiro natural
  - Se x é um inteiro natural, x+1 é um inteiro natural

- Quando uma função chama ela mesma, um novo espaço é reservado na pilha para a execução da nova função, e o ponteiro do programa, que indica qual linha do programa está sendo executado, pula para o início da função.
- O código executado é o mesmo, mas com um conjunto de variáveis e parâmetros novos, num espaço novo na pilha.
- Na hora do return, a função é desempilhada e volta para onde estava, descartando suas variáveis e parâmetros.
- O programa continua onde tinha parado com suas variáveis e parâmetros antigos.

- A recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos;
- Recursividade vale a pena para Algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha. Exemplos:
  - Dividir para Conquistar (Ex. Quicksort)
  - Caminhamento em Árvores (pesquisa, backtracking)

2> Fazer a expansão da função Ackermann:

$$A(m,n) = \begin{cases} n+1, & \text{se } m=0 \\ A(m-1,1), & \text{se } m>0 \text{ e } n=0 \\ A(m-1, A(m,n-1)), & \text{se } m>0 \text{ e } n>0. \end{cases}$$

Ex: $A(1,2) = A(0, A(1,1))$
$= A(0, A(0, A(1,0)))$
$= A(0, A(0, A(0,1)))$
$= A(0, A(0,2))$
$= A(0,3)$
$= 4.$

Acesse o repo abaixo e veja exercícios e provas dos semestres passados.

https://github.com/thaleseuflauzino/EDA-1