ED62A-COM2A
ESTRUTURAS DE DADOS

Aula 02 - Revisão de Ponteiros, Alocação Dinâmica de Memória, Recursividade e Tipos Abstratos de Dados, Arquivos

Prof. Rafael G. Mantovani

19/03/2019

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Apucarana - PR, Brasil
Março, 2019

Universidade Tecnológica Federal do Paraná (UTFPR)
Engenharia de Computação

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Ponteiros

- O que são **ponteiros**/**apontadores**?

Ponteiros

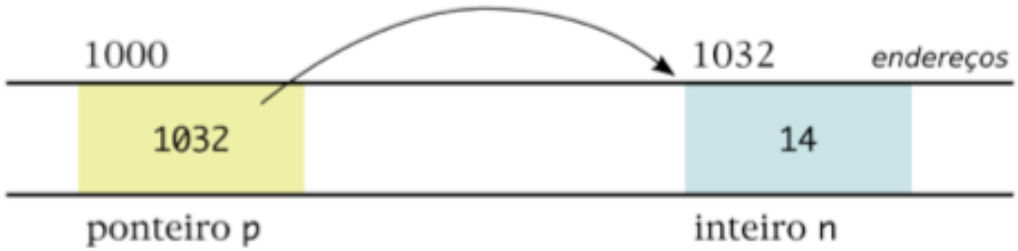
- Considerando uma variável declarada como:

```
int* p;
```

- ***p*** é um ponteiro para **int**, isto é, uma variável que **armazena o endereço de uma variável** do tipo **int**.
- Supondo que ***p*** armazene o valor 1032, tem-se que:

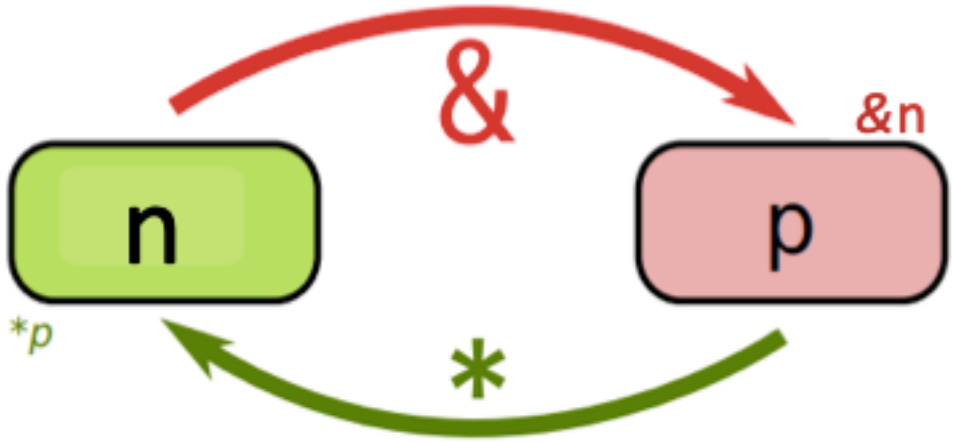
```
p = &n;
```

- Define-se **p* como sendo o valor contido na posição de memória apontada por ***p***. Assim, **p* vale 14.



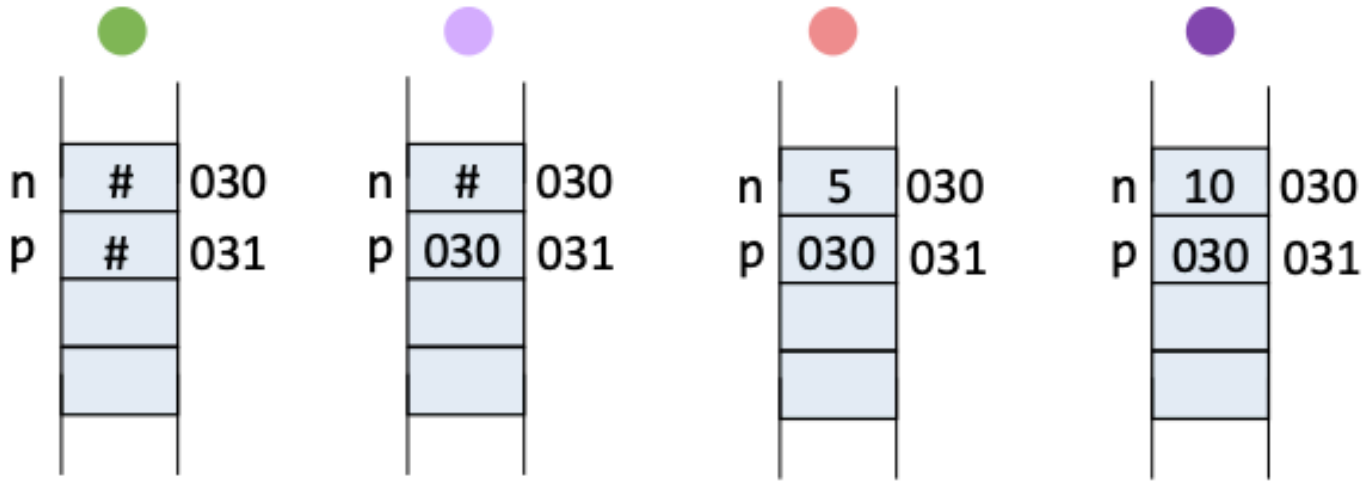
Ponteiros

- Para acessar a variável que é apontada por um ponteiro, usamos o operador * (o mesmo asterisco usado na declaração)
  - Se **p** é um ponteiro, podemos acessar a variável para a qual ele aponta com ***p**;
  - Esta expressão pode ser usada tanto para ler o conteúdo da variável quanto para alterá-lo.



Ponteiros

```
1  int main(){
2      int n, *p;
3      p = &n; //p aponta para a variável n
4
5      *p = 5;
6      printf("n = %d", n); //imprime 5
7      n = 10;
8      printf("*p = %d", *p); //imprime 10
9
10     return 0;
11 }
```



Exercícios - Ponteiros

1. Compile e execute os seguintes programas:

```c
int main (void) {
    typedef struct {
        int dia, mes, ano;
    } data;
    printf ("sizeof (data) = %d\n",
             sizeof (data));
    return EXIT_SUCCESS;
}

int main (void) {
    int i = 1234;
    printf (" i = %d\n", i);
    printf ("&i = %ld\n", (long int) &i);
    printf ("&i = %p\n", (void *) &i);
    return EXIT_SUCCESS;
}
```

Exercícios - Ponteiros

2. Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função **hm** que converta minutos em horas-e-minutos. A função recebe um inteiro **mnts** e os endereços de duas variáveis inteiras, digamos **h** e **m**, e atribui valores a essas variáveis de modo que **m** seja menor que 60 e que 60*h + m seja igual a **mnts**. Escreva também uma função **main** que use a função **hm**.

Roteiro

1. Ponteiros
2. **Alocação Dinâmica de Memória**
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Alocação dinâmica de memória

- Quando você declara um vetor em um programa em C, você deve informar quantos elementos devem ser reservados.
  - Se esse número de elementos é conhecido a priori, é trivial
  - Caso contrário, deve-se definir um tamanho máximo para acomodar os dados

- **Desperdício de memória**: caso poucos valores forem armazenados no vetor
- **Falta de Memória**: caso o vetor declarado seja insuficiente

- **Solução**: é usar **ALOCAÇÃO DINÂMICA**

Alocação dinâmica de memória

- **Alocação Dinâmica**: é o processo de solicitar e usar memória durante a execução de um programa;

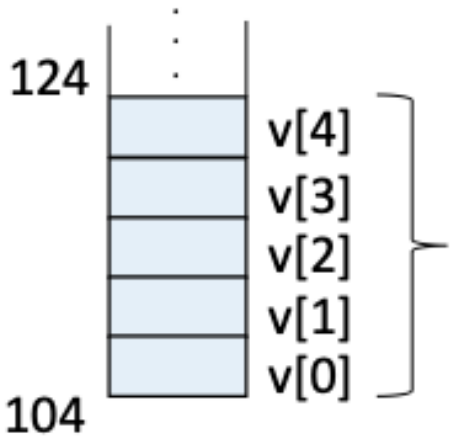
- Aplicada para que um programa utilize apenas a memória necessária para a sua execução, sem desperdício de memória;

- Sendo assim, deve-se ser usada quando não se sabe, por algum motivo, quanto espaço de memória será necessário para o armazenamento de um ou mais valores.

Na alocação estática ...

- Quando declaramos um vetor v com 5 posições para armazenar inteiros, o compilador reserva na pilha de memória: 5 espaços de memória para armazenar esses valores inteiros;
- Os valores são armazenados sequencialmente, formando um bloco **contínuo**
  - Por isso, podem ser acessados por meio de um índice

Se cada **int** ocupa 4 bytes, então: 4 bytes * 5 = 20 bytes contínuos na pilha



Lembre-se que v[5] não existe. Estaria sobrescrevendo o valor de uma outra variável armazenada antes (ou depois) do vetor

Na alocação estática ...

- Alocação estática é feita em "tempo de **compilação**"
  - Todo espaço de memória usado pelo programa é definido durante a compilação
  - Nenhum espaço extra para as variáveis pode ser requerido durante a execução

```
1  int typedef struct{
2       char nome[30];
3       int idade;
4  }Pessoa;
5  int main(){
6       int n;
7       char a;
8       float num;
9       int v[5];
10      Pessoa p;
11
12      return 0;
13 }
```

Variáveis alocadas estaticamente

Alocação dinâmica de memória

- Alocação dinâmica é feita em "tempo de **execução**"
  - Durante a execução do programa, mais ou menos memória pode ser utilizada baseada na demanda da aplicação

- No padrão ANSI C, existem 4 funções para se utilizar na alocação de memória:
  - malloc - *memory allocation*
  - calloc - *memory allocation and initialization*
  - realloc - *reallocation*
  - free - *free the memory*

- Todas essas funções pertencem à biblioteca **stdlib.h**

malloc (*memory allocation*)

```
void *malloc(num_bytes);
```

- Esta função recebe como parâmetros **num_bytes** correspondente de bytes consecutivos que deseja alocar
- O retorno é um ponteiro **void**, podendo ser atribuído a qualquer tipo de ponteiro;
  - o ponteiro indica a posição na memória em que se inicia o bloco de memória alocada
  - Caso o retorno seja **NULL**, a memória não pode ser alocada

```
1  int main(){
2       char *ptr;
3       ptr = malloc(1); //aloca 1 byte, ptr aponta para esse byte
4
5       return 0;
6  }
```

malloc (*memory allocation*)

```
void *malloc(num_bytes);
```

- Se precisarmos alocar memória para uma estrutura complexa, pode-se usar o operador **sizeof**, que diz quantos bytes tem a estrutura;

```
1  typedef struct{
2       int dia, mes, ano;
3  }Data;
4
5  int main(){
6       Data *d;
7       d = malloc(sizeof(Data)); //aloca memoria para armazenar uma variável Data
8       ...
9
10      return 0;
11 }
```

malloc (*memory allocation*)

- O ponteiro de retorno da função malloc é genérico:
  - pode ser convertido automaticamente para o tipo apropriado
  - ou pode ser convertido explicitamente se o programador assim desejar

```
1  int main(){
2     int *vet;
3
4     vet = malloc(10*sizeof(int));
5     ...
6
7     return 0;
8  }
```

```
1  int main(){
2     int *vet;
3
4     vet = (int *)malloc(10*sizeof(int));
5     ...
6
7     return 0;
8  }
```

São equivalentes!

free

```
void free(void* ptr);
```

- As variáveis alocadas estaticamente dentro de uma função (variáveis locais), desaparecem assim que a execução da função termina
- Variáveis alocadas dinamicamente continuam a existir mesmo depois que a execução da função termina;
- A função free desloca a porção de memória alocada;
- Recebe como parâmetro o ponteiro para a região de memória a ser desalocada:
  - se o ponteiro não apontar para uma região previamente alocada, o comportamento é indefinido
  - se o ponteiro estiver definido como NULL, a função não faz nada

free

```
void free(void* ptr);
```

```
1  int main(){
2     int *vet;
3
4     vet = (int *)malloc(10*sizeof(int));
5     ...
6     free(vet);
7
8     return 0;
9  }
```

Exemplo

- Alocando um vetor de inteiros dinamicamente e calculando a soma de seus elementos

```
1  int main(){
2     int *vet, tam, i, soma=0;
3
4     printf("Informe o tamanho do vetor: ");
5     scanf("%d", &tam);
6
7     vet = (int *)malloc(tam*sizeof(int));
8     for(i=0;i<tam;i++){
9         printf("informe o vet[%d]: ", i);
10        scanf("%d", &vet[i]);
11    }
12    for(i=0;i<tam;i++)
13       soma += vet[i];
14
15    printf("A soma dos elementos do vetor e' %d", soma);
16    free(vet);
17
18    return 0;
19 }
```

Alocando memória para um vetor do tamanho escolhido pelo usuário

Desalocando memória usada pelo vetor

Exercício - Alocação Dinâmica

1. Faça um programa que leia um valor **N** e crie dinamicamente um vetor de **N** elementos e passe esse vetor para uma função que vai ler os elementos desse vetor. Depois, no programa principal, o vetor preenchido deve ser impresso. Além disso, antes de finalizar o programa, deve-se liberar a área de memória alocada.

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Recursividade

- O que é uma função **recursiva**?

Recursividade

- função é dita **recursiva** quando dentro do seu código existe uma **chamada para si mesma**.

```
1  int fatorial(int n){
2      int fat;
3      if(n <= 1)
4          return 1;
5      else{
6          fat = n * fatorial(n-1);
7          return fat;
8      }
9  }
10
11 int main(){
12      int n=3, resultado;
13      resultado = fatorial(n);
14      printf("%d", resultado);
15
16     return 0;
17 }
```

Recursividade

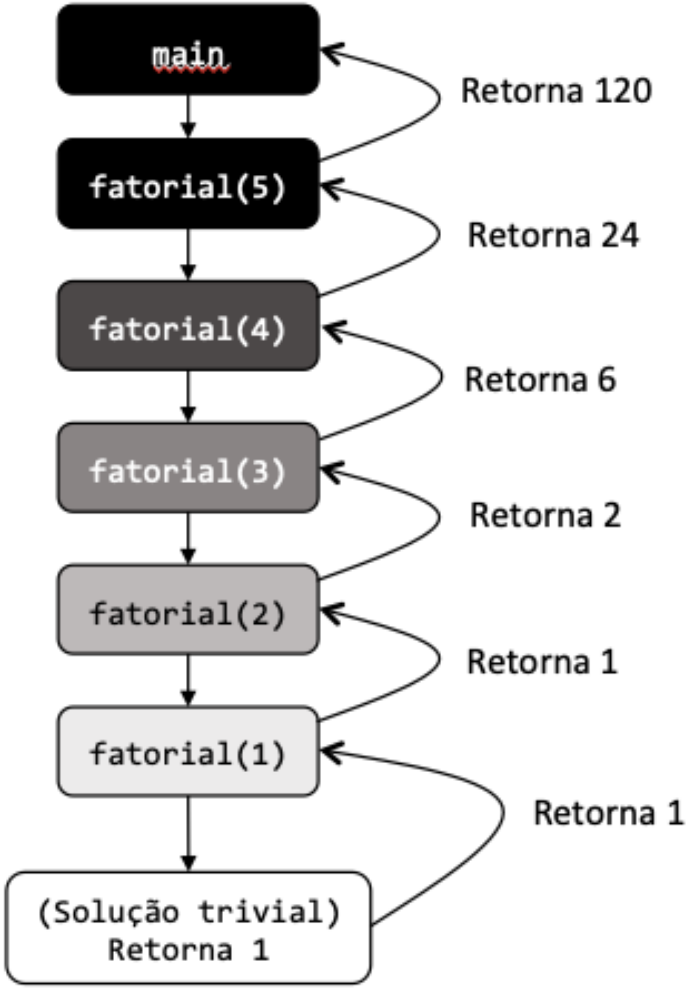
- **recursão** é uma técnica que define um problema em termos de um ou mais versões menores deste mesmo problema;
- portanto, pode ser utilizada sempre que for possível expressão a solução de um problema em função do próprio problema.

Recursividade

```
1  int fatorial(int n){
2      if(n == 0)
3          return 1;
4      else if(n < 0){
5          exit(0);
6      }
7      return n * fatorial(n-1);
8  }
9
10 int main(){
11     int n=3, resultado;
12     resultado = fatorial(n);
13     printf("%d", resultado);
14
15     return 0;
16 }
```

```
fatorial(5)
  => return 5 * fatorial(4)
     => return 4 * fatorial(3)
        => return 3 * fatorial(2)
           => return 2 * fatorial(1)
              => return 1 * fatorial(0)
                 => 0 == 0
                 <= return 1
              <= return 1 * 1 ------> (1)
           <= return 2 * 1 ------> (2)
        <= return 3 * 2 ------> (6)
     <= return 4 * 6 ------> (24)
  <= return 5 * 24 ------> (120)
```

Recursividade




```
fatorial(5)
  => return 5 * fatorial(4)
     => return 4 * fatorial(3)
        => return 3 * fatorial(2)
           => return 2 * fatorial(1)
              => return 1 * fatorial(0)
                 => 0 == 0
                 <= return 1
              <= return 1 * 1 ——→  (1)
           <= return 2 * 1 ——→  (2)
        <= return 3 * 2 ——→  (6)
     <= return 4 * 6 ——→  (24)
  <= return 5 * 24 ——→  (120)
```

Recursividade

- Em uma função recursiva pode ocorrer um problema de terminação do programa, como um loop infinito;

- Para determinar a terminação das repetições, deve-se:
  - definir uma função que implica em uma condição de terminação (**solução trivial**)
  - Provar que a função **decresce** a cada iteração, permitindo que, eventualmente, esta solução trivial seja atingida;

Recursividade

- Exemplo: função que recebe um parâmetro **par**

funcao(**par**)

- Teste de término de recursão utilizando **par**
  - Se teste for verdadeiro, retornar a solução final
- Processamento
  - Aqui a função processa as informações baseado em **par**
- Chamada recursiva utilizando **par**
  - **par** deve ser modificado para fazer a recursão chegar a um término

Exercícios - Recursividade

1. Escreva uma função recursiva para calcular o valor de uma base x elevada a um expoente y;

2. Usando recursividade, calcule a soma de todos os valores de um array de inteiros;

Exercícios - Recursividade

1. Escreva uma função recursiva para calcular o valor de uma base x elevada a um expoente y;

   Solução trivial: $x^0=1$

   Passo da recursão: $x^n = x * x^{n-1}$

2. Usando recursividade, calcule a soma de todos os valores de um array de inteiros;

   Solução trivial: Tamanho do array = 0. Soma é 0.

   Passo da recursão: v[n-1] + soma do restante do *array*

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Tipos Abstratos de Dados

- Em cada parte de um programa geralmente há várias variáveis associadas à realização de uma tarefa específica;
- É muito conveniente ter uma modo de agrupar um conjunto de variáveis relacionadas
- Vetores e matrizes agrupam uma séria de variáveis do **MESMO TIPO**, cada uma identificada por índices;
- Se, por outro lado, quisermos um agrupamento que englobe variáveis de **TIPOS DIFERENTES**, usamos **STRUCTS**.

Struct

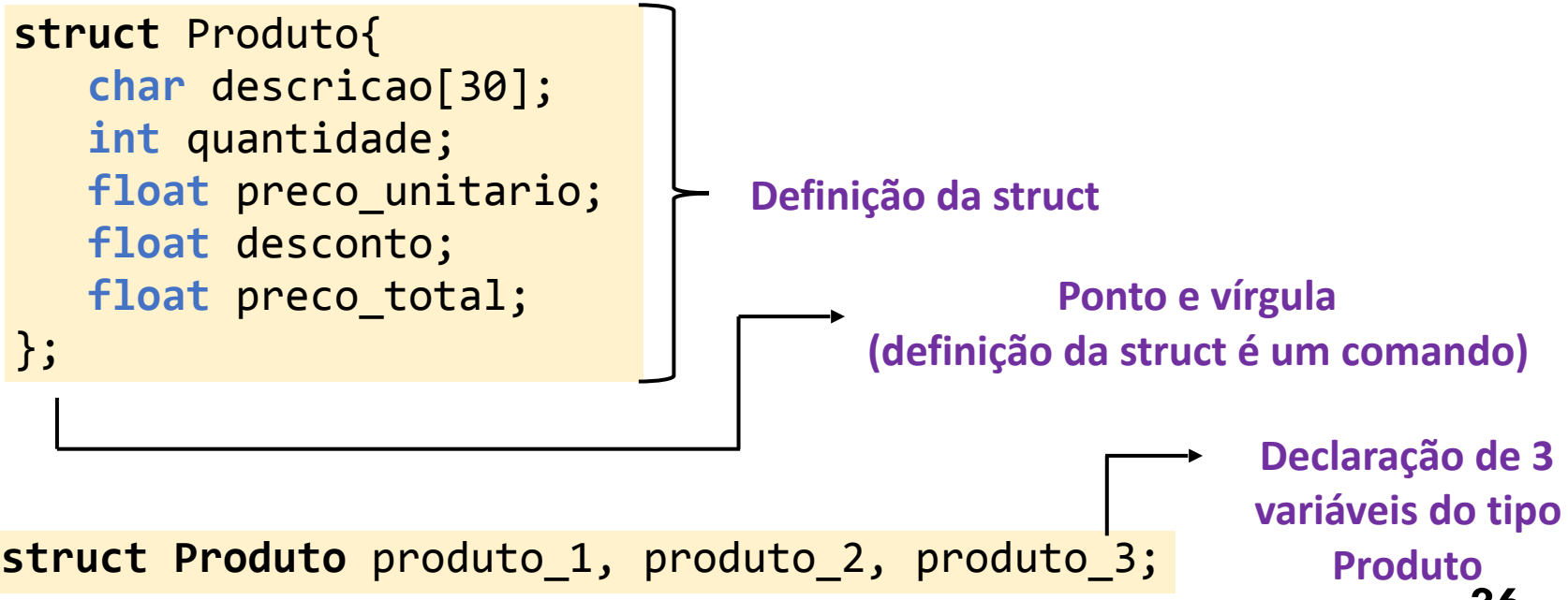
- Usamos um tipo de estrutura chamado de **registro** (mais conhecido por seu nome em inglês, ***struct***, uma abreviação de *structure*, 'estrutura');

- Esse recurso da linguagem C permite que o usuário "defina" **seus próprios tipos de dados** a partir dos tipos primitivos da linguagem (*int*, *float*, *char*, etc.);

- Struct contém um conjunto de variáveis, que têm tipos fixados e são identificadas por nomes (como as variáveis comuns);

Struct

- Uma **struct/registro** é declarada usando a palavra chave **struct** seguida de um bloco (delimitado por chaves) contendo as declarações dos membros, como se fossem declaração de variáveis comuns.



Acesso aos membros de uma struct

- Para acessar campos de um registro, usamos o operador . (um ponto), colocando à esquerda dele o nome da variável que contém o registro, e à direita o nome do campo.
- No exemplo anterior:

```
struct Produto{
   char descricao[30];
   int quantidade;
   float preco_unitario;
   float desconto;
   float preco_total;
};
```

```
struct Produto produto_1, produto_2, produto_3;
```

- Pode-se acessar o preço do produto_1 usando a expressão:

```
produto_1.preco_unitario
```

struct

- Declaração de um tipo de registro

```
struct nome_do_tipo{
    /* declarações dos membros */
};
```

- Declaração de um registro

```
nome_da_struct nome_da_variável;
```

- Acesso de membros de um registro

```
variavel.nome_do_membro;
```

Uso de structs

- Uma variável estrutura pode ser atribuída a outra do mesmo tipo por meio de uma atribuição simples

```
struct Produto{
   char descricao[30];
   int quantidade;
   float preco_unitario;
   float desconto;
   float preco_total;
};

struct Produto feijao = {“redondo”, 1, 20.0, 0, 20.0};
struct Produto feijao_carioca;

feijao_carioca = feijao;
```

Atribuição só pode ser feita com **structs** do mesmo tipo

structs

- É possível nomear um tipo (abstrato de dados) baseado em uma estrutura
  - Para isso utiliza-se **typedef** na declaração

```
typedef struct{
   char descricao[30];
   int quantidade;
   float preco_unitario;
   float desconto;
   float preco_total;
}Mercadoria;

int main(){
   Mercadoria feijao = {"redondo", 1, 20.0, 0, 20.0};
   Mercadoria feijao_carioca;

   feijao_carioca = feijao;
```

Mercadoria é o nome do tipo

Agora não existe mais a necessidade de:

struct Mercadoria feijao;

structs

- É possível nomear um tipo (abstrato de dados) baseado em uma estrutura
  - Para isso utiliza-se **typedef** na declaração

```
typedef struct{
   int dia, mes, ano;
}Data;


int main()
{
   Data atual;

   return 0;
}
```

structs

```
typedef struct{
   int dia, mes, ano;
}Data;


int main()
{
   Data atual;

   return 0;
}
```

São equivalentes

```
struct Data{
   int dia, mes, ano;
};


int main()
{
   struct Data atual;

   return 0;
}
```

Exercício - Struct

1. Considerando a estrutura

```
typedef struct {
    float x;
    float y;
    float z;
}Vetor;
```

Para representar um vetor em 3 dimensões, implemente um programa que calcule a soma de dois vetores.

structs

- É possível passar para funções:
  - variáveis membros da *struct*;
  - A variável *struct* como um todo.

- **Passagem por valor:** Uma cópia da variável é passada para a função;
- **Passagem por referência:** O endereço da variável é passado para a função.

- Quais as características de cada abordagem? Qual é melhor?

structs

- **Passagem por valor**

```
typedef struct{
   int x, y, z;
}Ponto;

void imprime(int v){
   printf("Valor: %d", v);
}

int main()
{
    Ponto p = {1, 2, 3};

    imprime(p.x);
    ...
}
```

Tem que ser do mesmo tipo!

structs

- **Passagem por referência**

```
typedef struct{
   int x, y, z;
}Ponto;

void incrementa_imprime(int *v){
   *v = *v + 1;
   printf("Valor: %d", *v);
}

int main()
{
    Ponto p;

    imprime(&p.y);
    ...
}
```

O operador **&** precede o nome da estrutura, não o nome da variável membro!

structs

- **Passando struct toda como valor**

```
1  typedef struct{
2     char nome[30];
3     char matricula[10];
4     float notas[4];
5  }Aluno;
6
7  void imprimeAluno(Aluno a){
8     int i;
9
10    puts(a.nome);
11    puts(a.matricula);
12
13    for(i=0;i<4;i++)
14       printf(" %f ", a.notas[i]);
15 }
```

```
16 int main(){
17       Aluno a;
18       int i;
19
20       scanf("%s", a.nome);
21       scanf("%s", a.matricula);
22
23       for(i=0;i<4;i++)
24          scanf("%f", &a.notas[i]);
25
26       imprimeAluno(a);
27
28       return 0;
29 }
```

structs

- **Passando struct toda como referência**

```
typedef struct{
   int x, y, z;
}Ponto;

void altera(Ponto *v){
   (*v).x = (*v).x + 1;
   (*v).y = (*v).y + 1;
   (*v).z = (*v).z + 1;
}

int main()
{
    Ponto p = {1, 2, 3};

    altera(&p);
    ...
}
```

Equivalentes

```
typedef struct{
   int x, y, z;
}Ponto;

void altera(Ponto *v){
   v->x = v->x + 1;
   v->y = v->y + 1;
   v->z = v->z + 1;
}

int main()
{
    Ponto p = {1, 2, 3};

    altera(&p);
    ...
}
```

structs

- **Retornando structs**

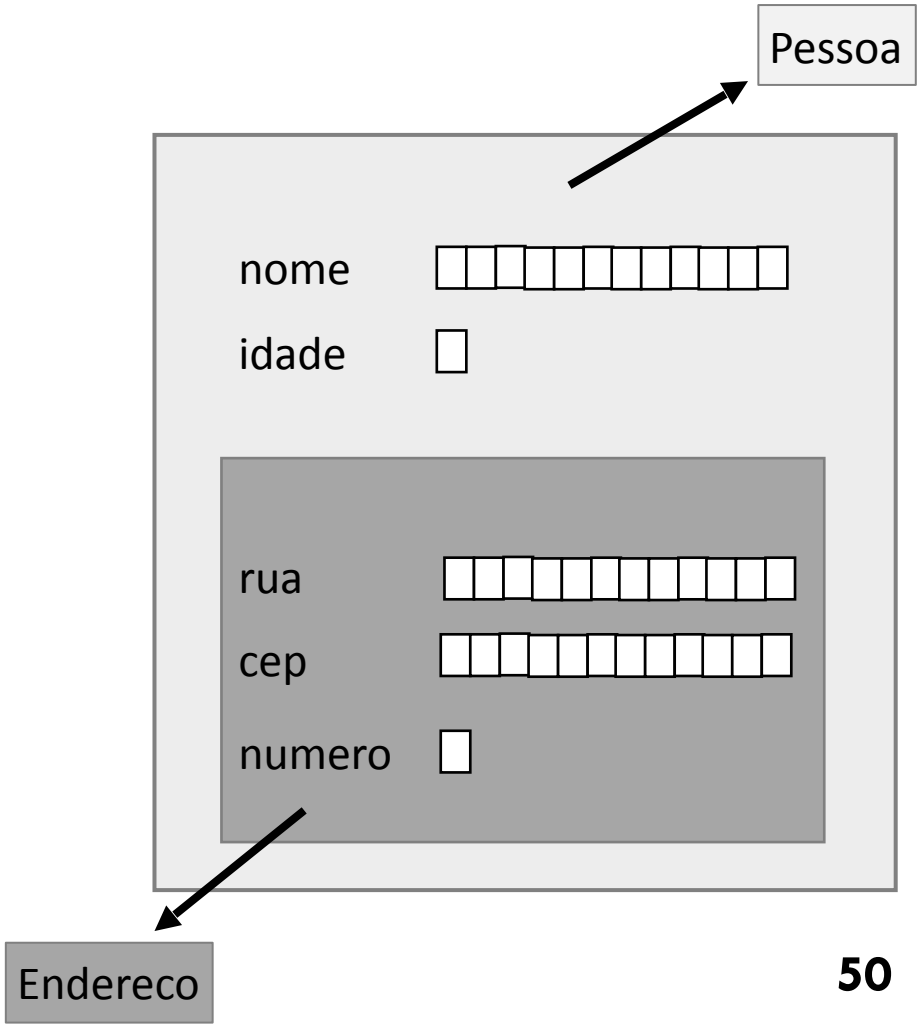
```
 1 typedef struct{
 2     char modelo[20], placa[8];
 3     int ano;
 4 }Carro;
 5
 6 Carro iniciaCarro(char *m, char *p, int a){
 7     Carro c;
 8
 9     strcpy(c.modelo, m);
10     strcpy(c.placa, p);
11     c.ano = a;
12
13     return c;
14 }
```

```
int main(){
    Carro novo_carro;

    novo_carro = iniciaCarro("Ferrari", "abc1234", 2018);
    ...
```

structs

- **estruturas aninhadas**

```
typedef struct{
    char rua[50], cep[9];
    int numero;
}Endereco;

typedef struct{
    char nome[50];
    int idade;
    Endereco end;
}Pessoa;
```



structs

- **estruturas aninhadas**

```
typedef struct{
    char rua[50], cep[9];
    int numero;
}Endereco;

typedef struct{
    char nome[50];
    int idade;
    Endereco end;
}Pessoa;
```

```
int main(){
    Pessoa p;

    strcpy(p.nome, "Cardoso");
    p.idade = 45;
    strcpy(p.end.cep, "123456789");
    p.end.numero = 4;
    strcpy(p.end.rua, "Rua Alan Turing");

    puts(p.nome);
    printf("%d", p.idade);
    puts(p.end.cep);
    puts(p.end.rua);
    printf("%d", p.end.numero);

    return 0;
}
```

Exercícios - Structs e Ponteiros

1. Defina uma estrutura que irá representar bandas de música. Essa estrutura deve ter o **nome** da banda, que **tipo** de música ela toca, o **número de integrantes** e em que **posição do ranking** essa banda está dentre as suas 5 bandas favoritas;
2. Crie uma função para preencher as 5 estruturas de bandas criadas no exemplo passado. Após criar e preencher, exiba todas as informações das bandas/estruturas. Não se esqueça de usar o operador → para preencher os membros das structs;
3. Crie uma função que peça ao usuário um número de 1 até 5. Em seguida, seu programa deve exibir informações da banda cuja posição no seu ranking é a que foi solicitada pelo usuário;

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Arquivos

- Basicamente, a linguagem C trabalha com dois tipos de arquivos:

Arquivo texto

- Armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de textos;
- Os dados são gravados como caracteres de 8 bits. Ex.: Um número inteiro de 32 bits com 8 dígitos ocupará 64 bits no arquivo.

Arquivo binário

- Armazena uma sequência de bits que está sujeita as convenções do programa que o gerou. Ex.: arquivos compactados;
- Os dados são gravados em binário, ou seja, do mesmo modo que estão na memória. Ex.: um número inteiro de 32 bits com 8 dígitos ocupará 32 bits no arquivo.

A manipulação de arquivos se dá por meio de fluxos (***streams***).

Arquivos

- A biblioteca `stdio.h` dá suporte à utilização de arquivos em C.
  - Renomear e remover;
  - Garantir acesso ao arquivo;
  - Ler e escrever;
  - Alterar o posicionamento dentro do arquivo;
  - Manusear erros;
  - Para mais informações: ***http://www.cplusplus.com/reference/cstdio/***

- A linguagem C não possui funções que leiam automaticamente toda a informação de um arquivo:
  - Suas funções limitam-se em **abrir/fechar** e **ler/escrever** caracteres ou bytes;
  - O programador deve instruir o programa na leitura do arquivo de uma maneira específica;

Arquivos

- **Todas** as funções de manipulação de arquivos trabalham com o conceito de "**ponteiro de arquivo**";
- Um ponteiro de arquivo é um ponteiro para informações que definem várias coisas sobre o arquivo, incluindo seu nome, *status* e a posição atual do arquivo;
- Um ponteiro de arquivo é uma variável ponteiro do tipo **`FILE`** (definido na biblioteca `stdio.h`);
- Pode-se declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *arq
```

- `arq` é o ponteiro para arquivos que permite manipular um arquivo.

Arquivos

- Para a abertura de um arquivo, usa-se a função fopen:

```
File *arq;

fopen(nome_arquivo, modo_de_abertura);
```

- O parâmetro *nome_arquivo* determina qual o arquivo ser aberto, incluindo a extensão
  - O nome deve ser válido no sistema operacional que estiver sendo utilizado;
  - **Caminho absoluto:** descrição de um caminho desde o diretório raiz
    - C:\Programação\aula18\dados.txt
  - **Caminho relativo:** descrição de um caminho dede o diretório corrente, ou seja, onde o programa está salvo
    - dados.txt
    - ..\dados.txt

Arquivos

```
FILE *arq;

arq = fopen(nome_arquivo, modo_de_abertura);
```

A função fopen retorna um ponteiro do tipo FILE

- O modo de abertura determina que tipo de **USO** será feito do arquivo
  - Abrir para leitura;
  - Abrir para escrita;
  - Abrir para leitura e escrita.

Arquivos

- Modos clássicos

Modo	Arquivo	Função
"r"	Texto	Leitura. Arquivo deve existir.
"w"	Texto	Escrita. Criar arquivo se não houver. Apaga o anterior se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no final do arquivo (*append*).
"rb"	Binário	Leitura. Arquivo deve existir.
"wb"	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab"	Binário	Escrita. Os dados serão adicionados no fim do arquivo (*append*)
"r+"	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+"	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir
"a+"	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo (*append*).
"r+b"	Binário	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+b"	Binário	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+b"	Binário	Leitura/Escrita. Os dados serão adicionados no fim do arquivo (*append*).

Arquivos

- Um arquivo do tipo texto pode ser aberto para escrita utilizando o seguinte conjunto de comandos:
  - A condição arq == NULL testa se o arquivo foi aberto com sucesso;
  - No caso de erro a função fopen retorna um ponteiro nulo (NULL).

```
int main(){
     FILE *arq;

     arq = fopen("teste.txt", "w");
     if(arq == NULL)
           printf("Ocorreu um erro na abertura do arquivo");
       ...
```

Arquivos

- Um arquivo pode ser fechado pela função `fclose();`
  - Escreve no arquivo qualquer dado que ainda permanece no *buffer*;
    - Geralmente as informações só são gravadas no disco quando o *buffer* está cheio.
  - O ponteiro do arquivo é passado como parâmetro para `fclose();`
  - **Esquecer de fechar** o arquivo pode gerar **inúmeros problemas**;

```
int main(){
     FILE *arq;

     arq = fopen("teste.txt", "w");
     if(arq == NULL)
           printf("Ocorreu um erro na abertura do arquivo");
           system("pause");
           exit(1);
     }
     ...
     fclose(arq);

     return 0;
}
```

Arquivos

- A maneira mais fácil de trabalhar com um arquivo é a leitura/escrita de **um único caractere por vez;**
- A função fputc (*put character*) pode ser utilizado para esse princípio;

```
1  FILE *arq;
2  char str[] = "Texto a ser gravado no arquivo";
3  int i;
4  arq = fopen("Teste.txt", "w");
5  if(arq == NULL){
6      printf("Erro ao abrir o arquivo");
7      system ("pause");
8      exit(1);
9  }
10 for(i=0;i<strlen(str);i++)
11     fputc(str[i], arq);
12
13 fclose(arq);
```

```
fputc(caractere, ponteiro);
```

Equivale à:

```
putc(caractere, ponteiro);
```

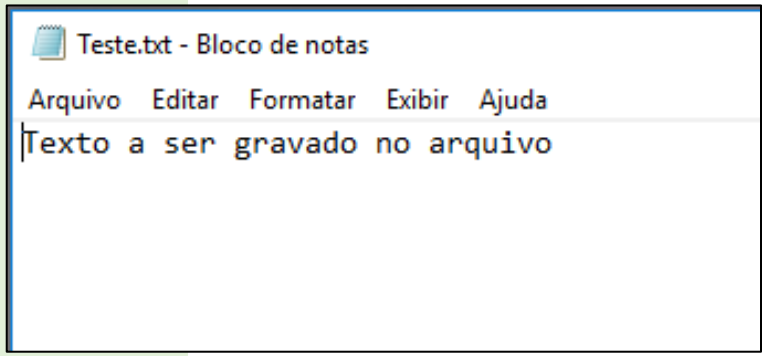
Usada também para impressão:

```
fputc('a', stdout);
```

Arquivos

- Agora usando **while**...

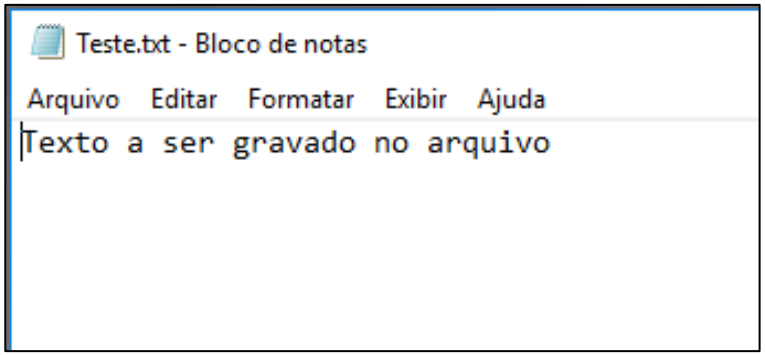
```
1  FILE *arq;
2  char str[] = “Texto a ser gravado no arquivo”;
3  int i;
4  arq = fopen(“Teste.txt”, “w”);
5  if(arq == NULL){
6      printf(“Erro ao abrir o arquivo”);
7      system (“pause”);
8      exit(1);
9  }
10 i = 0;
11 while(str[i] != ‘\0’){
12     fputc(str[i], arq);
13     i++;
14 }
15
16 fclose(arq);
```



Arquivos

- Também podemos ler caracteres um a um do arquivo;
- A função usada para isso é a fgetc (*get character*);

```
1  FILE *arq;
2  char c;
3  int i;
4  arq = fopen("Teste.txt", "r");
5  if(arq == NULL){
6      printf("Erro ao abrir o arquivo");
7      system ("pause");
8      exit(1);
9  }
10
11 c = fgetc(arq);
12 while(c != EOF){
13     printf("%c", c);
14     c = fgetc(arq);
15 }
16
17 fclose(arq);
```



Saída:

Texto a ser gravado no arquivo

Arquivos

- Podemos testar se chegamos ao final do arquivo por meio da função feof();

feof na condição do loop: **mal uso!**

```
 1 FILE *arq;
 2 char c;
 3
 4 arq = fopen("Teste.txt", "r");
 5 if(arq == NULL){
 6     ...
 7 }
 8
 9 while(!feof(arq)){
10    c = fgetc(arq);
11    printf("%c", c);
12 }
13 fclose(arq);
```

feof verifica o indicador de erro

feof na condição do loop: **uso melhor!**

```
 1 FILE *arq;
 2 char c;
 3
 4 arq = fopen("Teste.txt", "r");
 5 if(arq == NULL){
 6     ...
 7 }
 8
 9 while(1){
10    c = fgetc(arq);
11    if(feof(arq))
12       break;
13    printf("%c", c);
14 }
15 fclose(arq);
```

Exercícios - Arquivos

1. Faça um programa que receba do usuário um arquivo texto e mostre na tela quantas linhas esse arquivo possui.

2. Faça um programa que receba do usuário um arquivo texto. Crie outro arquivo texto contendo o texto do arquivo de entrada, mas com as vogais substituídas por '*'.

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Revisão

- Ponteiros → *int *p = &n;*
- Alocação Dinâmica → *malloc, free, sizeof*
- Recursividade → chamadas sucessivas da mesma função
- Tipos Abstratos → *structs, tyepdef*
- Arquivos → *fopen, fclose, fgetc, fputc, feof*

Próximas Aulas

- Pilhas
- Filas/ Deques

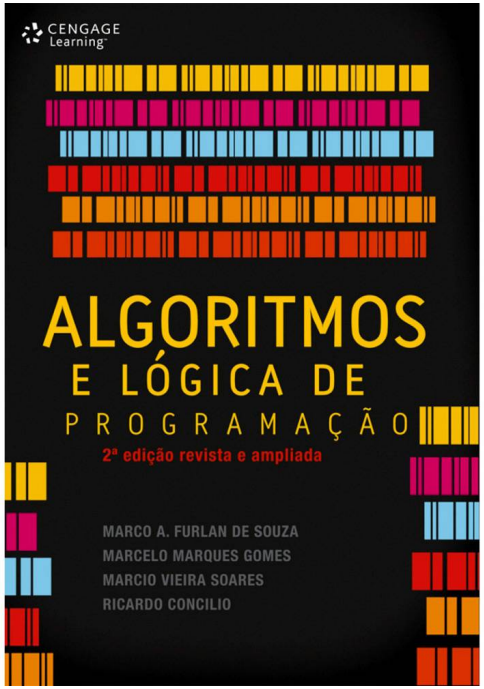
- Implementação de Listas Lineares
  - single-linked
  - double-linked

Roteiro

1. Ponteiros
2. Alocação Dinâmica de Memória
3. Recursividade
4. Tipos Abstratos de Dados
5. Arquivos
6. Síntese / Revisão
7. Referências

Referências

[Schildt, 1997]



[de Souza et al, 2011]

Referências

1. Notas de aula profa. Silvana M. A. de Lara. Universidade de São Paulo – São Carlos. ICMC.

2. Notas de aula prof. Luiz Fernando Carvalho. Universidade Tecnológica Federal do Paraná. UTFPR, Apucarana.

Perguntas?

Prof. Rafael G. **Mantovani**

rafaelmantovani@utfpr.edu.br