Processando a Informação: um livro prático de programação independente de linguagem

Rogério Perino de Oliveira Neves

Francisco de Assis Zampirolli

EDUFABC
editora.ufabc.edu.br

Notas de Aulas inspiradas no livro

Utilizando a(s) Linguagem(ns) de Programação:

C

Exemplos adaptados para Correção Automática no Moodle+VPL

Francisco de Assis Zampirolli

20 de novembro de 2022

Sumário

1 Processando a Informação: Cap. 7: Tipos Definidos Pelo Programador e Arquivos

Este caderno (Notebook em CONSTRUÇÃO) é parte complementar *online* do livro **Processando a Informação: um livro prático de programação independente de linguagem**, que deve ser consultado no caso de dúvidas sobre os temas apresentados.

> Este conteúdo pode ser copiado e alterado livremente e foi inspirado nesse livro.
>
> O conteúdo deste capítulo foi inspirado em: * Cap. 7: Conceitos de programação orientada a objetos do livro anterior * Notas de aula de professores da UFABC, em especial, dos professores Luiz Rozante e Wagner Botelho.

1.1 Sumário

- Revisão do capítulo anterior
- Introdução
- Paradigma Estruturado
- Paradigma Orientado a Objetos
- Tipos de dados
- Arquivos
- Revisão deste capítulo
- Exercícios

1.2 Revisão do capítulo anterior (Matriz)

- Introdução
- Instanciando matrizes
- Acessando elementos de uma matriz
- Formas de percorrer uma matriz
- Aplicações usando matrizes
- Exercícios

1.3 Introdução

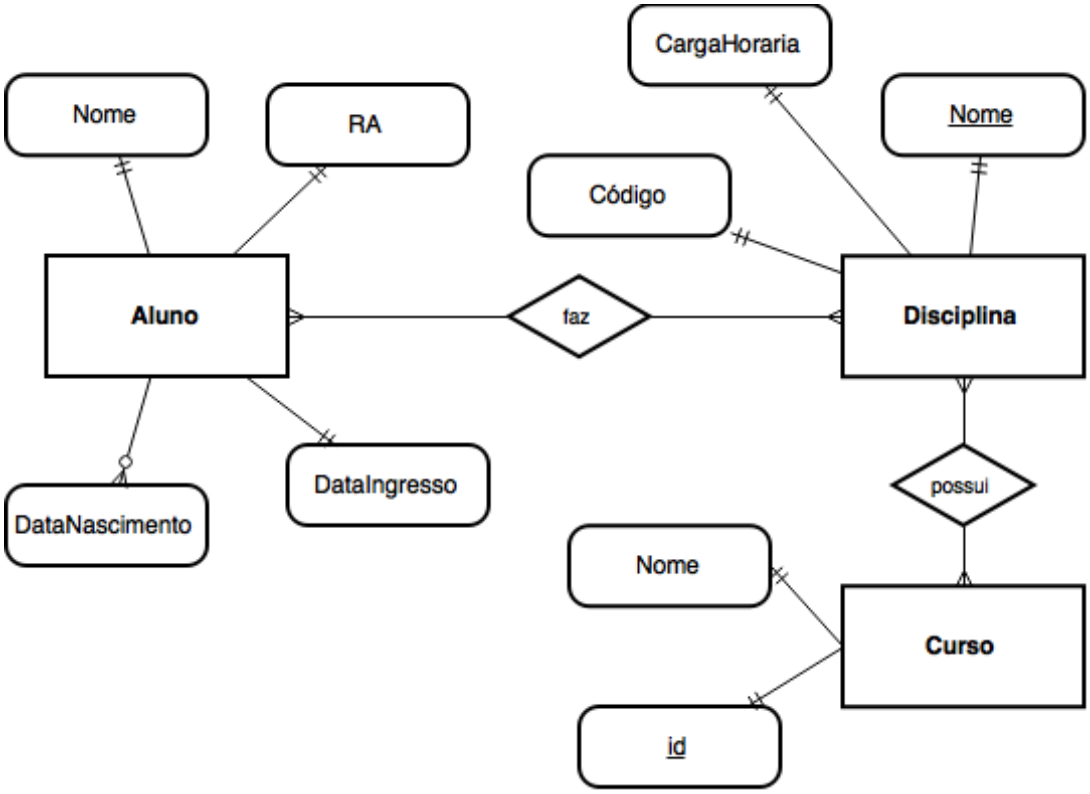
- Este capítulo introduz alguns conceitos usados em:
  - Programação Estruturada,
  - Programação Orientada a Objetos (POO) e
  - Engenharia de Software (ES).
- Esses conceitos serão úteis em estudos futuros.
- Não é o propósito deste capítulo cobrir completamente estes temas, mas sim proporcionar um ponto de partida para o estudo avançados em programação.
- Além de introduzir processos para desenvolver códigos envolvendo equipes e utilizando princípios da Engenharia de Software.

1.4 Paradigma Estruturado

- Uns dos primeiros estilos (ou paradigma) para estruturar um Software é chamado de **Programação Estruturada**.
- Neste paradigma estruturado, o programador define **Tabelas** (ou **Registros** ou também chamadas **Entidades**), onde se podem armazenar variáveis de vários tipos de dados,
  - diferentemente dos vetores e matrizes vistos nos capítulos anterios, onde todos os dados devem ser de um mesmo tipo de dado (exceto as listas em Python).
- Por exemplo, é possível definir uma tabela para armazenar as informações de um aluno num contexto de um sistema acadêmico, chamada **Aluno**.
- Esta tabela **Aluno**, na verdade, pode ser considera um novo tipo de dados e é possível, por exemplo, "instanciar" uma variável (ou registro), como *Julia* do tipo **Aluno**.
- Como atributos da tabela **Aluno**, é possível ter *nome*, *matrícula*, *data de nascimento*, *ano de ingresso*, etc.
- Observe que na tabela **Aluno** é importante também ter informação de curso e de disciplinas já cursadas.
- Estas informações podem ser "instâncias" de outras tabelas, como **Curso** e **Disciplina**, com seus respectivos atributos apropriados.
- Em Banco de Dados estas "instâncias" são possíveis através de um atributo representando a chave (estrangeira) de outra tabela.
- Além disso, cada tabela possui um atributo (chamado chave primária) que identifica o registro, por exemplo, Julia, que pode estar armazenada num registro de número 33.
- Resumindo, as **estrutura** definidas pelo programador podem armazenar em tempo de execução as **tabelas** definidas em Bancos de Dados (com informações armazenadas em disco).

1.5 Diagrama Entidade Relacionamento

- Para poder visualizar melhor esta relação entre as tabelas **Aluno**, **Curso** e **Disciplina**, existe um modelo apropriado, chamado **Diagrama Entidade Relacionamento (DER)**, como apresentado na figura abaixo.
- Observe o relacionamento "**Aluno** faz **Disciplina**(s)".
- Se for considerar um contexto onde um aluno está cursando uma disciplina, existe a necessidade de incluir uma nova tabela, chamada, por exemplo, **Turma**.
- Assim, "**Turma** possui **Aluno**(s)" e "**Disciplina** possui Turma(s)".

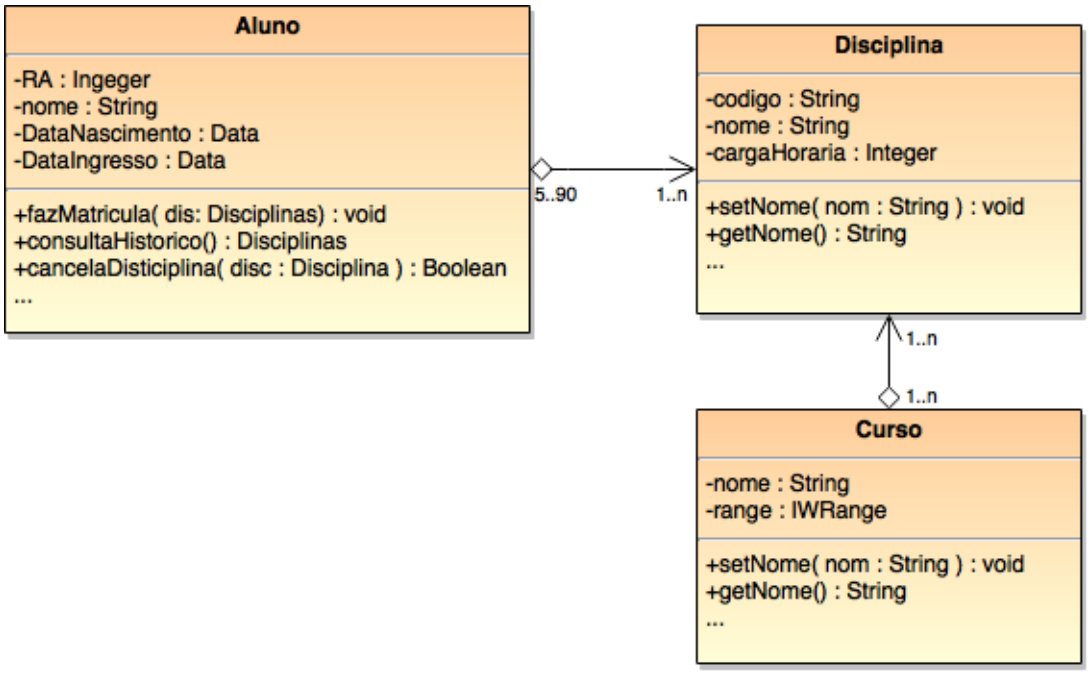


- Desse modo, para um sistema acadêmico completo, é necessário criar várias tabelas e seus relacionamentos.
- Todos esses dados das tabelas devem ser lidos (digitados) uma vez e armazenados em disco.
- Senão, toda vez que esse sistema acadêmico for executado, todos os dados deverão ser lidos novamente.
- Existem várias formas de guardar os dados de variáveis em disco:
  - arquivo texto,
  - arquivo binário ou
  - tabelas em um Sistema de Gerenciamento de Banco de Dados (SGBD), como Oracle, JDBC, SQL, PostgreSQL, etc.

- Nas disciplinas de Banco de Dados, o aluno aprende como criar essas tabelas em um SGBD e também como usá-las dentro de uma linguagem de programação.
- Essa forma de organizar vários tipos de dados em tabelas chama-se **encapsulamento de dados**.
- O grande desafio dos programadores e analistas de sistemas é definir corretamente essas tabelas e os seus relacionamentos. Os sistemas bem modelados são mais fáceis de realizar manutenções.
- Além disso, os programadores devem criar estruturas, funções e procedimentos de forma bem organizada, por exemplo, usando vários arquivos organizados em pastas.
- O paradigma estruturado não facilita completamente esta organização. Isso já não ocorre no paradigma orientado a objetos, resumido a seguir.

1.6 Paradigma Orientado a Objetos

- Na programação estruturada não existe uma forma eficiente de organizar (encapsular) as funções específicas de uma tabela, como ocorre no encapsulamento dos dados, visto na programação estruturada.
- Assim, surgiu a necessidade de criar um novo paradigma de programação, que é a **Programação Orientada a Objetos (POO)**, onde, além de encapsular os dados, é possível encapsular as funções ou métodos que os processam.
- Na POO, tabela passou a se chamar **Classe**, e um registro de uma tabela passou a ser chamado de **instância** de uma classe, também chamado de **objeto**.
- Como no exemplo anterior, para o sistema acadêmico, é possível criar a classe **Aluno** (que é um novo tipo de dados) e com ela instanciar uma variável *Julia* da classe **Aluno**.



O restante deste capítulo é complemetar ao livro **Processando a Informação: um livro prático de programação independente de linguagem**, apresentando conceitos e exemplos de Programação Estruturada.

1.7 Tipos de dados

- Os tipos de dados em C são:
  - `struct`

```
struct MEU_TIPO{
 tipo1 nome1;
 tipo2 nome2;
 ...
};
struct MEU_TIPO Entidade1, Entidade2;
```

  - Ou melhor, com `typedef` (utilizado para renomear um tipo de dado da própria linguagem ou definido pelo programador):

```
typedef struct {
tipo1 nome1;
tipo2 nome2;
...
} MEU_TIPO;
MEU_TIPO Entidade1, Entidade2;
```

  - `union`
  - `enum`

1.7.1 Exemplo 01 - Criar um registro de Aluno

Exemplo para criar a `struct` **TAluno** contendo 4 atributos.

[ ]:
```
%%writefile cap7ex01.c
#include <stdio.h>
#include <string.h>

struct TAluno {
  char nome[50];
  int idade;
  char rua[50];
  int numero;
};

int main() {
  struct TAluno ana; // instancia uma variável c do tipo TAluno
  // ENTRADA DE DADOS
  strcpy(ana.nome, "Ana Silva");
  ana.idade = 18;
  strcpy(ana.rua, "Avenida Paulista");
```

```
  ana.numero = 1000;

  // SAÍDA DE DADOS
  printf("nome: %s\nidade: %d\n", ana.nome, ana.idade);
  printf("rua: %s\nnúmero: %d\n", ana.rua, ana.numero);
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex01.c -o output
./output
```

1.7.2 Exemplo 02 - Criar um registro de Aluno, com `scanf`

Exemplo para criar a `struct` **TAluno** contendo 4 atributos lidos do teclado com `scanf`.

[ ]:
```
%%writefile cap7ex02.c
#include <stdio.h>
#include <string.h>

struct TAluno {
  char nome[50];
  int idade;
  char rua[50];
  int numero;
};

int main() {
  struct TAluno Alunos[2]; // instancia uma vetor do tipo TAluno
  // ENTRADA DE DADOS
  for (int i = 0; i < 2; i++) {
    printf("Entre com os dados: nome, idade, rua, número:\n");
    fflush(stdin);
    fgets(Alunos[i].nome, 50, stdin);
    scanf("%d", &Alunos[i].idade);
    fflush(stdin);
    fgets(Alunos[i].rua, 50, stdin);
    scanf("%d", &Alunos[i].numero);
  }
  // SAÍDA DE DADOS
  for (int i = 0; i < 2; i++) {
    printf("nome: %s\nidade: %d\n", Alunos[i].nome, Alunos[i].idade);
    printf("rua: %s\nnúmero: %d\n", Alunos[i].rua, Alunos[i].numero);
  }
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex02.c -o output
./output
```

1.7.3 Exemplo 03 - Criar um registro de Aluno, com `typedef`

Exemplo para criar a `struct` **TAluno** contendo 4 atributos lidos do teclado, renomeando com `typedef` para **Aluno**.

[ ]:
```
%%writefile cap7ex03.c
#include <stdio.h>
#include <string.h>

struct TAluno {
  char nome[50];
  int idade;
  char rua[50];
  int numero;
};

typedef struct TAluno Aluno;

int main() {
  // ENTRADA DE DADOS
  Aluno Ana = { "Ana Silva" , 18, "Avenida Paulista" , 1000 };

  // SAÍDA DE DADOS
  printf("%s %d %s %d", Ana.nome, Ana.idade, Ana.rua, Ana.numero);
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex03.c -o output
./output
```

1.7.4 Exemplo 04 - Criar um registro de Aluno, com `typedef`, opção 2

Exemplo para criar a `struct` **Aluno** de forma simplificada, contendo 4 atributos.

[ ]:
```
%%writefile cap7ex04.c
#include <stdio.h>
#include <string.h>

typedef struct TAluno {
  char nome[50];
  int idade;
  char rua[50];
  int numero;
```

```
} Aluno;

int main() {
  // ENTRADA DE DADOS
  Aluno ana = { "Ana Silva" , 18, "Avenida Paulista" , 1000 };

  // SAÍDA DE DADOS
  printf("%s %d %s %d", ana.nome, ana.idade, ana.rua, ana.numero);
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex04.c -o output
./output
```

1.7.5 Exemplo 05 - Criar dois registros

Exemplo para criar a `struct` **Cliente** contendo 3 atributos, sendo um deles uma outra estrutura **Endereco**.

[ ]:
```
%%writefile cap7ex05.c
#include <stdio.h>
#include <string.h>

typedef struct {
  char rua[50];
  int numero;
} Endereco;

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

void leiaCliente(Cliente* cliente) {
  fflush(stdin);
  printf("nome: ");
  fgets((*cliente).nome, 50, stdin);
  printf("idade: ");
  scanf("%d", &(*cliente).idade);
  fflush(stdin);
  printf("rua: ");
  fgets((*cliente).end.rua, 50, stdin);
  printf("numero: ");
  scanf("%d", &(*cliente).end.numero);
}
```

```
void escrevaCliente(Cliente cliente) {
  printf("nome: %s\nidade: %d\n", cliente.nome, cliente.idade);
  printf("rua: %s\nnumero: %d\n", cliente.end.rua, cliente.end.numero);
}

int main() {
  Cliente clientes[2];
  // ENTRADA DE DADOS
  for (int i = 0; i < 2; i++) {
    printf("Cadastrar dados do cliente %d\n", i + 1);
    leiaCliente(&clientes[i]);
  }

  // SAÍDA DE DADOS
  for (int i = 0; i < 2; i++) {
    printf("Imprimir dados do cliente %d\n", i + 1);
    escrevaCliente(clientes[i]);
  }
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex05.c -o output
./output
```

1.7.6 Exemplo 06 - Criar dois registros usando biblioteca

Exemplo para criar a `struct` **Cliente** contendo 3 atributos, sendo um deles uma outra estrutura **Endereco**, organizados em uma biblioteca.

[ ]:
```
%%writefile cap7ex06.c
#include "myBiblioteca.h"

int main() {
  Cliente clientes[2];
  // ENTRADA DE DADOS
  for (int i = 0; i < 2; i++) {
    printf("Cadastrar dados do cliente %d\n", i + 1);
    leiaCliente(&clientes[i]);
  }

  // SAÍDA DE DADOS
  for (int i = 0; i < 2; i++) {
    printf("Imprimir dados do cliente %d\n", i + 1);
    escrevaCliente(clientes[i]);
  }
  return 0;
}
```

[ ]:
```
%%writefile myBiblioteca.h
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
  char rua[50];
  int numero;
} Endereco;

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

void leiaCliente(Cliente* cliente);
void escrevaCliente(Cliente cliente);
```

[ ]:
```
%%writefile myBiblioteca.c
#include "myBiblioteca.h"

void escrevaCliente(Cliente cliente) {
  printf("nome: %s\nidade: %d\n", cliente.nome, cliente.idade);
  printf("rua: %s\nnúmero: %d\n", cliente.end.rua, cliente.end.numero);
}

// leia um cliente a partir do teclado
// nome, idade, rua, numero
void leiaCliente(Cliente* cliente) {
  fflush(stdin);
  printf("nome: ");
  fgets((*cliente).nome, 50, stdin);
  printf("idade: ");
  scanf("%d", &(*cliente).idade);
  fflush(stdin);
  printf("rua: ");
  fgets((*cliente).end.rua, 50, stdin);
  printf("numero: ");
  scanf("%d", &(*cliente).end.numero);
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex06.c myBiblioteca.c -o output
./output
```

1.8 Arquivos

Os arquivos armazenados em disco são considerados como uma sequência de caracteres e podem ser acessados em memória através de um ponteiro para o primeiro caracter do arquivo (texto ou binário), como segue:

```
FILE *fp;
fp = fopen("nomeArquivo.txt", "modo"); // modo é definido a seguir
```

Seguem as principais funções para manipular arquivos da biblioteca `stdio.h`:

Função	Descrição
`fopen()`	Abre arquivo
`fclose()`	Fecha arquivo
`putc()`	Escreve um caracter no arquivo
`fputc()`	Igual `putc()`
`getc()`	Lê um caracter do arquivo
`fgetc()`	Igual `getc()`
`fseek()`	Posiciona o arquivo em um determinado byte
`fprintf()`	Semelhante ao `printf()`, mas para arquivo
`fscanf()`	Semelhante ao `scanf()`, mas para arquivo
`feof()`	Verifica final de arquivo
`ferror()`	Verifica se ocorreu erro
`rewind()`	Posiciona o ponteiro para o início do arquivo
`remove()`	Apaga arquivo
`fflush()`	Descarrega o *buffer* do arquivo
`fgets()`	Obtém uma string do arquivo
`fread()`	Lê um bloco de dados do arquivo
`fwrite()`	Escreve um bloco de dados no arquivo
`ftell()`	Retorna a posição do ponteiro

Modo	**Arquivo**	**Função**
r	Texto	Leitura
w	Texto	Escrita em arquivo novo
a	Texto	Escrita em arquivo existente
r+	Texto	Leitura/Escrita
w+	Texto	Leitura/Escrita
a+	Texto	Leitura/Escrita
rb	Binário	Leitura
wb	Binário	Escrita
ab	Binário	Escrita
r+b	Binário	Leitura/Escrita
w+b	Binário	Leitura/Escrita
a+b	Binário	Leitura/Escrita

1.8.1 Exemplo 07 - Criar Arquivo

Exemplo para criar um arquivo texto.

[ ]:
```
%%writefile cap7ex07.c
#include <stdio.h>

int main() {
  char* filename = "teste.txt";
  FILE* file;

  file = fopen(filename, "w"); // Cria arquivo para escrita
  if (file == NULL) {
    printf("Erro ao abrir o arquivo: %s\n", filename);
    return -1;
  }

  // SAÍDA DE DADOS
  for (int i = 0; i < 10; i++)
    fprintf(file, "Linha %02d\n", i + 1); // Escrever algo no arquivo

  fclose(file); // fecha arquivo
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex07.c -o output
./output
cat teste.txt
```

1.8.2 Exemplo 08 - Ler arquivo

Exemplo para ler um arquivo texto.

[ ]:
```
%%writefile cap7ex08.c
#include <stdio.h>

int main() {
  char* filename = "teste.txt", ch;
  FILE* file;

  // ENTRADA DE DADOS
  file = fopen(filename, "r"); // Cria arquivo para leitura
  if (file == NULL) {
    printf("Erro ao abrir o arquivo: %s\n", filename);
    return -1;
  }

  // SAÍDA DE DADOS
  while ((ch = fgetc(file)) != EOF) { // Lê arquivo
    printf("%c", ch); // caracter por caracter
  }
```

```
  fclose(file); // fecha arquivo
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex08.c -o output
./output
```

1.8.3 Exemplo 09 - Criar dois registros usando biblioteca, lendo arquivo

Exemplo para criar a `struct` **Cliente** contendo 3 atributos, sendo um deles uma outra estrutura **Endereco**, organizados em uma biblioteca.

Destacando que os dados serão lidos do arquivo CSV abaixo:

[ ]:
```
%%writefile dados.csv
Maria Souza, 19, Avenida Paulista, 1000
Pedro Silva, 18, Avenida Rebouças, 2500
```

[ ]:
```
%%writefile cap7ex09.c
#include "myBiblioteca.h"

int main() {
  Cliente clientes[20];
  char* filename = "dados.csv";

  // ENTRADA DE DADOS
  FILE* file; // Cria arquivo para leitura
  if ((file = fopen(filename, "r")) == NULL) {
    printf("Erro ao abrir o arquivo: %s\n", filename);
    return -1;
  }

  int contador = 0;
  char linha[512]; // espaço para cada linha lida
  while (fgets(linha, sizeof(linha), file))  // para cada linha
    criaCliente(&clientes[contador++], linha);

  // SAÍDA DE DADOS
  for (int i = 0; i < contador; i++) {
    printf("\nImprimir dados do cliente %d\n", i + 1);
    escrevaCliente(clientes[i]);
  }

  fclose(file); // fecha arquivo
  return 0;
}
```

[ ]:
```
%%writefile myBiblioteca.h
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
  char rua[50];
  int numero;
} Endereco;

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

void leiaCliente(Cliente* cliente);
void escrevaCliente(Cliente cliente);
void criaCliente(Cliente* cliente, char linha[512]);
```

[ ]:
```
%%writefile myBiblioteca.c
#include "myBiblioteca.h"
// cria um cliente a partir de uma linha de texto. Exemplo:
// "Maria Souza, 20, Avenida Paulista, 1200"
void criaCliente(Cliente* cliente, char linha[512]) {
  // Ler nome
  char* token = strtok(linha, ","); // ler até a primeira ","
  strcpy((*cliente).nome, token); // nome

  // Ler idade
  token = strtok(NULL, ","); // continar lendo até a próxima ","
  (*cliente).idade = atoi(token); // converter str to int

  // Ler rua
  token = strtok(NULL, ",");
  strcpy((*cliente).end.rua, token + 1); // +1 retira 1o espaço em␣
 ↪branco

  // Ler numero
  token = strtok(NULL, ",");
  (*cliente).end.numero = atoi(token); // converter str to int
}
void escrevaCliente(Cliente cliente) {
  printf("nome: %s\nidade: %d\n", cliente.nome, cliente.idade);
  printf("rua: %s\nnúmero: %d\n", cliente.end.rua, cliente.end.numero);
}
```

```c
// leia um cliente a partir do teclado
// nome, idade, rua, numero
void leiaCliente(Cliente* cliente) {
  fflush(stdin);
  printf("nome: ");
  fgets((*cliente).nome, 50, stdin);
  printf("idade: ");
  scanf("%d", &(*cliente).idade);
  fflush(stdin);
  printf("rua: ");
  fgets((*cliente).end.rua, 50, stdin);
  printf("numero: ");
  scanf("%d", &(*cliente).end.numero);
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap7ex09.c myBiblioteca.c -o output
./output
```

1.9 Exercícios

Ver notebook Colab nos arquivos `cap7.partX.lab.*.ipynb` (`X` ∈ `[2,3,4,5]` e `*` é a extensão da linguagem), utilizando alguma linguagem de programação de sua preferência, onganizadas em subpastas contidas em `"gen"`, na pasta do Google Drive colabs.

1.10 Revisão deste capítulo

- Introdução
- Paradigma Estruturado
- Paradigma Orientado a Objetos
- Tipos de dados
- Arquivos
- Exercícios
- Revisão deste capítulo de Vetores