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

17 de dezembro de 2022

Sumário

1 Processando a Informação: Cap. 2: Organização de código

Este caderno (Notebook) 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.

1.1 Sumário

- Revisão do capítulo anterior
- Programas sequenciais
- Comentários
- Desvios de fluxo
- Programas e Subprogramas
- Funções, métodos e modularização
- Reaproveitamento e manutenção de código
- Revisão deste capítulo
- Exercícios

1.2 Revisão do capítulo anterior (Fundamentos)

- No capítulo anterior foram apresentados os fundamentos para se iniciar a programar utilizando alguma linguagem de programação, além de alguns exemplos de códigos e principalmente de como executar estes códigos em ambientes de programação.
- Neste capítulo iremos iniciar a organizar esses códigos, utilizando o conceito de sistema de informação em partes:
  - **ENTRADA DE DADOS ⇒ PROCESSAMENTO DA INFORMAÇÃO ⇒ SAÍDA**

1.3 Programas sequenciais

- Como vimos no capítulo anterior, um programa consiste de uma **sequência lógica de comandos e operações** que, executadas em ordem, realizam uma determinada tarefa.
- Em geral, um programa processa dados de entrada de forma a obter o resultado desejado ou saída.
- A **entrada** e a **saída** de dados são, comumente, realizadas utilizando os dispositivos de entrada e saída, como teclado e monitor.
- Os dados requeridos para o **processamento** podem estar também contidos no código do programa (no exemplo a seguir, poderia ser `conta = 100`).
- No exemplo a seguir em **pseudocódigo** a entrada é lida do teclado (comando `leia()`) e a saída é impressa no monitor (comando `escreva`).

```
#1 ENTRADA de dados:
#1.1 Definição das variáveis do programa:
Real: conta, gorjeta;

#1.2 Entrada de dados
conta = leia("Entre o valor da conta: ");

#2 PROCESSAMENTO:
gorjeta = conta * 0.2;

#3 SAÍDA de dados (na tela):
escreva("Valor da gorjeta = " + gorjeta);
```

Assim, o procedimento para especificar um programa é definir: 1. Os dados de **entrada** necessários (insumos) 2. O **processamento** ou transformação dos dados de entrada em saída 3. A **saída** desejada do programa

1.4 Comentários

- Comentários são partes do código usados pelo desenvolvedor para deixar notas, explicações, exemplos, etc.
- Quando definido um comentário, é dada uma instrução direta ao **compilador/interpretador para ignorar a parte comentada**.
- Isto quer dizer que os comentários não serão considerados quando o código for executado.
- Logo, os comentários não são parte da execução.
- Cada linguagem tem sua própria maneira de introduzir comentários no código.

Tabela. Identificadores de comentário.

Linguagem	Uma linha	Várias linhas
Java/JS/C/C++	//...	/*...*/
Python	#...	"..." ou '"..."'
Matlab	%...	%{...}%

Tabela. Identificadores de comentário.		
Pascal	{...}	{...}
SQL	–...	/.../
R	#..	

1.5 Desvio de Fluxo

- Um programa consiste em uma sequência de comandos executados em ordem, em uma linha contínua de execução.
- No entanto, esta linha (em inglês: *thread*) pode conter desvios ou descontinuidades, processando códigos de bibliotecas ou subprogramas.

1.6 Programas e Subprogramas

- Em grande parte das linguagens de programação, **o código dos programas pode ser dividido em um ou mais arquivos ou partes**.
- Cada parte contem uma sequência de comandos com um objetivo, realizando uma tarefa dentro do programa.
- Dentro de um mesmo programa podem existir **subprogramas** (ou partes) com funções específicas ou subconjuntos de comandos que só serão executados em condições especiais.
- Todas as linguagens vêm acompanhadas de **bibliotecas**, estas contendo funções ou programas de uso comum.
- São exemplos as funções para cálculos matemáticos, para operações de entrada e saída, para comunicação e conversão de dados.

1.7 Bibliotecas

Cada linguagem de programação possui um conjunto de bibliotecas disponíveis para uso. As bibliotecas podem guardar variáveis ou funções.

1.8 Funções ou Métodos de Usuário

- O uso de funções facilita a **reutilização de código**, dado que uma função é um programa autocontido, com **entrada**, **processamento** e **saída**.
- Uma função pode ser copiada de um programa para outro ou incorporado em uma biblioteca escrita pelo usuário, utilizando o comando em Python `import myBiblioteca`. Ou também, para C/C++/Java, `#include "myBiblioteca.h"`.
- Uma função é definida por um nome, retorno (opcional), argumento(s) (opcional) e um conjunto de instruções.
- A seguir temos um exemplo de função de usuário escrita em pseudocódigo:

```
# MINHA(S) FUNÇÃO(ÕES)
função delta(recebe: real a, real b, real c) retorna real d {
```

```
    d = b2 - 4ac
    retorne d
}

principal {
  # ENTRADAS
  a = 5
  b = -2
  c = 4

  # PROCESSAMENTO
  real valor = delta(a, b, c) # AQUI ESTÁ A CHAMADA DA FUNÇÃO

  # SAÍDA
  escreva(“O delta de ax2 +bx + c é ” + valor)
}
```

```
TERMINOLOGIA: Os métodos podem ser chamados também de
módulos, funções, subprogramas ou procedimentos.

Existe uma convenção: quando um método tem argumento(s)
(parâmetros) e um retorno é chamado de função, caso contrário,
é chamado procedimento.

Porém, poucas linguagens fazem distinção na sintaxe entre
função e procedimento, como a Pascal, tornando confusa
esta convenção.
```

Exemplo 01 - Uso de Funções Casos para Teste Moodle+VPL

Para o professor criar uma atividade VPL no Moodle para este Exemplo 01, basta incluir em `Casos para teste`, o seguinte texto (pode incluir mais casos):

```
case=caso1
output=
O delta de ax^2 + bx + c é -76.0
```

- Quando uma função não tem `return` ela deve retornar o tipo `void` (nada), por exemplo, `void escrevaDelta(float d){...}`.
- Os argumentos de uma função podem ser passados por **valor** ou por **referência**.
- Por **valor**: qualquer alteração do argumento dentro da função não será passada para quem chamou a função.
- Por **referência**: oposto ao anterior, qualquer alteração dentro da função será repassada para quem chamou e isso ocorre passando o endereço de memória da variável (com `&`) ao ser chamado. Por exemplo: * `incrementaUm(&x)`
- Esse endereço de memória é definido como o tipo de dados **ponteiro** e será estudado em detalhes na parte de alocação estática vs dinâmicas.

- Para acessar o conteúdo de uma variável ponteiro, usar o prefixo `*`. Por exemplo:

  ```
  – void incrementaUm(int *x) {
        *x=*x+1;
    }
  ```

- Observe que o **ESCOPO** da função é definido por `{ ... }`.
- Experimente também essa ferramenta *online* para visualizar o fluxograma do código a seguir (copie o código e cole na ferramenta): code2flow.

[ ]:
```
%%writefile cap2ex01.c
#include <stdio.h>
// MÉTODO

float delta(float a, float b, float c) {
  float d = b * b - 4 * a * c;
  return d;
}
// PROGRAMA PRINCIPAL
int main(void) {
  float a = 5.0, b = -2.0, c = 4.0;
  printf("O delta de ax^2 + bx + c e %.1f\n", delta(a, b, c));
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap2ex01.c -o output2
./output2
```

1.8.1 Tabulação

- Um conceito muito importante em programação é a "endentação" (*indentation*) ou tabulação do código.
- Note que sempre que um bloco ou subconjunto de comandos é iniciado com `{` a tabulação é incrementada, e quando um subconjunto de comandos se encerra com `}` a tabulação é recuada.
- Isto permite visualizar claramente quando um grupo de comandos define um subprograma.
- Alguns editores de programa e a maioria das IDEs já fazem a tabulação de forma automática. Pesquise como fazer isso em uma IDE que esteja utilizando.
- Códigos sem tabulações corretas são muito difíceis de ler por outra pessoa (**os professores geralmente descontam pontos em códigos desorganizados!**).

1.9 Escopo

- Os subprogramas são programas independentes dentro do programa.

- Logo possuem variáveis próprias para armazenar seus dados.
- Estas **variáveis locais** só existem no âmbito do subprograma e só durante cada execução (chamada) do mesmo, desaparecendo (ou sendo apagadas) ao término do subprograma ou ao retornar em qualquer ponto com o comando `return`.
- **Variáveis globais**, por outro lado, podem ser criadas em um escopo hierarquicamente superior a todos os métodos/funções, desta forma permeando todos os subprogramas.
- Logo, as **variáveis globais** têm escopo em todos os métodos.

1.10 Reaproveitamento e Manutenção de Código

- Outra vantagem do uso de funções/métodos, além da capacidade de se reaproveitar o código já escrito em novos programas copiando os subprogramas desejados, é
  - a possibilidade de se atualizar os métodos sem a necessidade de alterar o código do programa principal.
- Para tanto, basta que a comunicação do método (entradas e saídas) permaneça inalterada.
- Como exemplo, utilizamos um programa com métodos para entrada e saída de dados com os métodos/funções `leia()` e `escreva()`, baseado nos exemplos anteriores.
- Para programas muito simples, como poucas linhas de código, pode ter a impressão de deixar o código mais complicado, mas a principal vantagem é o reaproveitamento de código em outros programas similares.
- Esse recurso de métodos de entrada e saída serão muito úteis nos tópicos de Vetores e Matrízes, abordados nos Capítulos 5 e 6, respectivamente,
  - quando métodos para ler e escrever um vetor/matriz poderão ser reaproveitados em várias questões.
- Para não ter muitas cópias desses métodos, é possível criar as nossas bibliotecas.

Para criar uma biblioteca em C devemos:

1. Criar um arquivo *header*, por exemplo `meusMetodos.h`, contendo as assinaturas dos métodos. Para o Exemplo 2 a seguir, esse arquivo deve conter:

```
float delta(float a, float b, float c);
float leia();
```

2. Criar um arquivo com as implementações dos métodos, incluindo no início `#include "meusMetodos.h"`.
   - Observar que não devemos utilizar `<>`, como ocorrem nas bibliotecas padrão do C, como exemplo: `#include <stdio.h>`.
3. Nos programas que irão utilizar esses métodos, incluir também no início a biblioteca criada, por exemplo: `#include "meusMetodos.h"`.
4. Para compilar e rodar, basta fazer:

```
gcc -Wall -std=c99 meusMetodos.c meusProgramas.c -o meusProgramas.o
./meusProgramas.o
```

Exemplo 02 - Uso de Funções com Entrada e Saída de Dados Casos para Teste no Moodle+VPL

Para o professor criar uma atividade VPL no Moodle para este Exemplo 02, basta incluir em `Casos para teste`, o seguinte texto (pode incluir mais casos):

```
case=caso1
input=3
4
5
output=
-44.0
case=caso2
input=3
4
2
output=
-8.0
case=caso3
input=3
5
2
output=
1.0
```

- Experimente essa ferramenta *online* para visualizar o fluxograma do código a seguir (copie o código e cole na ferramenta): code2flow.

[ ]:
```
%%writefile cap2ex02.c
#include <stdio.h>

float delta(float a, float b, float c) {
  float d = b*b-4*a*c;
  return d;
}

float leia() {
  float valor;
  printf("Entre com um valor: ");
  scanf("%f", &valor);
  return valor;
}

int main(void) {

  // ENTRADAS
```

```
  float a, b, c, d;
  a = leia();
  b = leia();
  c = leia();

  // PROCESSAMENTO
  d = delta(a, b, c);

  //SAÍDA
  printf("Delta = %.1f", d);
  return 0;
}
```

[ ]:
```
%%shell
gcc -Wall -std=c99 cap2ex02.c -o output2
./output2
```

1.11 Exercícios

Ver notebook Colab no arquivo `cap2.part2.lab.*.ipynb` (`*` é a extensão da linguagem), utilizando alguma linguagem de programação de sua preferência, organizadas em subpastas contidas de `"gen"`, na pasta do Google Drive colabs.

1.12 Revisão deste capítulo de Organização de Código

- Programas sequenciais
  - organize o seu código em três parte: > Entrada ⇒ Processamento ⇒ Saída
- Comentários
  - São úteis para outros podem ententer o seu código
- Desvios de fluxo
- Programas e subprogramas
- Funções, métodos e modularização
- Reaproveitamento e manutenção de código
  - Esses 4 últimos tópicos são muito importantes para organizar o seu código em partes
  - Fique atento ao **escopo** de uma variável **local** ou **global**
- Exercícios