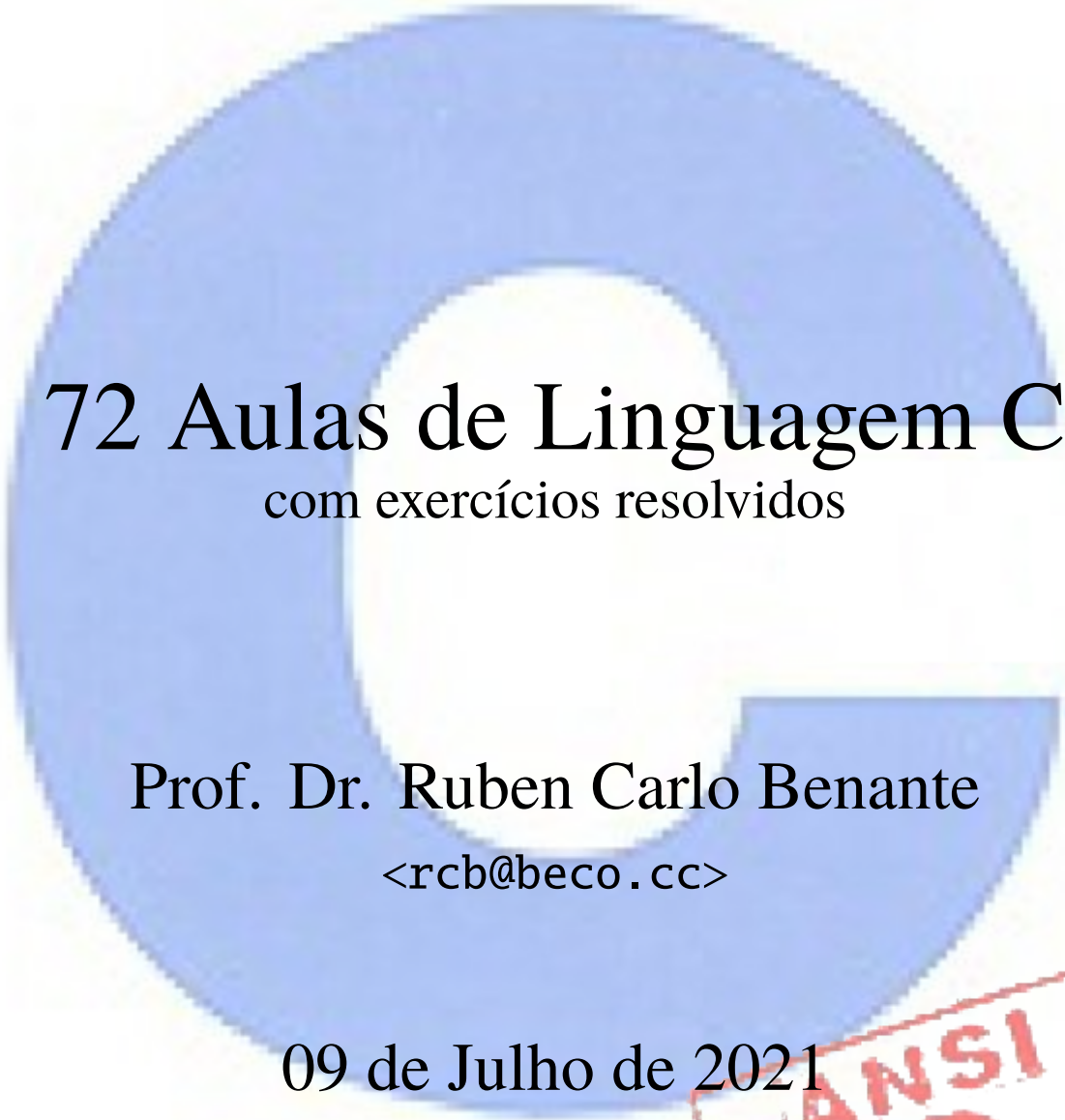

72 Aulas de Linguagem C

com exercícios resolvidos

Prof. Dr. Ruben Carlo Benante

<rcb@beco.cc>

09 de Julho de 2021

Sumário

Capítulo 1

Introdução

Página não incluída neste exemplar de amostra.

Capítulo 2

Elementos de um programa em C

A nossa primeira tarefa é aprender a identificar os elementos de um programa em C e como eles devem ser *preferencialmente* dispostos em um código fonte para padronizar e facilitar a legibilidade do mesmo. A leitura de um código é tão importante quanto sua lógica, e a padronização deve seguir boas práticas de indentação e disposição dos elementos. Desta forma o código pode ser facilmente interpretado, tanto pelo seu próprio autor, quando o ler meses (ou semanas) após sua escrita, como por outros colegas de profissão que recebam a tarefa de atualizá-lo.

Vejamos o código 2.1 para então discutirmos linha a linha estes elementos.

Cód. 2.1 : Elementos de um programa em C

```c
/* ------------------------------------------------ */
/* Program Name, Version
 * Program description and utility
 * Copyright (C) Author, Year
 * Warranty note
 * Author contact information
 * Creation and modification date
 */

/* ------------------------------------------------ */
/* includes */
#include <stdio.h> /* Standard I/O functions */
#include <stdlib.h> /* Miscellaneous functions */

/* ------------------------------------------------ */
/* defines */
#define VERSION "20190627.110330" /* Version Num. */
#define DEBUG 0 /* Activate/deactivate debug mode */
#define SBUFF 10 /* string buffer */
#define MAXCLI 100 /* Maximum of clients on memory */

/* ------------------------------------------------ */
/* typedefs, structs, etc. */
/* client register */
typedef struct sregister
{
    char nick[SBUFF];
    int id;
    float credit;
} reg_t;

/* ------------------------------------------------ */
/* function prototypes */
/* the help function does... */
void help(void);

```

```c
/* ---------------------------------------------- */
/* Brief description of the function
 * Values entered as arguments
 * Value(s) returned
 */
int main(void)
{
    /* local declarations */
    int i; /* general index */
    char s[SBUFF]; /* a string to hold ... */
    reg_t client[MAXCLI]; /* clients */

    /* your code */
    help();

    return EXIT_SUCCESS;
}

/* ---------------------------------------------- */
/* Brief description of the function
 * Values entered as arguments
 * Value(s) returned
 */
void help(void)
{
    printf("This program does...\n");
    /* add more stuff here */
    return;
}

/* ---------------------------------------------- */
/* vi:set ai et ts=4 tw=0 sw=4 wm=0 cc=54 fo=croql:*/
/* C config for Vim modeline by Dr. Beco v.190727  */

```

Discussão de cada trecho:

1 Comentário separador de seção. Você verá vários destes ao longo do código.

2-8 Cabeçalho. Deve conter: **a.** o nome do programa (ou nome do arquivo); **b.** versão; **c.** breve descrição do programa; **d.** mensagem de copyright com autor e ano; **e.** licença e garantia; **f.** informações de contato, e **g.** a data de criação e opcionalmente datas de futuras modificações.

9 Pule sempre uma linha para separar blocos distintos de código. Isso facilita a leitura. (Facilita também a navegação com o editor `vi`, usando teclas de atalho `{` e `}` para movimentar-se entre parágrafos).

10-13 Os `#include`'s são em geral os primeiros códigos que você vai adicionar, no topo do arquivo, antes de qualquer comando. Uma exceção é a biblioteca `<assert.h>` que depende do `#define NDEBUG` para se ajustar, e deve vir após o mesmo.

15-20 Seguem os `#define`'s. Nunca use os chamados *números mágicos*.

22-30 Seção para inclusão de `typedef`'s, `struct`'s, etc. Evite usar *variáveis globais*, mas em um caso extraordinário de precisar, aqui é o lugar da declaração.

32-35 Protótipos de funções (também chamados de

declaração. Em inglês você verá *prototype* ou *forward declaration*). Lembre-se de colocar um breve comentário explicando cada função. O comentário pode ser padronizado para aparecer na linha *anterior* ao protótipo, ou se forem pequenos, na mesma linha do protótipo. Não misture padrões. Uma vez escolhido, use o mesmo padrão em todo o programa e demais arquivos.

37-53 A função principal. Vamos analisar a estrutura interna de uma função:

37-41 Toda função deve ser precedida de um cabeçalho contendo: breve descrição da função, explicação sobre os argumentos de entrada e explicação sobre o valor e tipo de retorno. Se os argumentos servem de entrada e saída, acrescentar a explicação.

42 Cabeça da função (*definição*), composta por:
tipo nome(argumentos, . . .)
e aqui, diferente do protótipo, não vai `;`
A função `main` obrigatoriamente tem o tipo de retorno `int`, e os argumentos podem ser apenas uma entre duas opções: `int main(void)` para indicar que não serão lidos nenhum argumento do Sistema Operacional (SO), ou `int main(int argc, char *argv[])` para indicar que uma string `argv` contendo `argc` elementos será recebida pelo programa vinda do SO, contendo a linha de comando usada para executar o programa.

44-47 Declaração de variáveis locais. Sempre faça as declarações todas juntas e no início da função. Para cada variável declarada, inclua um comentário indicando qual será o seu uso. Use nomes claros e bem indicativos da tarefa que a variável desempenhará.

49-50 Seu código vai aqui. A função `main` não deve ter muito código. O código na função `main` deve ser pensado de modo abstrato, delegando tarefas mais detalhadas para outras funções. Tente dividir o código em 3 blocos: *leitura de dados*, *processamento* e *saída de dados*.

52 O retorno. Toda função bem escrita tem explícito pelo menos um `return`. Funções do tipo `void` teoricamente não precisam de `return`, já que ao terminar não causam nenhum *aviso de compilação*, mas é de boa prática sempre ser explícito em tudo, e portanto, usar `return`.

43-53 O *corpo da função*. Uma vez terminada a execução de sua última linha, a função retorna para o processo ou função que a chamou. No caso da função `main`, o programa termina e retorna-se ao SO.

55-65 Abaixo da função `main` inicia o trecho com as funções definidas pelo usuário. Sua estrutura é a mesma: comentário com cabeçalho explicativo, entradas e saídas; cabeça e corpo da função.

67-69 O *rodapé*. O editor de textos `vi` permite executar comandos de configuração (`modeline`) escritos em uma sintaxe pré-definida, posicionados nas 5 primeiras ou 5 últimas linhas de um arquivo. O rodapé aqui descrito configura a indentação para ter 4 espaços (não use `TAB`'s no seu código).

70 Alguns antigos programas, editores e compiladores podem reclamar se o arquivo não tiver uma linha em branco ao final. Termine sempre com uma linha em branco.

Sobre a largura da linha, a recomendação é não ultrapassar a padronização de 80 caracteres por linha. Neste livro, os códigos estão limitados a 54 para melhor visualização (veja no rodapé, linha 67, `cc=54` e troque por `cc=80`), mas uma tela permite linhas maiores.

Capítulo 3

Praticando a teoria

1 Olá mundo!

A função `printf` é normalmente a função responsável pela saída formatada de dados em um programa. Em uma primeira rápida visita a esta função, podemos começar com:

biblioteca:
```
#include <stdio.h>
```

protótipo:
```
int printf(const char *formato, ...);
```

Onde o *formato* é uma *string* a ser impressa, que contém *máscaras* especiais para a impressão de valores vindos de variáveis. As variáveis constam na lista de argumentos, após o formato, na respectiva ordem das máscaras. A função parceira

de `printf`, usada para entrada formatada de dados em um programa, é a função `scanf`, que usa um sistema quase idêntico de máscaras, com algumas poucas exceções.

Lista de máscaras mais comuns para `printf`:

`%c` Variáveis de caractere, `char`.

`%d` Variáveis inteiras com sinal, na base 10, `int`.

`%i` O mesmo que **%d**. Prefira usar **%d** para `printf`.

`%u` Variáveis inteiras sem sinal, base 10, `unsigned int`.

`%p` Imprimir endereços de ponteiros.

`%x` Imprimir inteiros convertendo para base hexadecimal.

`%o` Imprimir inteiros convertendo para base octal.

`%f` Variáveis de ponto flutuante `float` ou `double`. O `printf` não tem um formato específico apenas para `float`. Internamente sempre converte para `double`.

`%lf` Variáveis de ponto flutuante `double`. Inócuo, apenas para explicitar.

`%Lf` Para imprimir variáveis do tipo `long double`.

`%ld` Para imprimir variáveis do tipo `long int`.

%s Imprime uma `string` que deve terminar com o caractere nulo `'\0'`.

Exercício ex1.c:

Imprima "Ola mundo!" e "Eu te amo 3000 vezes"

2 Ler um valor externo para uma variável

A função `scanf` é utilizada quando se deseja ler uma entrada de dados formatada para as variáveis de um programa. Faz par com a função de entrada formatada `printf`. Um primeiro contato com esta função pode ser resumido como:

biblioteca: `#include <stdio.h>`

protótipo: `int scanf(const char *formato, ...);`

Onde o *formato* é uma *string* que define exatamente como os dados devem ser entrados, através de *máscaras* especiais para a leitura de valores para as variáveis. As variáveis devem constar na lista de argumentos, após o formato, na respectiva ordem das máscaras. As máscaras são muito parecidas com as da função `printf`, com algumas poucas exceções:

Lista de máscaras mais comuns para `scanf`:

`%c` Lê um caractere `char`.

`%d` Lê um inteiro com sinal, na base 10 `int`. Exemplo: `070` é lido como `70`, e `0x70` dá erro de leitura. A leitura ignora espaços precedentes aos números.

`%i` Diferente do `printf`, lê um inteiro com sinal, mas aceita outras bases, a depender do prefixo do número dado. Exemplo: base hexadecimal `0x70==112`, base octal `070==56` e base decimal `70==70`.

`%u` Lê valores inteiros sem sinal, base 10 `unsigned int`.

`%x` Lê um inteiro na base hexadecimal.

`%o` Lê um inteiro na base octal.

`%f` Lê um valor de ponto flutuante para `float`. Diferente do `printf`, é preciso um tipo específico para `float`, pois a leitura usa ponteiros para a variável que irá receber o dado valor.

`%lf` Lê um valor de ponto flutuante para `double`.

`%Lf` Para ler variáveis do tipo `long double`.

`%ld` Para ler variáveis do tipo `long int`.

`%s` Lê uma *string* de caracteres, terminando a leitura no primeiro espaço. Um caractere nulo `'\0'` é adicionado

ao final da *string*.

Nota importante: os termos *entrada formatada* e *saída formatada* referem-se a entrada e saída *automatizada* por algum processo, que garante a formatação. A função `scanf` não deve ser usada para *entrada de dados pelo usuário*, pois não se garante que a mesma esteja no formato codificado. Para estes primeiros exercícios, utilizaremos `scanf` para entrada do usuário, para simplificação. Mas tão logo vejamos a função `fgets` e manipulação de *strings*, devemos abandonar o `scanf` para entrada de dados de usuário e não cometer mais este equívoco.

Exercício ex2.c:

Converta graus Fahrenheit em graus Celsius

Dica: A fórmula para o inverso, de Celsius para Fahrenheit, é:

$$F = \frac{9}{5} \cdot C + 32$$

3 Comando de decisão

Uma linguagem de programação não tem sentido de existir se não permitir que o computador tome decisões e siga por diferentes caminhos para cada situação encontrada. A essa divisão do fluxo de execução se dá o nome de "bifurcação". Alguns comandos permitem que o programa siga diferentes caminhos baseado em uma *condição*. O mais básico deles é o *comando de decisão* `if-else`.

Em sua forma mais simples, tem o formato:

```
1  comando_anterior;
2  if(condicao)
3      comando_subordinado;
4  comando_posterior;
```

O `comando_anterior` ao `if` será sempre executado, pois até então vem do fluxo de execução que no exemplo é supostamente único.

Ao chegar na linha 2, o computador encontra um *comando de decisão* que poderá ou não fazê-lo executar a linha 3, a depender da *condição* ser *verdadeira* ou *falsa*:

- Sendo a *condição verdadeira*, o `comando_subordinado`

é executado, e após sua execução o fluxo segue normalmente para a linha 4, para executar o `comando_posterior`.

- Sendo a *condição falsa*, o `comando_subordinado` é *pulado*, sem ser executado, indo o fluxo diretamente para o `comando_posterior`.

Em qualquer dos casos, para enfatizar, os comandos anterior e posterior ao *if* são executados.

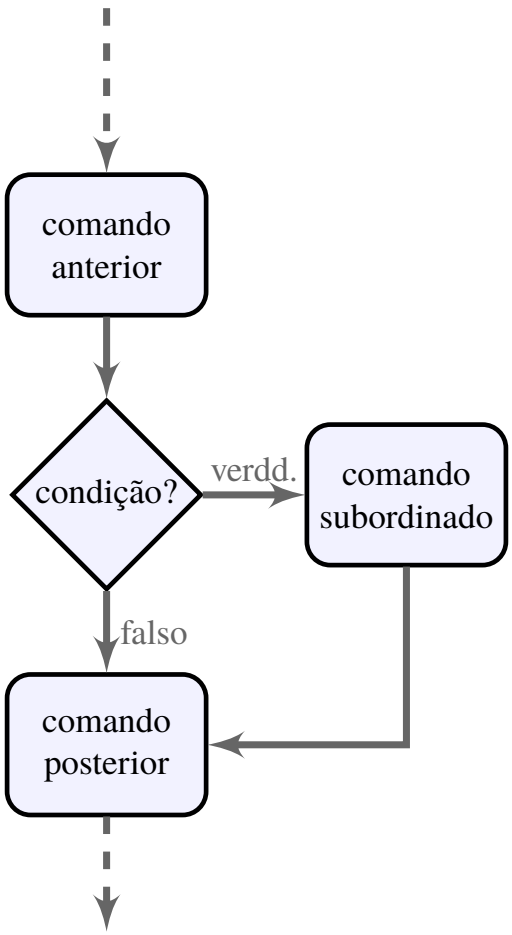

Figura 3.1: Fluxograma do comando `if`

Exercício ex3.c:

Calcule a média e imprima se foi “aprovado” ou não

4 Decisões aninhadas

Na seção anterior vimos uma introdução ao comando `if-else` (também chamado *se-então-senão*) opcionalmente sem o complemento `else`. O comando *se* pode ser pareado com um *senão* caso desejado. Os comandos subordinados ao `else` executarão quando a *condição for falsa*.

Os comandos após o fechamento do `else`, ou seja, os *comandos posteriores*, executarão sempre, independente da condição.

Veja melhor na figura 3.2.

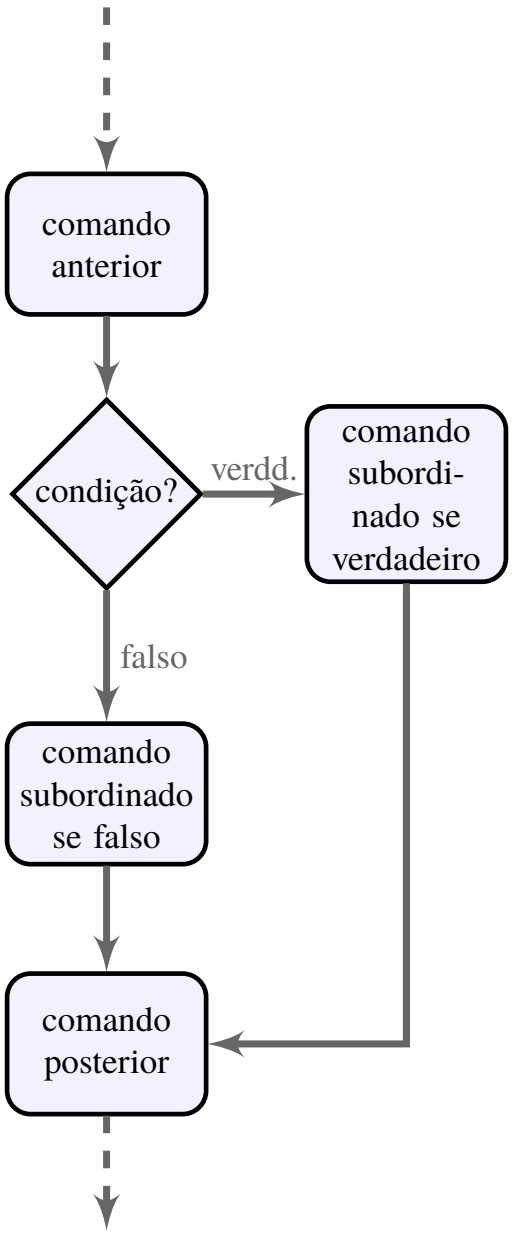

Figura 3.2: Fluxograma do comando `if-else`

Nota importante: Após comando `if`, bem como após o `else`, é reconhecido **apenas um** comando. Se for desejável colocar uma lista de comandos, é necessário abrir e fechar chaves agrupando-os: `{lista_de_comandos...}`

Veja o exemplo:

```
comando_anterior;
if(condicao)
{
    comando_subordinado_caso_verdadeiro_1;
    comando_subordinado_caso_verdadeiro_2;
}
else
{
    comando_subordinado_caso_falso_1;
    comando_subordinado_caso_falso_2;
}
comando_posterior;
```

Exercício ex4.c:

Leia 3 inteiros e imprima o maior

5 Conectivos lógicos

Página não incluída neste exemplar.

Capítulo 4

Soluções dos Exercícios

ex1.c: Imprima "Ola mundo!" e "Eu te amo 3000 vezes"

Cód. 4.1 : Imprima "Ola mundo!" e "Eu te amo 3000 vezes"

```
1  /* ------------------------------------------- */
2  /* ex1.c
3   * Imprimir "Ola mundo!" e "Eu te amo 3000 vezes."
4   * Copyright (C) Ruben Carlo Benante, 2019
5   * License GNU/GPL v2.0
6   * This program comes without ANY WARRANTY
7   * Author: Ruben Carlo Benante <rcb@beco.cc>
8   * Date: 2019-06-29
9   */
10
11 /* ------------------------------------------- */
12 /* includes */
13 #include <stdio.h> /* Standard I/O functions */
```

```
14  #include <stdlib.h> /* Miscellaneous functions */
15
16  /* ---------------------------------------------- */
17  /* defines */
18  #define AMOVEZ 3000
19
20  /* ---------------------------------------------- */
21  /* Descricao: Imprime a mensagem e termina
22   * Entrada: Sem argumentos de entrada
23   * Saida: Retorna sucesso ao final
24   */
25  int main(void)
26  {
27      printf("Ola mundo!\n");
28      printf("Eu te amo %d vezes.\n", AMOVEZ);
29      return EXIT_SUCCESS;
30  }
31
32  /* ---------------------------------------------- */
33  /* vi:set ai et ts=4 tw=0 sw=4 wm=0 cc=54 fo=croql:*/
34  /* C config for Vim modeline by Dr. Beco v.190727  */
35
```

Lições deste exercício

Neste exercício primeiro exercício tiramos como lição os aspectos de estrutura e legibilidade de código. Vamos ressaltar alguns itens importantes:

Trechos que merecem apreciação:

1-35 Repare bem na *indentação*. Indentar o código

é colocá-lo em sequência de comandos, respeitando quais comandos são subordinados a quais outros. Neste código existem apenas 3 comandos subordinados à função `main`, nas linhas 27, 28 e 29, e eles estão indentados com 4 espaços em branco para a direita.

Neste livro usamos a indentação conhecida como `all man`, (também chamada de estilo `bsd`, estilo `break` ou `-A1`). Veja a documentação do comando linux:
`$ astyle -A1`
Claro, você deverá sempre seguir as recomendações da empresa onde for contratado. Nada pior que uma guerra de estilos de indentação nos *commits* do *git*.

18 Este `#define` cria a *macro* `AMOVEZ` e evita o chamado *número mágico*. Adicionar números diretamente no código sem uso de macros é garantia de dor-de-cabeça caso seja preciso alterar os valores. Além disso, valores iguais podem representar coisas diferentes (por exemplo um formulário em um vetor com 11 valores e um CPF com 11 dígitos). Então uma simples busca/substituição de valores num código grande não é tão simples. Resumindo: sempre use *macros*

14, 29 A macro `EXIT_SUCCESS` é declarada na biblioteca `#include <stdlib.h>`. O seu valor é normalmente 0 na maioria dos sistemas operacionais. O valor 0 retornado pela função `main` representa sucesso, e valores de 1 a 255

são códigos de erros específicos de cada sistema, ou definidos pelo programador. Mas como isso é dependente de fatores fora da linguagem C, não é garantida a unicidade dos códigos de erro. Por isso, usar a macro ao invés de retornar um valor fixo (outro exemplo de *número mágico*) é importante.

3, 21, 23, 27 Não use acentos no código, nem em comentários. Palavras como "descrição", "Olá", ou mensagens como "O resultado é...", apenas irão transformar a saída do seu código em um amontoado de caracteres incompatíveis com outros programas. Seu código deve usar apenas caracteres da tabela *ASCII*, e evitar problemas de internacionalização. Hoje o mundo está interligado e você não quer desperdiçar suas oportunidades lá fora por ter código ilegível ou que não compila. Se for necessário usar acentos, use um arquivo separado de tradução e codificação `UTF-8` e a ferramenta `gnu/gettext`.

25 A função `main` é declarada como retornando `int`, e pode receber 0 ou 2 argumentos, em um dos dois formatos: `int main(void)` para 0 argumentos, e `int main(int argc, char *argv[])` , para 2 argumentos. Usaremos a opção sem argumentos na maioria dos exercícios, mas veremos como utilizar a segunda opção mais à frente. Não existem outras opções além destas duas pré-definidas.

Como melhorar este código

É possível alterar as linhas 27 e 28 para fazer uma única chamada para a função `printf()`. Esta função é considerada *pesada* (pesquise sobre *function call overhead*), e se pudermos evitar chamá-la, melhor. Pode parecer um custo insignificativo neste exemplo, mas para se ter uma ideia comparativa: rodar um laço com 1 bilhão de adições diretas leva $1.4s$ em um processador $i5$ de $2.27GHz$, enquanto fazer o mesmo cálculo chamando uma função leva $4.4s$. Um aumento de 215% (ou um *overhead* de 3 nanosegundos por chamada).

Uma primeira opção seria trocar ambas chamadas por um único `printf()` assim:

```
1    printf("Ola mundo!\nEu te amo %d vezes.\n", AMOVEZ);
```

Uma segunda opção dada abaixo, mais legível, aproveita a característica da linguagem C de *concatenar* (unir) *strings* consecutivas. Veja no exemplo abaixo as linhas 27 e 28. O compilador considera ambas *strings* como sendo uma só, o que é equivalente ao exemplo acima.

Cód. 4.1v2: Imprima "Ola mundo!" e "Eu te amo 3000 vezes"

```
25  int main(void)
26  {
27      printf("Ola mundo!\n"
28             "Eu te amo %d vezes.\n", AMOVEZ);
29      return EXIT_SUCCESS;
30  }
```

Como piorar este código

• Não declare o tipo de retorno da função `main(void)`, removendo o `int`.

• Não explicite que a função `main()` não receberá argumentos, removendo o `void`

• Confunda a indentação da abertura da chave após a função `main(){`, ao invés de alinhá-la com seu par que a fecha, usando uma das horríveis indentações estilo `-A2` ou `-A3`.

• Use *números mágicos* no seu código, e economize uma linha de `#define`.

• Não use nenhum `return` ao final da função, ou pior, use um `return;` com o *tipo* incorreto. O *tipo* de retorno deve sempre casar com o *tipo* da função, certo? Um `return;` sem valor deveria ser usado apenas para funções do tipo `void`.

Aqui a nova versão:

Cód. 4.1v3: Imprima "Ola mundo!" e "Eu te amo 3000 vezes"

```
1  #include <stdio.h>
2  main() {
3      printf("Ola mundo!\n");
4      printf("Eu te amo %d vezes.\n", 3000);
5  }
```

O código até "parece" bem feito, e garanto que já vi livros assim. Mas não se engane, este não é o jeito certo de fazer do seu código um belo *artesanato*. Programar é uma *arte*, e como tal deve ser tratada.

ex2.c: Converta graus Fahrenheit em graus Celsius

Cód. 4.2 : Converta graus Fahrenheit em graus Celsius

```
1  /* ------------------------------------------- */
2  /* ex2.c
3   * Converter um valor flutuante de graus Fahrenheit
4   *     para graus Celsius
5   * Copyright (C) Ruben Carlo Benante, 2019
6   * Licenca GNU/GPL v2.0
7   * Este programa nao tem QUALQUER GARANTIA
8   * Autor: Ruben Carlo Benante <rcb@beco.cc>
9   * Data: 2019-06-30
10  */
```

```
11
12  /* ------------------------------------------------ */
13  /* includes */
14  #include <stdio.h> /* Funcoes de entrada/saida */
15  #include <stdlib.h> /* Funcoes Miscelaneas */
16
17  /* ------------------------------------------------ */
18  /* prototipos de funcoes */
19  /* le um float e o retorna */
20  float le_f(void);
21  /* Converte Fahrenheit para Celsius */
22  float converte_fc(float f);
23
24  /* ------------------------------------------------ */
25  /* Chama funcao de leitura de dados,
26   * funcao de processamento de dados e finalmente
27   * imprime o resultado.
28   * Sem argumentos de entrada
29   * Retorna sucesso ao final
30   */
31  int main(void)
32  {
33      float f, c; /* fahrenheit e celsius */
34
35      /* entrada de dados */
36      printf("Digite o valor em Fahrenheit: ");
37      f=le_f();
38      /* processamento de dados */
39      c=converte_fc(f);
40      /* saida de dados */
41      printf("Celsius: %f\n", c);
42
43      return EXIT_SUCCESS;
44  }
45
46  /* ------------------------------------------------ */
47  /* Le um valor flutuante do teclado
48   * Entrada: nao ha
```

```
 * Retorno: o valor lido
 */
float le_f(void)
{
    float f;

    scanf("%f", &f);

    return f;
}

/* ---------------------------------------------- */
/* Converte Fahrenheit para Celsius
 * Entrada: um valor float em fahrenheit
 * Saida: o valor convertido para celsius
 */
float converte_fc(float f)
{
    float c;

    c = 5.0/9.0 * (f - 32.0);

    return c;
}

/* ---------------------------------------------- */
/* vi:set ai et ts=4 tw=0 sw=4 wm=0 cc=54 fo=croql:*/
/* C config for Vim modeline by Dr. Beco v.190727  */

```

Lições deste exercício

Uma das lições mais importantes desse código é como a função `main` é tratada. Você deve evitar jogar código de baixo

nível na função principal.

Pense em um programa como um longo livro, em que os capítulos são as funções e a função principal é uma mistura de resumo com sumário. Na função principal você deve levar o *leitor* a entender todo o programa apenas lendo, na ordem, as chamadas das funções. A tarefa mais importante da função `main` é delegar tarefas, em uma ordem lógica, quebrando o problema em *objetivos*, para uma primeira camada de funções um nível *menos* abstrata.

Esse segundo nível de funções vão por sua vez quebrar os objetivos em *sub-objetivos*, e assim por diante, indo do abstrato para o código que realmente *trabalha* e faz as coisas, o código *real*.

O programa deve ser organizado de forma lógica, com as funções mais abstratas mais acima, e as funções mais detalhistas ao final. Se preciso, cria-se outros arquivos paras as funções, chamados bibliotecas, que veremos em exercícios futuros.

Os nomes das funções devem ser claros e um bom indicativo do que elas fazem. Use alguma padronização para nomear as variáveis, e mantenha o seu padrão em todo o código.

Nota importante: **Entrada / Processamento / Saída**:

sempre que possível, divida o programa nestas três etapas, e tente não misturá-las. Faça toda a entrada de dados necessária de uma só vez, liberando o seu usuário para ir tomar um café ao invés de ter que ficar esperando o seu programa fazer algum processamento (que pode demorar horas) e responder a mais entradas de dados intercaladas. Você como programador, deve pensar no bem-estar de quem vai usar o seu programa, tornar a vida *deles* mais fácil. Os usuários com certeza vão apreciar como você pensou no conforto deles, ao invés de ter pensado em programar do jeito mais fácil e terminar logo o programa.

Como melhorar este código

Um dos pontos fracos deste programa é mesmo a leitura de dados do usuário utilizando `scanf`. Esse erro comum, que se vê em muitos livros de introdução à linguagem C, deve ser evitado. Normalmente os livros ensinam em uma ordem pedagógica, aumentando a complexidade dos exemplos, e isso causa o problema, já que o correto seria fazer entrada de dados de usuário com *strings*, mas *strings* em C são um tanto complexas e sempre ficam para depois.

Vamos resolver isso imediatamente, já no exercício 2, utilizando uma leitura de dados com *strings* e convertendo o valor para *ponto flutuante*, criando assim uma *validação* dos

dados de entrada. Neste momento não vamos explicar *strings*, apenas utilizá-las de modo intuitivo. E para utilizar uma *string* precisamos definir o seu tamanho máximo, o que faremos com um `#define MAXUIS 10`. Note que nesse código o tamanho máximo da *string* ainda não está sendo respeitado, o que é um erro grave. Mas resolveremos isso nos exercícios futuros.

Outras pequenas melhoras podem ser feitas, em especial no nome das variáveis. Se é um programa pequeno, os nomes atuais estão bons. Mas se uma variável como a temperatura Fahrenheit será usada e referenciada em diversos pontos do programa, você vai preferir dar um nome melhor a ela que simplesmente `f` (raciocínio análogo para a variável Celsius ou outras).

É comum hoje usar mais o tipo `double` do que o tipo `float`, por causa do aumento da capacidade de memória e velocidade dos modernos computadores. Mas se o seu programa vai tratar por exemplo de unidade monetária que só utiliza centavos, não faz sentido usar `double`. Além disso, cálculos com `float` são muito mais rápidos. Então entenda a precisão necessária para seu programa e use o tipo adequado conforme sua preferência. Lembre-se que as funções na biblioteca `<math.h>` retornam `double`.

Repare na linha 45 e 46 que foi utilizado um caractere de tabulação `'\t'`. Se estivermos em um laço e desejarmos

imprimir uma lista de dados, usar tabulação nos permite, por exemplo, importar os dados para outros programas como o *gnuplot* ou mesmo uma planilha. A tabulação é um separador natural de dados em colunas.

A função de entrada foi modificada para evitar problemas de leitura da entrada de usuário. Esta versão já ajuda no problema de travamento que o `scanf` apresenta quando se pede números e o usuário digita letras ou outros *lixos*. Mas ainda não está perfeita, pois não previne *buffer overflow*. Veremos isso adiante.

A função `sscanf`, em uma explicação simples, *passa* o valor encontrado na variável `user_input` para a variável `d`. Pense no primeiro "s" de `sscanf` como indicativo de entrada via *string*, assim como o "f" de `fscanf` indica entrada via "arquivo" (*file*).

Assim, podemos considerar estas modificações:

Cód. 4.2v2: Converta graus Fahrenheit em graus Celsius

```
1  /* ------------------------------------------- */
2  /* ex2v2.c
3   * Converter um valor flutuante de graus Fahrenheit
4   *     para graus Celsius, versao 2.0
5   * Copyright (C) Ruben Carlo Benante, 2019
6   * Licenca GNU/GPL v2.0
7   * Este programa nao tem QUALQUER GARANTIA
8   * Autor: Ruben Carlo Benante <rcb@beco.cc>
```

```
 * Data: 2019-06-30
 */

/* ---------------------------------------------- */
/* includes */
#include <stdio.h> /* Funcoes de entrada/saida */
#include <stdlib.h> /* Funcoes Miscelaneas */

/* ---------------------------------------------- */
/* defines */
#define MAXUIS 10 /* Tamanho maximo da string */

/* ---------------------------------------------- */
/* prototipos de funcoes */
/* le um float e o retorna */
double le_double(void);
/* Converte Fahrenheit para Celsius */
double fahrenheit_celsius(double f);

/* ---------------------------------------------- */
/* Chama funcao de leitura de dados,
 * funcao de processamento de dados e finalmente
 * imprime o resultado.
 * Sem argumentos de entrada
 * Retorna sucesso ao final
 */
int main(void)
{
    double fah, cel; /* fahrenheit e celsius */

    /* entrada de dados */
    printf("Digite o valor em Fahrenheit: ");
    fah=le_double();
    /* processamento de dados */
    cel=fahrenheit_celsius(fah);
    /* saida de dados */
    printf("Fahrenheit:\tCelsius:\n");
    printf("%.4lf\t%.4lf\n", fah, cel);
```

```
47
48      return EXIT_SUCCESS;
49  }
50
51  /* ---------------------------------------------- */
52  /* Le um valor flutuante do teclado
53   * Entrada: nao ha
54   * Retorno: o valor lido
55   */
56  double le_double(void)
57  {
58      char user_input[MAXUIS]; /* entrada do usuario */
59      double d; /* double convertido da entrada */
60
61      scanf("%s", user_input);
62      sscanf(user_input, "%lf", &d);
63
64      return d;
65  }
66
67  /* ---------------------------------------------- */
68  /* Converte Fahrenheit para Celsius
69   * Entrada: um valor double em fahrenheit
70   * Saida: o valor double convertido para celsius
71   */
72  double fahrenheit_celsius(double fah)
73  {
74      double cel;
75
76      cel = 5.0/9.0 * (fah - 32.0);
77
78      return cel;
79  }
80
81  /* ---------------------------------------------- */
82  /* vi:set ai et ts=4 tw=0 sw=4 wm=0 cc=54 fo=croql:*/
83  /* C config for Vim modeline by Dr. Beco v.190727  */
84
```

Como piorar este código

1. Para um teste inicial, experimente executar o programa e entrar com um valor maior que 9 dígitos. O *string* criada é um vetor de 10 posições, mas a última posição de uma *string* deve ser reservada para o caractere *nulo*, o que efetivamente deixa a *string* com no máximo 9 letras. Em alguns compiladores, talvez não dê erro usar um número de 10 dígitos, por pura coincidência, já que você não estará escrevendo fora do vetor na função `scanf`. Mas você não terá deixado espaço para o caractere nulo terminar a *string*, e a função `sscanf` não saberá onde parar. Digitando além de 10 dígitos, porém, você garante ver o temido *segmentation fault*. Este erro é simples de entender: você tentou usar memória fora do "segmento" de memória que lhe é autorizado. Isso ocorre muito com ponteiros, mas vetores com índice fora de faixa também caem neste caso.

2. Use variáveis globais ao invés de variáveis locais. Uma ótima fonte de erros. Se você quer *piorar* um código, esta é a melhor recomendação e vale para qualquer programa. Para declarar as variáveis `f` e `c` da versão 1 como globais, mova a linha 33 `float f, c; /*fahrenheit e celsius */` para fora e acima da função `main`, por exemplo para a linha 16 (veja ca-

pítulo 2, onde declarar variáveis globais). O grande problema das variáveis globais é que elas quebram uma das propriedades mais importantes para um bom código: *encapsulamento*! Isto é: os dados são protegidos de interferências externas, podendo ser alterados apenas dentro das funções que lhes competem.

ex3.c: Calcule a média e imprima se foi "aprovado" ou não

Cód. 4.3 : Calcule a média e imprima se foi "aprovado" ou não

```
1  /* ---------------------------------------------- */
2  /* ex3.c
3   * Calcular a media de duas notas e imprimir se o
4   *    aluno foi aprovado ou nao
5   * Copyright (C) Ruben Carlo Benante, 2019
6   * Licenca GNU/GPL v2.0
7   * Este programa nao tem qualquer GARANTIA
8   * Autor: Ruben Carlo Benante <rcb@beco.cc>
9   * Data: 2019-06-30
10  */
11
12 /* ---------------------------------------------- */
13 /* includes */
14 #include <stdio.h> /* Standard I/O functions */
15 #include <stdlib.h> /* Miscellaneous functions */
16
17 /* ---------------------------------------------- */
18 /* defines */
```

```c
#define CORTE 5.0

/* ------------------------------------------------ */
/* prototipos de funcoes */
/* le um float e o retorna */
float leia_f(void);
/* Calcula e retorna a media entre dois valores  */
float media(float, float);

/* ------------------------------------------------ */
/* Chama funcao de leitura de dados,
 * funcao de processamento de dados e finalmente
 * imprime se foi aprovado ou nao
 * Sem argumentos de entrada
 * Retorna sucesso ao final
 */
int main(void)
{
    float a, /* primeira nota */
          b, /* segunda nota */
          m; /* media */

    /* entrada de dados */
    printf("Digite a primeira nota: ");
    a=leia_f();
    printf("Digite a segunda nota: ");
    b=leia_f();
    /* processamento de dados */
    m=media(a, b);
    /* saida de dados */
    printf("Sua media: %.2f\nVoce ", m);
    if(m<CORTE)
        printf("nao ");
    printf("foi aprovado\n");

    return EXIT_SUCCESS;
}

```

```c
/* ---------------------------------------------- */
/* Le um valor flutuante do teclado
 * Entrada: nao ha
 * Retorno: o valor lido
 */
float leia_f(void)
{
    float f;

    scanf("%f", &f);

    return f;
}

/* ---------------------------------------------- */
/* Calcula a media entre dois valores
 * Entrada: duas notas, float
 * Saida: a media entre as notas, float
 */
float media(float n1, float n2)
{
    float m;

    m = (n1 + n2)/2.0;

    return m;
}

/* ---------------------------------------------- */
/* vi:set ai et ts=4 tw=0 sw=4 wm=0 cc=54 fo=croql:*/
/* C config for Vim modeline by Dr. Beco v.190727  */

```

Lições deste exercício

• Repare que o `#define CORTE 5.0` declara `CORTE` como sendo um valor de *ponto flutuante*. Colocar uma casa decimal em um número é uma forma simples e eficaz de convertê-lo a `float`. Isso é especialmente importante nas divisões, pois a linguagem C diferencia divisão de inteiros e divisão de reais.

• Veja na linha 80 a fórmula da média, onde não se faz divisão por 2 e sim por 2.0, garantindo assim que a divisão arredonde o valor (bem, tecnicamente a divisão inteira em C "trunca" o valor, não arredonda). Mantenha essa prática sempre, para evitar dissabores. Lembre-se: seja sempre explícito no que deseja, e nunca conte com comportamentos padrão do compilador.

• Novamente a função principal é um *gerente* das funções secundárias, chamando o código em uma ordem lógica.

• A indentação na declaração das variáveis nas linhas 37, 38, 39, permitindo comentários individuais.

• O modo como a função `printf` se comporta ao encontrar um caractere de mudança de linha `'\n'`.

• Detalhe: a máscara `%.2f` indica ao `printf` para imprimir apenas duas casas decimais do `float`.

• Como a função `media` recebe o conteúdo de duas variáveis em *passagem por valor*, e retorna um valor do tipo `float`.

Como melhorar este código

Se você solucionou este exercício com código semelhante ao mostrado abaixo, está ótimo. A clareza às vezes é melhor que a economia, principalmente se o trecho não está dentro de um laço.

```
49      printf("Sua media: %.2f\n", m);
50      if(m>=CORTE)
51          printf("Voce foi aprovado\n");
52      else
53          printf("Voce foi reprovado\n");
54
55      return EXIT_SUCCESS;
56  }
```

Quando se programa pensando em usar posteriormente técnicas de *tradução de texto*, para que o seu programa possa ser apresentado em vários idiomas, é melhor ter frases completas. Idiomas diferentes não compartilham dos mesmos *atalhos lógicos* possíveis na língua portuguesa.

Além disso, como vimos no exercício anterior, não se deve usar `scanf` para entrada de usuário. Melhor ler uma *string* e depois extrair o valor numérico dela sem interferências externas. Mas se o `scanf("%f", &f)` causa problemas se o usuário digitar letras ao invés de números, e o `scanf("%s", s)` causa problemas se o usuário digitar uma *string* maior que o tamanho do *buffer* alocado, qual a solução?

A solução vem na função `fgets(string, tamanho, entrada);` A função `fgets` recebe como parâmetro o tamanho do *buffer*, e com isso garante que a variável não vai "estourar" (*buffer overflow*). Uma melhoria na função `leia_f()` anterior pode ser assim implementada:

```
62  double le_double(void)
63  {
64      char ui[MAXUIS]; /* user input em uma string */
65      double d;
66
67      fgets(ui, MAXUIS, stdin);
68      sscanf(ui, "%lf", &d);
69
70      return d;
71  }
```

E claro, deve-se colocar o respectivo `#define MAXUIS 10` após o `#define CORTE 5.0` , na linha 20, e corrigir o protótipo da função.

Como piorar este código

Com a versão dada acima da função `le_double`, corrigimos o problema da entrada de dados por usuário, finalmente! A função `fgets` tem o nome derivado da função `gets`, onde o *f* inicial indica leitura de arquivo (como também é em `fprintf` e `fscanf`).

Para arruinar nossa solução de vez e voltarmos a ter problema com estouro de *buffer*, basta trocar `fgets` pela sua versão perigosa e depreciada, `gets`. Ela é mais simples de usar, basta substituir a linha 67 por `gets(ui);` e pronto. Está feita a lambança.

Para experimentar com o truncamento da divisão por inteiros, faça as seguintes modificações na função `media`:

1. Troque o tipo dos argumentos para `int`. Lembre-se de trocar também o protótipo (mas antes compile para ver o erro e se acostume com as mensagens de erro do *gcc*).

2. Experimente rodar o programa com a divisão por 2 e com 2.0 e veja o resultado.

```
77  /* float media(float n1, float n2) */
78  float media(int n1, int n2)
79  {
```

```
80      float m;
81
82      m = (n1 + n2)/2.0; /* troque para 2 */
83
84      return m;
85  }
```

Garantido: depois que você quebrar a cabeça por horas para descobrir um `BUG` como esse, você nunca mais esquecerá quando quer dividir com casas decimais.

ex4.c: Leia 3 inteiros e imprima o maior

Cód. 4.4 : Leia 3 inteiros e imprima o maior

```
1   /* ------------------------------------------------ */
2   /* ex4.c
3    * Ler tres valores e imprimir o maior, v1.0
4    * Copyright (C) Ruben Carlo Benante, 2019
5    * Licenca GNU/GPL v2.0
6    * Este programa nao tem NENHUMA GARANTIA
7    * Author: Ruben Carlo Benante <rcb@beco.cc>
8    * Date: 2019-07-01
9    */
10
11  /* ------------------------------------------------ */
12  /* includes */
13  #include <stdio.h> /* Standard I/O functions */
14  #include <stdlib.h> /* Miscellaneous functions */
15
16  /* ------------------------------------------------ */
17  /* Le tres numeros e imprime o maior
```

```c
 * Sem argumentos de entrada
 * Retorna sucesso ao final
 */
int main(void)
{
    int a, b, c; /* os tres valores lidos */
    int m; /* o maior */

    printf("Digite 3 valores:\n");
    scanf("%d %d %d", &a, &b, &c);

    if(a>b)
        if(a>c)
            m=a;
        else
            m=c;
    else
        if(b>c)
            m=b;
        else
            m=c;

    printf("O maior valor: %d\n", m);

    return EXIT_SUCCESS;
}

/* ---------------------------------------------- */
/* vi:set ai et ts=4 tw=0 sw=4 wm=0 cc=54 fo=croql:*/
/* C config for Vim modeline by Dr. Beco v.190727  */

```

Lições deste exercício

Nota: para economia de espaço, já entendidos os elementos de um programa em C, não iremos mais imprimir o cabeçalho, rodapé, e outros elementos desnecessários ao entendimento do código.

Nota importante: O comando `else` sempre fará par com o `if` imediatamente anterior. Caso não seja este o par correto, é preciso indicar explicitamente usando chaves. Veja na seção abaixo como as chaves das linhas 29 e 32 isolam o `if` subordinado, separando-o do próximo `else`.

Como melhorar este código

Segue uma nova versão da solução, com duas atribuições a menos (`else m=c;`).

Cód. 4.4v2: Leia 3 inteiros e imprima o maior

```
21  int main(void)
22  {
23      int a, b, c; /* os 3 valores lidos */
24
25      printf("Digite 3 valores:\n");
26      scanf("%d %d %d", &a, &b, &c);
27
28      if(a>b)
```

```
29      {
30          if(a>c)
31              c=a;
32      }
33      else
34          if(b>c)
35              c=b;
36
37      printf("O maior valor: %d\n", c);
38
39      return EXIT_SUCCESS;
40  }
```

Como piorar este código

- Não use indentação. A linguagem C não se importa com inutilidades como espaços em branco e mudanças de linha.

- Remova comentários também. Quem precisa ler comentários em código?

- A biblioteca `#include <stdlib.h>` está sendo usada apenas para declarar a macro `EXIT_SUCCESS`. Desnecessário, não é? Remova ambas, já que na *maioria* dos sistemas `EXIT_SUCCESS` é igual a 0. Não é garantido, mas se funciona no seu computador para que duvidar que funcionará no computador do cliente, certo?

Aqui a nova versão (programa completo, em apenas 2 linhas):

Cód. 4.4v3: Leia 3 inteiros e imprima o maior

```
1  #include <stdio.h>
2  int main(void) {int a, b, c; printf("Digite 3 valores
       :\n"); scanf("%d %d %d", &a, &b, &c); if(a>b) {if
       (a>c) c=a;} else if(b>c) c=b; printf("O maior
       valor: %d\n", c); return 0;}
```

ex5.c: Leia 3 inteiros e imprima-os em ordem

Fim do exemplar de amostra.

Benante, Ruben Carlo. *72 Aulas de Linguagem C: com exercícios resolvidos*. Recife: O Autor, 2021. 52 p.

Entrada BibTeX:

```
@BOOK{Benante2021,
      author = {Ruben Carlo Benante},
      title = {72 Aulas de Linguagem C: com
               exercícios resolvidos},
      publisher = {O Autor},
      year = {2021},
      address = {Recife},
      note = {UNI2021}
}
```

Tipografado em LaTeX, com *Kile 2.9*, em um sistema `Debian Gnu/Linux`.

A fonte do corpo do texto é a Times New Roman 10pt. Preparado para *ebook* (ratio 3 : 4).

Preâmbulo LaTeX:

```
\documentclass[10pt,openany]{book}
\usepackage[paperwidth=9cm, paperheight=12cm, top=0.38cm, bottom=0.71cm,
           left=0.25cm, right=0.15cm, asymmetric, head=0pt, foot=7.801]{geometry}
\usepackage[noinfo,center,pdflatex,cross]{crop}
\linespread{1.01}
\setlength{\parskip}{.6\baselineskip}
```