Guia de Estilos para ICC I

Este documento tem como objetivo apresentar definições e exemplos das chamadas "boas práticas" no desenvolvimento de programas. A intensão é melhorar a legibilidade dos códigos escritos pelos alunos e facilitar a correção dos trabalhos por parte dos monitores.

> ⚠ **ATENÇÃO:** O descumprimento das regras estabelecidas aqui pode acarretar em uma diminuição na nota final da atividade em alguns casos.

Linhas

Uma linha de código deve expressar com clareza sua funcionalidade e, portanto, deve ser de fácil leitura e executar somente uma tarefa.

1. Comandos

Com excessão de situações específicas, como as componentes internas de um `for()`, declaração de variáveis ou o uso dos operadores `++` e `--`, uma linha deve conter **uma e só uma instrução**, tendo somente um `;` no final.

2. Tamanho

Apesar de raros, existem casos, por exemplo na passagem de diversos parâmetros para uma função, onde uma linha pode passar dos **120 caracteres**, ou sendo mais conservador, dos **80 caracteres**; nessas situações recomenda-se quebrar a linha e indentá-la de forma a alinhar as parcelas equivalentes.

```
// Linha com 93 caracteres
linhas = (char **) realloc(linhas, (numero_linhas + BUFFER_REALLOC_LINHAS)
                                   * sizeof(char *));
```

Blocos

Qualquer programa é composto por diversas instruções que podem e devem ser agrupadas para um melhor entendimento e re-uso do código fonte.

1. Blocos reais

Um bloco de comandos bem definido pela linguagem e que será executado somente nas circunstâncias corretas, por exemplo, uma função, loop ou condicional. Em C é possível identificar esses casos quando é necessário envolver as instruções em `{}`. Os comandos internos a um bloco real devem ser indentados.

```
while (*string != '\0') {
    // Código interno ao bloco real: while()
    printf("%c", *string);
    string++;
}
```

2. Blocos lógicos

Para facilitar a leitura de uma seção mais longa, como a função `main()` por exemplo, é recomendável a separação das etapas de execução em blocos lógicos. Os grupos devem ser separados com uma linha em branco e normalmente são acompanhados de um comentário explicando sua função.

```
// Bloco lógico de leitura
for (int i = 0; i < tamanho; i++) {
    scanf("%i", &vetor[i]);
}

inverte_vetor(vetor, tamanho);

// Bloco lógico de impressão
printf("Vetor invertido:");
for (int i = 0; i < tamanho; i++) {
```

```
    printf(" %i", vetor[i]);
}
```

Chaves

Na linguagem C, muitas vezes é necessário explicitar os limites de um bloco utilizando `{}`. Nesse contexto, não há um padrão único, o mais importante é manter a consistência dentro de um programa, não misturando os padrões em um mesmo código.

1. Utilizando chaves

Existem três formas corretas de se posicionar chaves em um código exemplificas a seguir.

```
// Toda chave com linha própria
if (nota < 3.0)
{
    printf("Reprovado.\n");
}
else if (nota < 5.0)
{
    printf("Recuperacao.\n");
}
else
{
    printf("Aprovado.\n");
}
```

```
// Chaves de fechamento com linha própria
if (nota < 3.0) {
    printf("Reprovado.\n");
}
else if (nota < 5.0) {
    printf("Recuperacao.\n");
}
else {
    printf("Aprovado.\n");
}
```

```
// Chave final com linha própria
if (nota < 3.0) {
    printf("Reprovado.\n");
} else if (nota < 5.0) {
    printf("Recuperacao.\n");
} else {
    printf("Aprovado.\n");
}
```

É importante lembrar que o número de linhas de um arquivo não é indicador definitivo de qualidade, os exemplos anteriores têm total de linhas diferentes, mas são igualmente válidos.

2. Eliminando chaves

Na literatura sobre o assunto, a não utilização de chaves na identificação de blocos é considerada por vezes uma má prática, portanto tome como opcional essa seção e aplique suas dicas com cautela.

Caso um bloco seja formado por um único comando é possível dispensar as chaves, isso pode deixar, por vezes, a estrutura mais limpa.

```
for (int i = 0; i < numero_frases; i++) {
    // Bloco sem chaves
    if (frases[i] == NULL) continue;

    printf("%s\n", frases[i]);
}
```

Se a linha ficar muito grande ou caso a condicional seja composta, é recomendado que a instrução seja colocada na linha posterior e indentada, mesmo que as chaves não estejam presentes.

```
if (nota < 3.0)
    printf("Reprovado.\n");
else if (nota < 5.0)
    printf("Recuperacao.\n");
```

```
else
    printf("Aprovado.\n");
```

Indentação

É de extrema importância que um bloco real de código seja identificado com facilidade, para isso existe a indentação que se basea na inclusão de espaços em branco no início de cada linha.

1. Tamanho

Pode-se indentar uma linha utilizando "Tab"s ou espaços, mas em qualquer um dos casos, recomenda-se que o tamanho vazio seja equivalente a **4 espaços**, o tamanho padrão de um "Tab" na maioria dos editores, mas é possível alterar essa configuração.

```
do {
    // Código indentado com 4 espaços
    fread(&funcionario, sizeof(struct Funcionario), 1, contratados);
    imprime_funcionario(funcionario);
}
while (!feof(contratados));
```

2. Composição

É extremamente comum que um bloco seja interno a outro, nessas situações acrescenta-se uma identação para cada agrupamento ao qual a instrução pertence.

```
for (int i = 0; i < linhas; i++) {
    for (int j = 0; j < colunas; j++) {
        if (matriz[i][j] != 0) {
            // Código indentado 3 vezes: for(i) + for(j) + if()
            produto *= matriz[i][j];
        }
    }

    // Código indentado 1 vez: for(i)
    printf("Produto %i: %i\n", i, produto);
    produto = 1;
}
```

3. Switch case

Por utilizar o conceito de labels, o `switch()` é um caso particular onde qualquer uma das seguintes convenções é aceita.

```
// Sem indentação na especificação dos cases
switch(operacao) {
case ADD:
    resultado = operandos[0] + operandos[1];
    break;

case SUB:
    resultado = operandos[0] - operandos[1];
    break;
}
```

```
// Com indentação na especificação dos cases
switch(operacao) {
    case ADD:
        resultado = operandos[0] + operandos[1];
        break;

    case SUB:
        resultado = operandos[0] - operandos[1];
        break;
}
```

Note, porém, que independente da opção adotada, as instruções internas aos cases são sempre indentadas para separação dos blocos reais condicionais.

Comentários

Idealmente, um código tem bons nomes de variáveis e funções e é legível ao ponto de não necessitar de qualquer comentário. No entanto, com algoritmos complicados ou cálculos otimizados por operadores bitwise por exemplo, isso nem sempre é possível e pra isso existem os comentários.

1. Comentários do programa

Procure comentar blocos ao invés de instruções, lugares comuns para comentários são acima de blocos lógicos e funções especificando seu papel,

parâmetros e retorno. Somente quando uma linha é muito complexa que ela recebe uma descrição exclusiva. Nessas situações recomenda-se o uso de `//` com um espaço antes do texto.

```
// Explicação do bloco
for (int i = 0; i < numero_jogadores; ++i) {
    int chave = (jogadores[i]->ataque >> shift) & 0xff; // Explicação da linha
    contador[chave]++;

    copia[i] = jogadores[i];
}
```

2. Código não usado

Por vezes, durante o desenvolvimento, é útil deixar parcelas do arquivo comentadas com `/* */` para que sejam facilmente colocadas e removidas sem a necessidade de reescrita, porém, é de extrema importância que tais seções **não estejam presentes** na versão final entregue, pois dificultam a correção.

> 💡 **DICA:** Existem formas mais apropriadas de se executar códigos somente quando requisitado, pesquise sobre a flag `--DDEBUG` do `gcc` e as macros `#ifdef` e `#ifndef`.

3. Cabeçalho

Uma prática comum na escrita de software é colocar um comentário no topo do arquivo explicitando sua funcionalidade, seus autores, data de modificação dentre outros.

```
/**
 * Exemplo de cabeçalho
 *
 * Nome: Aluno Exemplo
 * Número USP: 123
 */
```

Operadores

Uma grande parcela de qualquer programa é composta por operações sobre dados e, portanto, é de extrema importância que essas seções sejam tão claras para o leitor e programador quanto for possível.

1. Espaços

Coloque espaços em branco entre os operadores aritméticos, lógicos ou de comparação e seus respectivos operandos de forma a agilizar a identicação de cada uma das componentes.

```
// Com espaços
while (i - 1 < tamanho && string[i - 1] != '\0')
```

```
// Sem espaços
while (i-1<tamanho&&string[i-1]!='\0')
```

Nos casos onde excuta-se sobre somente um operando, como com `~`, `!`, `++`, `--`, `*` e `&`, é preferível unir o simbolo representante da ação ao valor no qual será aplicado. O mesmo vale para qualquer tipo de casting.

```
if (!isalpha('A' + (char)codigo))
```

Recomenda-se, também, colocar um espaço entre as palavras reservadas `if`, `for` e `while` e seus `()` para tornar a leitura mais natural, isso não se aplica, porém, a chamada de funções pois nesses casos é preciso deixar bem claro de quem são os parâmetros especificados.

```
// Com espaço entre o for e ()
for (int i = 0; i < numero_registros; i++) {
    // Sem espaço entre a função e ()
    le_registro(&registros[i]);
}
```

2. Simplificação de sintaxe

A linguagem C, como diversas outras, oferece algumas sintaxes simplicadas para casos de uso específicos, exemplificados pelos operadores de acesso `[]` e `->`, de incremento `++` e `--`, de atribuição `+=`, `/=`, `&=` dentre outros. O uso desses artefatos é incentivado por reduzirem a poluição visual e serem facimente interpretados por um programador.

```
// Com simplificadores
monstro->defesa *= monstro->escudo + BONUS_TURNO;
```

```
// Sem simplificadores
(*monstro).defesa = (*monstro).defesa * ((*monstro).escudo + BONUS_TURNO);
```

3. Parênteses

Assim como na matemática, os parênteses podem ser utilizados para alterar a precedência das operações executadas e, ainda seguindo o exemplo dos matemáticos, os desenvolvedores somente os explicitam caso a ordem desejada não seja a padrão ou esta não seja suficientemente clara a primeira vista.

```
// Parênteses não obrigatórios
if ((sexo == 'F' && imc > 39) || (sexo == 'M' && imc > 43))
```

Constantes

Definir constantes em uma aplicação é um fator que aumenta muito a legibilidade do código fonte em questão, além de facilitar alterações, tendo que mudar o valor em um único local diminuindo, também, as chances de causar um erro na edição.

1. Utilizando constantes

Por convensão, os nomes de constantes seguem o padrão **UPPER_CASE**. Deve-se criar constantes quando um valor numérico tem um significado, por exemplo é um limite de velocidade ou uma constante da matemática ou física.

```
// Constante de fácil alteração
const int LIMITE_VELOCIDADE = 80;
```

```
if (velocidade > LIMITE_VELOCIDADE) {
    printf("Excedeu o limite de %i km/h em %i km/h", LIMITE_VELOCIDADE,
           velocidade - LIMITE_VELOCIDADE);
}
```

Diversas bibliotecas oferecem algumas constantes bastantes úteis como é o caso da `limits.h` que disponibiliza os mínimos e máximos suportados por cada tipo.

2. Declarar const ou #define

Existem duas formas de se criar constantes simples em C, usar a palavra reservada `const`, ou a diretiva de compilador `#define`. De forma geral o uso do `const` é mais recomendado, mas existem diferenças e elas devem ser consideradas no momento de escolher qual usar.

Diferenças entre const e #define

Declaração	Tempo de	Escopo	Tamanho	Tipagem
const	execução	local	do tipo	☑
#define	compilação	global	0	☐

3. Utilizando enum

Diferente dos anteriores, o `enum` é utilizando para declarar conjuntos de constantes, sendo extretamente recomendado por facilitar a leitura e a escrita. Isso porque, caso não seja especificado o valor de uma das constantes, ela terá um valor padrão equivalente ao `anterior + 1` ou `0` caso seja a primeira.

```
// Constantes com valores padrão de 0 até 6
enum Dias {
    DOMINGO,
    SEGUNDA_FEIRA,
    TERCA_FEIRA,
    QUARTA_FEIRA,
    QUINTA_FEIRA,
    SEXTA_FEIRA,
    SABADO
};
```

Assim como no caso do `const`, o `enum` tem checagem de tipos e obedece os padrões de escopo. É possivél, ainda, utilizar o `enum` criado como um tipo especial, tendo como valores possíveis somente o que foi declarado em seu interior.

```
enum Dias dia = SEGUNDA_FEIRA;
```

Variáveis

Variáveis, unidas a funções, são provavelmente os dois conceitos mais fundamentais da programação e, sendo assim elas merecem um tratamento especial, tendo nomes claros e expressivos, tipagem adequada e escopo correto.

1. Nomes

Por convensão, utiliza-se o **snake_case** ou o **camelCase** para variáveis, sendo o **snake_case** o mais comum em C, o importante é escolher um e manter o código consistente. Nunca use nomes de uma só letra e evite, inclusive, abreviações. A ideia é que quanto mais claro o nome melhor. Abreviações paracem uma boa ideia de início, mas elas simplesmente acrescentam uma barreira ao verdadeiro nome do dado.

```
// Sem cotexto
char *n, *s;
```

```
// Com contexto
char *nome, *sobrenome;
```

⚠ **ATENÇÃO:** Diversas vezes nas propostas dos exercícios, as variáveis são dadas com nomes como *n* ou *p*. Isso é feito para agilizar a leitura e se aproximar dos padrões matemáticos, porém os nomes internos ao código **não precisam** e **não devem** ser idênticos aos fornecidos nesses casos.

Faça com que toda e qualquer variável seja legível sem a necessidade de interpretar uma abreviação ou sigla imventada. É muito mais fácil entender *data_e_hora* do que decifrar que *amdhms* significa ano, mês, dia, hora, minuto e segundo.

```
// Nome não legível
char *amdhms;
```

```
// Nome legível
char *data_e_hora;
```

Uma boa dica é analisar se a variável necessita de um comentário para que seja completamente entendida, pois, caso sim, há um forte indício de que seu atual nome é ruim e deve ser alterado.

2. Singular ou plural

Dê nomes no singular para qualquer variável que carregue somente uma informação e no plural em casos de variáveis compostas como vetores. O plural pode ser, por vezes, utilizado para variáveis contadoras, porém a preferência é sempre para variáveis com múltiplas informações.

```
// Ideia incorreta de ponteiro com uma linha
int linhas;
char *linha;
```

```
// Ideia de ponteiro com múltiplas linhas
int numero_linhas;
char *linhas;
```

Evite numerar as variáveis, procure um nome mais específico para cada uma e, caso não haja, declare um vetor com o número de posições necessárias.

```
// Variáveis redundantes
int divisor1, divisor2;
```

```
// Variável única
int divisores[2];
```

3. Declaração

Os mais puristas aconselham a declarar somente uma variável por linha, apesar de não ser necessário chegar nesse extremo é bom ter cautela para que linhas de declarações não fiquem grandes e confusas. É recomendável, porém, que a declaração de ponteiros seja separada das variáveis comuns para evitar confusão.

```
// Parece que delimitador é um ponteiro
char *palavra, delimitador;
```

```
// Diferenciação fácil entre ponteiro e caractere
char *palavra;
char delimitador;
```

Declare e inicialize variáveis a partir do momento em que elas são necessárias, isso ajuda a evitar erros de acesso antes do momento adequado e possibilita a leitura do código sem a necessidade de ficar subindo para checar se há um valor inicializado naquele ponto.

4. Booleanos

Em C, os valores booleanos são baseados em números, 0 implica falso e qualquer outro implica verdadeiro. Porém, para melhorar a legibilidade na utilização de flags, por exemplo, pode-se definir valores lógicos utilizando a biblioteca `stdbool.h` que oferece o tipo `bool` e as macros `true` e `false`.

5. Declarar float ou double

A diferença entre esses dois tipos básicos em C são tamanho e precisão. Pela maior capacidade de memória dos computadores atuais, recomenda-se usar `double` para que os resultados sejam mais próximos da realidade, já que o custo de memória é normalmente desprezível.

6. Escopo

Assim como deve-se criar uma variável somente a partir do momento que ela é útil, é importante encapsulá-la no escopo mais curto possível. Por exemplo, uma variável auxiliar de leitura ou uma conta matemática dentro de um laço deve existir somente dentro dele.

```
while (!feof(stdin)) {
    conta_bancaria_t conta;
    le_conta_bancaria(&conta);
    escreve_conta_bancaria(conta, contas);
}
```

Note que esse tipo de declaração não implica que o programa irá destruir e criar uma nova variável a cada iteração do laço, mas criará uma que existirá somente durante a execução dele. Evite ao máximo declarar variáveis globais pois elas tornam o código difícil de acompanhar e podem, portanto, provocar erros no desenvolvimento.

7. Loops

Em loops, principalmente do tipo `for()`, a regra de nomeação pode ser quebrada, isso porque existe uma convenção para a utilização de variáveis `i`, `j` e `k` para contagem nessas situações, sendo, inclusive, o mais aconselhado por simplificar a linha do loop.

```
// Linha complexa
for (int contador_cartas = 0; contador_cartas < numero_cartas; contador_cartas++)
```

```
// Linha simples
for (int i = 0; i < numero_cartas; i++)
```

Note que a variável é delarada dentro do `for()` para que a regra do escopo seja mantida. A troca para a próxima letra só ocorre caso exista um loop interno a outro, utilizando o `i` para o mais externo, depois o `j` e por fim o `k`.

```
// Letra i no for() externo
for (int i = 0; i < matriz.linhas; i++) {
    // Letra j no for() interno
```

```
    for (int j = 0; j < matriz.colunas; j++) {
        printf("%i\t", matriz.dados[i][j]);
    }

    printf("\n");
}

// Letra i no for() externo
for (int i = 0; i < matriz.linhas; i++) {
    free(matriz.dados[i]);
}
free(matriz.dados);
```

> 💡 **DICA:** Nunca utilize a letra `l` ('L' minúscula) como variável de contagem, isso porque em diversas fontes ela é facilmente confundida com o número 1. Não é comum ter quatro loops compostos e se for o caso é valido repensar a lógica.

Estruturas

Em diversas situações, dados tem relações entre si e podem ser agrupados como parcelas de um objeto maior. Em C isso pode ser feito com o auxílio de uma `struct`.

1. Utilizando estruturas

Uma convenção de linguagens orientadas a objetos que pode ser adotada é a nomeação de estruturas com o **PascalCase**. Suas componentes internas, porém, são nomeadas como variáveis comuns e não necessitam de informações reduntantes, ou seja, caso seja definida um estrutura *Pessoa*, uma componente *nome* é mais recomendada que *nome_pessoa*.

```
struct Produto {
    char *nome;
    char *codigo_barras;
    int preco;
    int quantidade;
};
```

2. Composição

É possível, também, colocar uma estrutura interna à outra, o que em programação orientada a objetos é conhecido como composição. Isso pode melhorar a organização do código e evitar desperdícios quando somente a estrutura menor precisa ser usada. Nesses casos, recomenda-se declarar as estruturas separadamente, começando pela mais interna.

```
// Estrutura simples
struct Musica {
    char *autor;
    char *titulo;
    char *data_lancamento;
    long long int duracao;
};

// Estrutura composta
struct Album {
    char *titulo;
    int numero_musicas;
    struct Musica *musicas;
};
```

3. Tamanho e ordem

É preciso ter cuidado ao se utilizar o operador `sizeof()` em uma `struct`, pois devido a otimizações do compilador é provável que o tamanho não corresponda a soma dos tamanhos das componentes. Além disso não é garantido que os dados sejam armazenados de forma sequencial.

```
// Escrita incorreta
fwrite(&aluno, sizeof(struct Aluno), 1, matriculados);
```

```
// Escrita correta
fwrite(&aluno.nome, sizeof(char), strlen(aluno.nome), matriculados);
fwrite(&aluno.numero, sizeof(int), 1, matriculados);
```

Tipos Definidos

O `typedef` em C é utilizando quando quer-se encuntar o nome de um tipo, defindo-o como um novo tipo válido. Isso pode ser aplicado para variações de tipos básicos como um `unsigned long long int`, mas sem dúvidas seu uso mais comum é com `struct`. Um padrão adotado para os nomes de tipos definidos é deixar tudo minúsculo e acrescentar um **_t** no final.

```
typedef struct Matriz {
    int linhas;
    int colunas;
    double **dados;
}
matriz_t;
```

Note que apesar de não ser obrigatório, o nome *Matriz* da `struct` foi mantido, pois ajuda editores e IDEs no reconhecimento do tipo que está sendo utilizado.

Funções

A última peça fundamental para a construção de qualquer aplicação são as funções e, tendo tamanha importância, todo programador deve ser capaz de as construir bem, dando-lhes bons nomes, parâmetros razoáveis e uma funcionalidade clara.

1. Nomes

Por convensão, utiliza-se o **snake_case** ou o **camelCase** para funções, diferenciando-as das variáveis com nomes que sejam verbos ou ações. Assim como nos dados, os nomes aqui devem ser capazes de transmitir todo o intuito da função e o nome de seus parâmetros pode auxiliar nessa tarefa.

```
// Nome e parâmetros confusos
void maiusc(const char *o, char *d)
```

```
// Nome e parâmetros claros
void string_para_maiusculo(const char *origem, char *destino)
```

> 💡 **DICA:** Pode-se adotar um padrão próprio para diferenciar variáveis e funções, por exemplo usando **snake_case** para variáveis e **camelCase** para funções, porém não é necessário.

2. Parâmetros

Recomenda-se que uma função tenha no máximo **3 ou 4 parâmetros**, sendo esse um limite baseado na facilidade para decorar quais e qual a ordem dados que devem ser passados. É possível reduzir o número de parâmetros retirando os que podem ser deduzidos a partir de outros, utilizando `struct` ou quebrando a função em parcelas menores.

```
// Muitos parâmetros
void move_peca(char movimento, int x, int y, char **tabuleiro, int linhas, int colunas)
```

```
// Poucos parâmetros
void move_peca(char movimento, coordenadas_t posicao, tabuleiro_t tabuleiro)
```

3. Passagem por referência

Passe parâmetros por referência somente se for necessário alterá-los internamente, pois essa forma pode acabar consumindo mais memória e processamento, além de poluir visualmente o código se aplicada em situações incorretas.

```
// Passagem por referência desnecessária
void imprime_vetor(int **vetor, int *tamanho) {
    for (int i = 0; i < *tamanho; i++) {
        printf("%i\t", (*vetor)[i]);
    }
    printf("\n");
}
```

```
// Passagem por referência necessária
void troca_valor(int *primeiro, int *segundo) {
```

```
    int temporario = *primeiro;
    *primeiro = *segundo;
    *segundo = temporario;
}
```

Em casos mais simples, prefira utilizar o retorno ao invés da passagem por referência. Esse é um fluxo de dados mais natural que alterar o valor de um parâmetro e permite ao usuário da função chamá-la sem a preocupação de perder os dados originais.

```
// Alterando o parâmetro
void aumenta_salario(int *salario, double aumento) {
    *salario = (int)(*salario * aumento);
}
```

```
// Retornando o valor
int aumenta_salario(int salario, double aumento) {
    return (int)(salario * aumento);
}
```

4. Funcionalidades

É dito que uma função deve ter **uma e somente uma tarefa**. Um bom paralelo é que função é um bloco lógico transformado em uma rotina que pode ser chamada várias vezes. É difícil definir, porém, o que é uma funcionalidade e existem, ainda, funções cujo papel é controlar a chamada de outras. Por isso a dica principal é manter as funções curtas e, caso seja possível quebrá-la em pedaços menores o faça.

```
// Múltiplas tarefas na função
void multiplica_e_imprime_matriz(matriz_t primeira, matriz_t segunda) {
    matriz_t produto = cria_matriz(primeira.linhas, segunda.colunas);

    for (int i = 0; i < produto.linhas; i++) {
        for (int j = 0; j < produto.colunas; j++) {
            produto.dados[i][j] = 0;
            for (int k = 0; k < primeira.colunas; k++) {
                produto.dados[i][j] += primeira.dados[i][k] * segunda.dados[k][j];
            }
        }
    }
```

```
    for (int i = 0; i < produto.linhas; i++) {
        for (int j = 0; j < produto.colunas; j++) {
            printf("%lf\t", produto.dados[i][j]);
        }
        printf("\n");
    }

    libera_matriz(produto);
}
```

```
// Uma tarefa na função
matriz_t multiplica_matriz(matriz_t primeira, matriz_t segunda) {
    matriz_t produto = cria_matriz(primeira.linhas, segunda.colunas);

    for (int i = 0; i < produto.linhas; i++) {
        for (int j = 0; j < produto.colunas; j++) {
            produto.dados[i][j] = 0;
            for (int k = 0; k < primeira.colunas; k++) {
                produto.dados[i][j] += primeira.dados[i][k] * segunda.dados[k][j];
            }
        }
    }

    return produto;
}

// Uma tarefa na função
void imprime_matriz(matriz_t matriz) {
    for (int i = 0; i < matriz.linhas; i++) {
        for (int j = 0; j < matriz.colunas; j++) {
            printf("%lf\t", matriz.dados[i][j]);
        }
        printf("\n");
    }
}
```

Uma boa dica é analisar se para descrever as ações da função é necessário utilizar um *e*, pois, caso sim, há um forte indício de que sua função pode ser dividida em partes.

5. Declaração

Como a leitura de um programa começa pela `main()`, é recomendável que acima dela esteja somente aquilo que é necessário, ou seja, é preferível que acima da função principal sejam colocados somente os cabeçalhos das funções que ela

chama e, abaixo dela, fiquem as verdadeiras declaração das outras funções utilizadas.

6. Retorno da main

Em C, o programador tem a liberdade de escolher o retorno que sua função `main()` terá, no entanto, como esse é um valor normalmente direciado ao sistema operacional, existem alguns padrões que devem ser seguidos. O retorno deve ser um inteiro, onde 0 é utilizado para sinalizar que não houveram falhas enquanto 1 indica que algo de errado ocorreu. A bilioteca `stdlib.h` oferece as constantes `EXIT_SUCCESS` e `EXIT_FAILURE` para tornar mais legíveis esses retornos.

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Numero de parametros incorretos");
        exit(EXIT_FAILURE);
    }

    // ...

    return EXIT_SUCCESS;
}
```

📙 O run.codes pode ter comportamentos inconvenientes quando o retorno da `main()` é 1 e, portanto, em códigos para envio recomenda-se sempre retornar 0 ou `EXIT_SUCCESS` em qualquer ocasião.