Boas práticas de programação em C
Um estudo através de exemplos

Luiz Albini, Luis de Bona, Marcos Castilho, André Grégio

Departamento de Informática/UFPR
Março de 2022

Resumo

Neste texto comentamos sobre as boas práticas de programação a partir de programas reais submetidos por estudantes do segundo período dos cursos de Ciência da Computação e de Informática Biomédica da UFPR. Apresentaremos o problema e diversas soluções comentadas, visando aperfeiçoar a redação de bons programas, em particular na linguagem C.

Introdução

Comentar sobre boas práticas é sempre polêmico devido aos diferentes gostos e preferências de cada programador. Contudo existem aspectos universalmente aceitos e as divergências quase sempre se devem a pequenos detalhes.

Nós autores decidimos em comum acordo quais são as boas práticas que devem ser seguidas ao longo da disciplina CI1001 – Programação I ofertada para os cursos de Ciência da Computação e Informática Biomédica da UFPR – Universidade Federal do Paraná.

Esta disciplina tem como pré-requisito a disciplina de Algoritmos e Estruturas de Dados I, ofertada para calouros. Em Algoritmos I os e as estudantes aprenderam os princípios básicos da construção de algoritmos e aprenderam a programar o básico na linguagem `Pascal`. Agora devem aprender a programar em `C` para o bom aprendizado de conceitos de Tipos Abstratos de Dados, particularmente na implementação de listas ligadas com alocação dinâmica.

O texto é escrito a partir de programas reais feitos por estudantes desta disciplina. Manteremos anônimos os autores destes programas e avisamos que as críticas não são pessoais, são sobre os códigos feitos e a motivação é puramente didático-pedagógica.

Nós acreditamos que comentar sobre boas práticas a partir de programas reais pode ser mais produtivo para o leitor pois o código em estudo torna explícito os motivos dos comentários sobre uma boa prática não ter sido seguida.

Os programas recebidos procuraram resolver o seguinte problema:

> Fazer um programa em `C` que leia um número inteiro positivo n do teclado e em seguida leia dois vetores de números inteiros quaisquer e imprima `sim` caso os vetores sejam iguais e imprima `nao` em caso contrário.

Trata-se de um problema simples para quem já fez Algoritmos 1 em `Pascal`, o desafio era somente escrever algo já conhecido na nova linguagem.

Foram recebidas 84 soluções, nós escolhemos 18 delas para comentar ou para deixar como exercícios de fixação. Cada programa escolhido tem uma motivação diferente, mas todos eles servirão para ilustrar um ou mais problemas encontrados no código, todos eles têm algum "defeito" com relação a alguma boa prática não seguida e servirá para poderemos explicar a situação e apontar diferentes maneiras de escrever o código de modo que eles fiquem mais bem escrito, afinal sabemos que quem lê um programa é um ser humano.

Este texto será focado nas boas práticas, embora alguns comentários sobre os algoritmos escolhidos serão feitos em algum momento.

A preferência para compilação naquele semestre foi pelo uso das opções `-Wall` e `-std=c90` passadas ao compilador `gcc`. O uso do `-Wall` é para o compilador ser "chato" e "reclamão" ao máximo, pois isso é bom para aprendizes da linguagem `C`.

O motivo do uso do `-std=c90` foi para estabelecermos um padrão mínimo de uso. Isto provoca algumas limitações, o que é bom para aprendizes, força o estudante a aprender técnicas básicas, entre outras coisas, e também limita os estilos possíveis. O resultado é que, coletivamente, se formam programadores melhores.

Qualquer outro padrão poderia ter sido escolhido, a opção por um padrão mais antigo é devido a esta disciplina, de Programação I, ser ofertada para estudantes do segundo período que sabem programar em `Pascal`, mas estão tendo suas primeiras aventuras com a linguagem `C`.

1 Um programa sem funções

```
1  #include <stdio.h>
2
3  int main ()
4  {
5      int n, cont;
6
7      int igual = 1;
8
9      int a[200], b[200];
10
11     scanf("%d", &n);
12
13     for (cont=1 ; cont<=n ; cont=cont+1)
14     {
15         scanf("%d", &a[cont]);
16     }
17
18     for (cont=1 ; cont<=n ; cont=cont+1)
19     {
20         scanf("%d", &b[cont]);
21     }
22
23     for (cont=1 ; cont<=n ; cont=cont+1)
24     {
25         if (a[cont]!=b[cont])
26             igual=0;
27     }
28
29     if (igual==1)
30         printf("sim\n");
31     else
32         printf("nao\n");
33 }
```

Figura 1: Primeira solução.

Como o título da seção sugere, o principal problema deste programa é a falta de funções no código. Definitivamente programar sem usar funções é um problema, mesmo para programas pequenos como esse.

Observem que as linhas 13–16 são basicamente as mesmas das linhas 18–21, a menos do nome das variáveis. O código está claramente duplicado e isso é certamente uma péssima prática.

Mesmo a parte mais crítica da solução, que é o pedaço que efetivamente compara os dois vetores, poderia ser uma função, pois torna o código da função `main` mais legível e bonito.

Mas este não é o único defeito desta solução, a seguir apontamos outras e ao final da seção mostraremos o código refatorado melhorado.

Uso de constantes: Este programador usou o número 200 fixo (*hardcoded*) no programa. Deveria ter usado um *define*;

Uso desnecessário de chaves: Praticamente nenhuma das chaves é necessária neste programa, a menos das chaves que definem o corpo da função `main`. Isto torna o código mais difícil de ler, pois quem vê um par de chaves espera mais de um comando entre elas;

Identificadores das variáveis: Normalmente vetores usam as letras `v` e `w`, é raro ver programas com identificadores de vetores sendo as letras `a` e `b`. Menos importante é o uso do identificador `cont` para índice de vetor, normalmente se usa `i` ou `j`;

Comentários: Este programa não tem nenhum comentário, isto não é boa prática em praticamente nenhum programa;

Incremento de um: Também menos importante, mas é raro ver um programador `C` usar a construção `cont= cont + 1`, normalmente o que se vê é `cont++`;

Falta de padrão: Embora seja um programa pequeno, este código não tem um padrão. De fato, se nota que foram utilizados espaços neste comando: `int igual = 1` (linha 7), mas em outras construções ficou tudo colado, como nestes exemplos: `if (igual==1)` (linha 29). A mesma coisa para o espaçamento utilizado antes dos parênteses na função `main`, nos comandos `for`, mas não nos comandos `scanf` e `printf`, que ficaram com os parênteses colados no comando;

Vetor inicia em zero: Este ponto também não é de tanta importância, mas programadores em `C` costumam iniciar os índices do vetor sempre em zero, o que faria com que a comparação do limite do vetor fosse `cont < n` e não `cont <= n` como está;

Falta do retorno: Este caso já é um problema que inclusive gera um alerta (*warning*) do compilador (veja abaixo), pois a função `main` deve retornar um `int` e não retorna nada.

```
$ gcc -Wall -std=c90 -o exemplo1_v0 exemplo1_v0.c
exemplo1_v0.c: In function ‘main’:
exemplo1_v0.c:33:1: warning: control reaches end of non-void function [-Wreturn-type]
   33 | }
      | ^
```

O programa mostrado na figura 2, embora ainda possa ser melhorado, servirá como um gabarito para fins de comparação com as outras soluções discutidas no restante do texto.

```c
#include <stdio.h>
#define MAXTAM 200

/* Le um vetor (sem se preocupar com o tamanho maximo) */
void ler_vetor (int v[], int tam)
{
    int i;

    for (i=0; i < tam; i++)
        scanf("%d", &v[i]);
}

/* Retorna 1 se os vetores sao iguais e 0 caso contrario */
int sao_iguais (int v[], int w[], int tam)
{
    int i;

    for (i=0; i < tam; i++)
        if (v[i] != w[i])
            return 0;
    return 1;
}

int main ()
{
    int n;
    int v[MAXTAM], w[MAXTAM];

    scanf("%d", &n); /* poderia testar se n < MAXTAM - 1 */
    ler_vetor (v, n);
    ler_vetor (w, n);

    if (sao_iguais (v, w, n))
        printf("sim\n");
    else
        printf("nao\n");

    return 0;
}
```

Figura 2: Um possível programa gabarito.

2 Um código repleto de problemas

O próximo código, de um outro estudante, permite observar com mais foco os problemas que a falta de padrões pode causar. Como o programa dele é um pouco grande, vamos analisá-lo em várias partes, iniciando pela função `main`.

```
1  int main(void){
2      printf("insira um numero n: "); // aqui pede ao usuario um numero n para o tam
           do vetor
3      int n;
4      scanf("%d",&n); //recebe o n
5
6      printf("\n");
7      //declaracao dos vetores
8      int v[n];
9      int w[n];
10
11     //criacao dos vetores
12     cria_vetor(v,n);
13     printf("agora o proximo vetor \n");
14     cria_vetor(w,n);
15
16     //imprime os vetores
17     imprime_vetor(v,n);
18     printf("\n");
19     imprime_vetor(w,n);
20     printf("\n");
21
22     // verifica se os vetores sao iguais e imprime na tela o resultado da
           verificacao
23     int resultado;
24     resultado = verifica_igualdade(v,w,n);
25     if (resultado==0){
26         printf("Sim");
27     }
28     else if (resultado == 1) {
29         printf("Nao");
30     }
31
32
33 }
```

Figura 3: Segunda solução, função main.

Este programa merece ser estudado em forma de subseções, que seguem abaixo.

2.1 A aparente complexidade

Impossível não comparar com o programa da figura 2, cuja função `main` tem apenas 16 linhas, enquanto este em estudo tem 33, das quais duas são linhas em branco estranhas ao final do código.

Como este código se tornou tão complexo?

A primeira linha já tem dois problemas: o primeiro é que a linha é muito comprida e o texto quebra na linha de baixo; o segundo problema é o comentário inútil. Este trecho de código é suficientemente básico e óbvio para que qualquer programador entenda que se está solicitando a leitura de um número ao usuário. O comentário é redundante.

Em termos de programas feitos para a disciplina, na qual seguimos as ideias das competições de programação, em que os programas neste momento servem para compreender conceitos mais importantes e também considerando que os programas terão suas correções feitas semi-automaticamente, o comando `printf` da linha 2 poderia ter sido removido.

Mais abaixo, entre as linhas 16–20, encontramos a impressão dos vetores lidos, porém o enunciado não pediu para imprimir os vetores. Pediu somente para imprimir sim ou não.

Na linha 22, outro comentário que poderia ser mais curto ou inexistente, o nome da função por si deveria deixar claro que os vetores estão sendo comparados para saber se são iguais, com foi feito na figura 2 na linha 33.

O último problema desta seção é o uso da variável `resultado`, que não precisaria existir. A função pode ser usada no corpo do comando `if`, sem necessidade de variáveis adicionais. Neste caso pelo menos é óbvio este aspecto.

2.2 O não uso do padrão C90

Este programa não está no padrão C90. Conforme explicamos no início do texto, este foi o padrão adotado na disciplina naquele semestre, o ou a estudante não respeitou isso.

O fato de não ter respeitado tem duas implicações maiores: a primeira é grave, que é um descumprimento de uma regra. Se fosse em uma empresa isso poderia custar uma bela advertência. Para um programador, é como descumprir uma lei.

A segunda consequência é menos grave, mas atrapalha a correção semi-automática que é feita pelos professores. Se fosse uma competição de programação, tais como as da ACM, SBC ou OBI, a equipe teria levado uma penalidade por `compilation error`.

O padrão foi desreipeitado quando se usa comentários com `//` e não o padrão C90, que é com os delimitadores `/*` e `*/`.

O padrão também foi violado ao se declarar variáveis "no meio" do código, tal como nas linhas 3, 8, 9 e 23. No padrão C90, deve-se declarar todas as variáveis no início da função.

O autor que escreve esta subseção na verdade prefere neste padrão C90, ninguém ainda o convenceu que declarar variáveis no "meio" do código tem alguma utilidade nesta fase do aprendizado de um aluno ou aluna.

2.3 Vetores de tamanho variável

Nas linhas 8 e 9 foram declarados dois vetores de tamanho `n`, que por sua vez foi lido do teclado na linha 4. Isto caracteriza um *Vetor de tamanho variável*, ou no inglês VLA – *Variable Lenght Array*. Este conceito existe na linguagem `C`, até onde sabemos, desde o padrão ANSI C, que é bastante antigo.

Nós não recomendamos seu uso, basicamente por dois motivos:

- A alocação de memória é feita na *stack* e não na *heap*, o que pode ser um problema se o usuário digitar um valor muito grande para `n`, dado que o espaço na *stack* normalmente não é muito grande.
  - O tamanho da*stack* depende do sistema operacional. Atualmente o padrão costuma ser 8Mb. Se, ao utilizar VLA, não houver espaço para alocação do vetor de tamanho variável, o programa vai executar com erro. Além disso, o uso de VLA torna seu código vulnerável a ataques de *stack smashing*, que serão vistos futuramente na disciplina de Segurança Computacional.
- Se o programador quer usar vetores de tamanho variável, tem a sua disposição a alocação dinâmica através do uso do `malloc`, que por sinal aloca espaço na *heap*.
- O Sistema Operacional Linux, utilizado em nosso curso, não possui VLA em seu kernel desde a versão 4.20 (2018), devido aos problemas que esse tipo de construção pode causar (`https://lkml.org/lkml/2018/3/7/621`).
- A versão C11 da linguagem tornou VLAs opcionais, a fim de que essa característica não seja utilizada por padrão.

- O MSVC (compilador de C/C++ da Microsoft), embora suporte o padrão C11, **não suporta)** suas características opcionais, como VLAs, tanto pelos problemas de segurança acima citados quanto por questões de eficiência (`https://devblogs.microsoft.com/cppblog/c11-and-c17-standard-support-arriving-in-msvc/`).

2.4 Demais problemas

Finalmente, este programa não tem o retorno, conforme já apontamos na seção anterior, o que além de gerar um *warning* pelo compilador, é inconsistente com a declaração da função `main` que deve devolver um `int`.

Novamente apontamos que, em `C`, os comandos `if` e `while`, testam um comando que resulta em um *número* que se for igual a zero é considerado falso e se for qualquer outro valor é verdadeiro. Consequentemente a construção dos comandos nas linhas 24–30 é bastante esdrúxula na linguagem `C`. Vejamos os motivos.

Este programador escolheu equivocadamente, infelizmente, os valores de retorno da função `verifica_igualdade`, ele optou por retornar zero em caso de sucesso e 1 em caso de insucesso. Justamente o contrário do padrão básico da linguagem!

Isto o obrigou a usar o `== 0` e `== 1`. Se ele tivesse feito corretamente, não precisaria destes pedaços de código. Foi exatamente isso que foi feito na solução da figura 2 nas linhas 33-36.

Por último um erro de lógica, quando o programa contém um `if` absolutamente inútil na linha 28 logo após o `else`. Quando se cai no `else` é justamente porquê o resultado é diferente de zero, o que seria suficiente para ir diretamente ao comando `printf` da linha 29. Este é um erro típico de calouros que não deveria ser visto no segundo período.

Veremos nas próximas seções os códigos deste mesmo estudante para as funções.

2.5 A função cria_vetor

Vejamos o código feito para a função `cria_vetor` na figura 4.

O problema maior nem é a brincadeira feita com a falta de criatividade para o nome da variável `bah`, mas sim com os comentários na mesma linha dos comandos, o que torna o texto difícil de ler. O texto fica carregado demais e a linha quase quebra na linha de baixo.

Por outro lado, o nome da variável é algo de extrema importância em programação. A escolha deste nome é fundamental para ajudar a tornar o

```
1 void cria_vetor(int bah[], int n){     // cria vetor, bah pq eu tava sem ideia xD
2                                        //e queria ver oq acontecia c/ qualquer nome
3     int i;
4     for (i=0;i<n;i++){
5         printf("insira insira um numero para o vetor[%d]: ", i);
6         scanf("%d", &bah[i]);
7     }
8 }
```

Figura 4: Segunda solução, função cria vetor.

código legível para seres humanos. Ademais em se tratando de vetores que tipicamente recebem os nomes `v`, `w`, `vet`, e assim por diantes. Então não é questão de ter criatividade, é questão de dar o nome certo para as coisas.

Finalmente, a redação da linha 4 poderia ter alguns espaços em branco separando as coisas e também o `printf` da linha 5 poderia ser removido, conforme comentários já feitos na seção 2.1. Consequentemente também as chaves poderiam sair.

2.6 A função verifica igualdade

Vejamos o código da função `verifica_igualdade` na figura 5.

```
1  int verifica_igualdade(int v[], int w[], int n){ // verifica a igualdade entre dois
       vetores
2      int resultado_igualdade;
3      resultado_igualdade = 0; //inicia-se a comparacao dos vetores admitindo que sao
           iguais
4      int i;
5      for(i=0;i<n; i++){
6
7          if (v[i]!=w[i]){ // se encontrar numeros diferentes nas mesmas posicoes do
               vetor, termina a execucao
8              resultado_igualdade = 1; // e nos da o resultado
9              printf("A desigualdade ocorre em: \n");
10             printf("v[%d] = %d\n", i, v[i]);
11             printf("w[%d] = %d\n", i, w[i]);
12             return resultado_igualdade; // se nao tiver numeros diferentes em
                   nenhuma posicao do vetor
13         }                                   // retorna o resultado e encerra a execucao
               da funcao
14     }
15     return resultado_igualdade;
16 }
```

Figura 5: Segunda solução, função cria vetor.

Esta função é muito estranha e difícil de compreender. A lógica de programação é a primeira coisa que chama a atenção. Comparem com o código da figura 2 nas linhas 14–22, onde o código é limpo, a função faz só o que tem que fazer e mais nada.

A lógica deveria ser somente assim: vai comparando um a um e para tão logo se descubra que os vetores são diferentes. A função acima tem três comandos `printf` que não deveriam estar ai.

Não é que *poderiam estar ai*. É o caso de *não deveriam estar ai*. A função tem o nome `verifica_igualdade` então ela deve se limitar a verificar a igualdade.

Quando a igualdade for determinada, a função deve somente dar um valor de retorno indicando o que houve.

Isto deveria estar claro em um comentário logo antes do protótipo da função:

```
/* Esta funcao retorna 0 se os vetores sao diferentes
   e retorna 1 caso contrario */
```

Vejam que o comentário que ele faz é, mais uma vez, reduntante com o nome da função e também, de novo, estoura a linha, o que prejudica a legibilidade.

Por último, um típico programador `C` não usaria a variável `resultado_igualdade`, ele simplesmente faria um `return 1` ou `return 0` nos locais apropriados.

3 Uma estrutura de dados estranha

Vamos analisar mais um código com novos problemas para estudarmos, pouco a pouco vamos aprendendo as boas práticas em geral e as boas práticas em `C`.

```c
#include <stdio.h>
#include <stdbool.h>

// Tamanho maximo que a sequencia pode ter.
#define limite 200

int i, j, n, seq[2][limite];

// Verifica se a sequencia tem todos os elementos iguais.
bool areSame(int seq[2][limite]){
    for(j = 0; j<n; j++){
        if ( seq[0][j] != seq[1][j] ){
            return false;
        }
    }
    return true;
}

int main(){
// Leitura dos dados.
    do{
        printf("Digite um tamanho para a sequencia: \n");
        scanf("%d", &n);
        if(n <= 0 || n>limite)
            printf("Tamanho invalido, digite algo entre 0 e %i.", limite);
    } while (n <= 0 && n>limite);

    for (i = 0; i<2; i++){
        printf("Sequencia %i: \n", i+1 );
        for (j = 0; j<n; j++){
            scanf("%d", &seq[i][j]);
        }
    }

// Compara e returna o resultado.
    if ( areSame(seq) == true )
        printf("sim\n");
    else
        printf("nao\n");
}
```

Figura 6: Terceira solução.

3.1 Uma matriz de duas linhas?

O código da figura 6 é de um terceiro estudante que escolheu uma estrutura de dados bizarra para seu algoritmo: ao invéz de usar o básico, que consiste em declarar dois vetores, ele resolveu inventar moda e utilizou uma matriz de duas linhas, uma para o primeiro vetor, outra para o segundo vetor.

Ele provavelmente diria: mas funciona professor!

Sim, mas não está bom, torna o código confuso, complicado. A função `areSame` tem seu código basicamente com a mesma lógica do programa gabarito da figura 2. Mas observem atentamente as linhas 10 e 12, aqui reproduzidas:

```
bool areSame(int seq[2][limite]){
...
if ( seq[0][j] != seq[1][j] ){
```

É muito estranho uma função ter o nome `areSame` e receber um único argumento! Tem que prestar atenção para ver que está sendo passado uma matriz de duas linhas. Aliás, para quem vê só a função, qual o motivo de uma matriz de duas linhas?

O problema se reflete também na linha 12, que é a linha do `if`, que compara algo indexado com `[0]` com algo indexado com `[1]`. Difícil de entender o motivo de tal escolha bizarra.

3.2 Boolean em C?

Outra coisa que chama a atenção é o uso do tipo `boolean`, que não existe nativamente na linguagem `C` padrão. Foi necessário importar a biblioteca `stdbool.h` para poder usar as constantes `true` e `false` sem necessidade alguma.

Ao contrário, até complicou pois na função `main` ele teve que fazer o teste com `== true` e `== false`. Não que seja proibido usar a biblioteca, mas ela não é útil, não faz parte da filosofia da linguagem.

3.3 O programa é modular?

Definitivamente este programa não é modular. Talvez a preocupação em lidar com matrizes e booleanos tirou a atenção do programador ou programadora, que cometeu um erro grave ao não definir funções para a leitura do vetor e tampouco para a leitura do valor do tamanho dos vetores.

O trecho de código entre as linhas 21–26 deveriam estar necessariamente em uma função, nem que fosse para limpar um pouco o código sujo do programa principal.

A sujeira aqui é referente aos testes de consistência, dos limites do valor de `n` ter que estar entre 0 e 200, testes estes que não fazem o menor sentido para quem lê a função `main`.

Mais grave ainda é a leitura de dois vetores, que no nosso gabarito consiste da leitura de um vetor e a função é chamada duas vezes. Aqui, ao contrário, mesmo que o código escrito nas linhas 28–33 fosse feito em uma função, é um código horrível.

Ele tem que usar um laço duplo por causa da solução bizarra do uso da matriz no lugar do uso de dois vetores, sendo que o laço externo vai de zero a um!

Este é um típico exemplo de quem quer complicar a coisa, talvez porquê ache que sabe programar e quer "mostrar sua competência" para os professores.

Mas o pior de tudo é o uso das variáveis globais, não apenas os vetores, mas inclusive as variáveis auxiliares `i` e `j` usados nos comandos `for`. Inaceitável!

4 Função "apascalzada" que imprime

O programa da figura 7 é um típico caso de programa mal estruturado.

Observem que o `main` parece uma "tripa" e é muito feio do ponto de vista algoritmico.

Em geral, uma função que imprime deveria ser chamada de `imprime_algo` o que não é o caso pois a função tem o nome `comparaVetor`.

Se a função se chama `comparaVetor` ela deve se limitar a comparar vetores. Compara e devolve um código, por exemplo, zero para falha e um para sucesso.

Quem chamou a função é que deve coletar este código de retorno e decidir o que fazer, por exemplo, pode imprimir isso ou aquilo.

Outro ponto é o código da função que parece ter sido traduzido diretamente do `Pascal` para `C`.

Em `Pascal` uma função executa até o `end;` final, por isso a solução apresentada seria boa para esta linguagem. Mesmo assim não é, pois é ineficiente, já que testa todos os números ainda que logo de cara se descubra falha.

```
1  #include <stdio.h>
2
3  void leVetor(int n, int v[])
4  {
5      for (int i = 0; i < n; i++ )
6          scanf("%d", &v[i]);
7  }
8
9  void comparaVetor(int n, int vA[], int vB[])
10 {
11     int flag = 1;
12     for (int j = 0; j < n; j++ )
13     {
14         if (vA[j] != vB[j])
15             flag = 0;
16     }
17     if (flag == 1)
18         printf("sao iguais!\n");
19     else
20         printf("nao sao iguais!\n");
21 }
22
23 int main()
24 {
25     int n = 0;
26     scanf("%d", &n);
27     int a[n], b[n];
28     leVetor(n, a);
29     leVetor(n, b);
30     comparaVetor(n, a, b);
31     return 0;
32 }
```

Figura 7: Quarto exemplo.

Mas não é uma típica função escrita para a linguagem `C` pelo simples motivo de que em `C` existe o `return`.

Havendo return, tão logo se descubra falha, já se pode encerrar a função, por exemplo trocando-se na linha 15 de `flag = 0;` para `return 0;`.

Além do mais não tem nenhum comentário e não usa C90, pois na linha 27 vemos declarações de variáveis depois de um comando `scanf` já ter sido utilizado.

5 Outro programa "apascalzado"

O programa da figura 8 também é bem interessante de ser comentado.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #define MAX 100
4
5  int levetor (int v[], int n){
6      int i,num;
7      for(i = 0; i <= n; i++){
8          scanf("%d", &num);
9          v[i] = num;
10     }
11 }
12
13 int compara(int v1[] ,int v2[],int n){
14     int a;
15     for(int i = 0; i <= n; i++){
16         for(int j = 1; j<=n; j++){
17             if (v1[i] == v2[j])
18                 a = 1;
19             else
20                 a = 0;
21         }
22     }
23     return a;
24 }
25
26 int main(){
27     int n,resul;
28     int v1[MAX], v2[MAX];
29     printf("Digite o Tamanho do vetor:");
30     scanf("%d",&n);
31     if(n > 0){
32         printf("Digite a primeira sequencia: \n");
33         levetor(v1,n);
34         printf("Digite a segunda sequencia: \n");
35         levetor(v2,n);
36         resul = compara(v1,v2,n);
37         if (resul == 1)
38             printf("\n SIM");
39         else
40             printf("\n NAO");
41     }
42     else
43         printf("Numero invalido!");
44
45 }
```

Figura 8: Quinto exemplo.

O interesse aqui é analisar a função `main` e pelo menos por enquanto

vamos ignorar o código absurdo da função `compara` que está nas linhas 13–24. Também não vamos mais insistir que não há return neste programa.

O interesse é o código apresentado na função `main` que está nas linhas 26–43, que aliás, não precisava dos `printf` das linhas 29, 32 e 34.

É que este estudante optou por fazer testes adicionais que normalmente não são requeridos nos problemas apresentados nesta disciplina. Ele fez questão de testar se o número `n` dado como entrada era positivo, caso não seja imprime uma mensagem de erro.

Bem, já dissemos que isso não é importante nem necessário na nossa disciplina, pois adotamos os padrões das maratonas de programação, ele pelo menos poderia ter feito um programa mais com cara de linguagem `C` e não este código ainda meio "apascalzado".

Novamente aqui chamamos atenção para a existência do `return` nas funções. Isso permite fazer os testes de consistência todos no início da função. Qualquer erro já aciona um `return` que pode devolver diferentes valores para cada tipo de erro.

Mas em `C`, normalmente, não é necessário nenhum `else`, pois se cair na excessão o retorno será feito. Se não cair, não precisa do `else`. Vejamos na figura 9 a função `main` refatorada neste sentido, além de outras refatorações já comendatas anteriormente nas seções anteriores.

```
1  int main(){
2      int n;
3      int v1[MAX], v2[MAX];
4
5      scanf("%d",&n);
6      /* trata o caso particular primeiro */
7      if(n < 0){
8          printf("Numero invalido!");
9          return 1;
10     }
11
12     /*
13      * daqui para frente estamos livres dos problemas
14      * e podemos seguir com um nivel de indentacao a menos
15     */
16     levetor(v1,n);
17     levetor(v2,n);
18     if (compara(v1, v2, n))
19         printf("\n SIM");
20     else
21         printf("\n NAO");
22     return 0;
23 }
```

Figura 9: Programa refatorado.

6 Mais um que "sabe" programar

O código mostrado nas figuras 10 e 11 ilustra bem aquele estudante que "sabe" programar[1]. Ele quer mostrar que tem conhecimento de coisas que nunca foram ensinadas em sala de aula, pensa que está surfando na crista da onda, mas comete erros gravíssimos e o código dele vale nota perto de zero. Com o conhecimento adquirido até aqui já podemos indicar:

```
1  /* Programa que, dadas duas sequencias numericas de N numeros, compara ambas
       sequencias, indicando
2  se estas sao iguais ou nao */
3
4  /* ------------------------------------Bibliotecas------------------------------ */
5  #include <stdio.h>
6  #include <string.h>
7
8  /* ------------------------------------Variaveis globais------------------------ */
9  int n;
10 int vet1[100], vet2[100];
11
12 /* ------------------------------------Prototipo das funcoes-------------------- */
13 int le_tam_das_sequencias (void);
14 int le_sequencias (void);
15 int compara_sequencias (void);
16
17 /* ------------------------------------Corpo da funcao principal---------------- */
18 void main (void)
19 {
20     printf ("\nIrei verificar se 2 sequencias de N numeros sao iguais ou nao\n");
21     le_tam_das_sequencias ();
22     le_sequencias ();
23     compara_sequencias ();
24 }
```

Figura 10: Sexto exemplo, parte 1.

- Uso de variáveis globais!!! Existem raros casos onde isso se justifica.
- Funções com parâmetros `void`!!! Faz sentido em alguns casos.
- Função `main` do tipo "tripa" que não quer dizer nada, tem vários dos erros já apontados em outros programas;
- A função `main` chama várias funções sem parâmetros.
- Comentários em excesso (prejudica leitura).

[1] O código está muito grande e precisou de duas figuras. Fora o fato de termos que ter removido toda a acentuação que foi usada e só atrapalha.

```
1  /* ----------------------------------------Funcoes----------------------------------------
       */
2
3  int le_tam_das_sequencias (void)
4  {
5      printf ("\nQuantos numeros terao ambas as sequencias A e B?\n");
6      scanf ("%d", &n);
7      return (0);
8  }
9
10 int le_sequencias (void)
11 {
12     int i;
13
14     printf ("\nEscreva os %d numeros da sequencia A\n",n);
15     /* Le sequencia A */
16     for (i=0;i<n;i++)
17         scanf ("%d", &vet1[i]);
18
19     printf ("\nAgora escreva os %d numeros da sequencia B\n",n);
20     /* Le sequencia B */
21     for (i=0;i<n;i++)
22         scanf ("%d", &vet2[i]);
23
24     return (0);
25 }
26
27 /*  Verifica a igualdade de ambas as sequencias numericas indo de numero em numero.
28 Caso elas sejam diferentes printa "nao". Ja se as sequencias forem iguais printa "
       sim" */
29 int compara_sequencias (void)
30 {
31     int i;
32     char iguais[6]= "True";
33
34     for (i=0;i<n;i++)
35         if (vet1[i] != vet2[i])
36         {
37             strcpy(iguais, "False");
38             printf ("\nnao");
39             break;
40         }
41     if (strcmp (iguais,"True") == 0)
42         printf ("\nsim");
43 }
```

Figura 11: Sexto exemplo, parte 2.

Com relação às implementações das funções, podemos comentar o seguinte:

- Função `le_tam_das_sequencias` devolve um `int` mas na `main` não cai dentro de um `if` nem nada. Poderia ou ter devolvido `void` ou ter feito

algum teste na `main`;

- Idem para `le_sequencias`;
- O uso das variáveis globais o obrigou a replicar código na leitura dos vetores. Péssima escolha, desastrosa;
- A função `compara_sequencias` é uma obra prima do contraexemplo! Ela é terrivelmente complicada com relação à simplicidade do problema. Qual é a necessidade de se usar a função `strcpy`, de se usar `break`, de se usar uma string inicializada com `True`?
- E, principalmente, de uma função do tipo `int` não ter nenhum `return`?

Em suma tem tanto erro grave que nem vale a pena comentar os erros menos graves.

7 Estava indo tão bem...

O código apresentado na figura 12 ilustra que mesmo que o programa tenha vários defeitos, ele pode ter alguma coisa interessante.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int n = 0, tmp;
    int* vec;

    // le o numero de elementos do vetor
    scanf("%d", &n);

    // aloca o espaco necessario
    vec = malloc(n * sizeof(int));

    // le o vetor
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &vec[i]);
    }

    // le e ja compara os elementos do segundo vetor com os do primeiro
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &tmp);
        if (tmp != vec[i])
        {
            // se nao sao iguais termina o programa
            printf("nao\n");
            return 1;
        }
    }

    // se nao foi pego no if anterior entao os vetores sao iguais
    printf("sim\n");

    // libera o espaco alocado
    free(vec);

    return 0;
}
```

Figura 12: Sétimo exemplo.

O uso do `malloc`, seguido ao final pelo uso do `free` é uma boa prática de programação em `C`, pois faz com que o espaço do vetor seja alocado na `heap` e não na `stack`, que normalmente tem menos espaço.

O problema é que ele não usou funções e também não usou dois vetores.

A leitura da segunda sequência é feita em um laço que usa uma variável simples e que joga fora os elementos lidos. Ela faz o programa responder certo, mas não é um programa genérico o suficiente.

Normalmente, problemas como este que estamos resolvendo, o de saber se dois vetores são iguais, são na verdade um subproblema de um programa maior. Um programador deve sempre pensar em soluções genéricas e raramente em casos particulares. Mas em termos de boas práticas ele é melhor do que o programa anterior.

8 Aritmética de ponteiros

Esta versão que apresentaremos agora é tão longa que só vai caber em duas figuras separadas. Mas os comentários adicionais são praticamente os mesmos que já fizemos antes na maior parte dos casos.

Provavelmente ele ou ela fez este programa assim para aprender a usar atitmética de ponteiros. O problema é que o código ficou muito mais complicado do que deveria. O código deveria ficar muito parecido com aquele do nosso gabarito.

Alem disso:

- Uso excessivo de comentários;
- A função `sao_iguais` é muito complicada;
- Uso de booleanos;
- `igualdade == true`
- O enunciado pede sim ou não, e ele imprime mensagens mais longas;
- Tela muito cheia, carregada, com linhas estouradas.

Mas o principal, já que ele se deu ao trabalho de aprender a trabalhar com aritimética de ponteiros, é que ele declarou os vetores estaticamente no `main`, além de ter usado VLA (Vetores de tamanho variável, veja seção 2.3)...

```c
#include <stdio.h>
#include <stdbool.h>

void le_vetor(int tam, int vet[]) {
//funcao que le um vetor ja inicializado
    int *i;             //ponteiro do elemento atual do vetor
    int *fim;           //ponteiro que indica o fim do vetor
    fim = vet + tam;
    for (i=vet;i< fim;i++) {
        scanf("%i",i);
    }
}


bool sao_iguais(int vet1[], int vet2[], int tam) {
//funcao que compara dois vetores de tamanho "tam" e se forem iguais, devolve "true"
    int *pont1;     //ponteiro dos elementos do vetor 1
    int *pont2;     //ponteiro dos elementos do vetor 2
    int *fim1;      //ponteiro que indica o fim do vetor 1
    int *fim2;      //ponteiro que indica o fim do vetor 2
    bool igualdade;

    fim1=vet1+tam;
    fim2=vet2+tam;
    pont1=vet1;
    pont2=vet2;
    igualdade = true;   //comecamos considerando os 2 vetores iguais

    while (pont1 < fim1 && pont2 < fim2) {  //percorre todos os elementos dos
         vetores 1 e 2
        if (*pont1 != *pont2) {
            igualdade= false;    //se dois elementos respectivos do vetor 1 e 2
                                 //forem diferentes,"igualdade" deixa de ser verdadeiro
        }
        pont1++;    //atualiza a posicao atual no vetor1
        pont2++;    //atualiza a posicao atual no vetor2
    }
    return igualdade;   //devolve "true" se os 2 vetores sao iguais e "false" caso
         contrario
}
```

Figura 13: Oitavo exemplo, parte 1.

```
1  int main(){
2  //PROGRAMA PRINCIPAL
3
4      int tam;
5      bool igualdade;
6
7      printf("Digite o tamanho dos vetores que serao lidos: ");
8      scanf("%i",&tam);       //le o tamanho dos vetores que serao comparados
9
10     int vet1[tam], vet2[tam];
11
12     printf("Digite o primeiro vetor:\n");
13     le_vetor(tam,vet1);      //le os elementos do vetor 1
14     printf("Digite o segundo vetor:\n");
15     le_vetor(tam,vet2);      //le os elementos do vetor 2
16     igualdade= sao_iguais(vet1,vet2,tam);   //compara se os dois vetores sao iguais
17                                             //e atribui essa resposta para "
                                                    igualdade"
18     if (igualdade == true)
19         printf("Sao iguais\n");
20     else
21         printf("Nao sao iguais\n");
22     return 0;
23 }
```

Figura 14: Oitavo exemplo, parte 2.

9 Exercícios

Com base no estudado até aqui, quais críticas você faria nos programas abaixo?

9.1 Exercício 1

```c
#include <stdio.h>

//tamanho maximo dos vetores
#define MAX 10

int vetA[MAX];
int vetB[MAX];

int isEqual(int a[], int b[], int n);
void writeVector(int v[], int n);


//========PROGRAMA PRINCIPAL ========
int main(){
    int size;
    scanf("%d", &size);
    writeVector(vetA, size);
    writeVector(vetB, size);

    if ( isEqual(vetA, vetB, size) == 1)
        printf("Sim\n");
    else
        printf("Nao\n");

    return 0;
}

//========ESCREVE UM VETOR========
void writeVector(int v[], int n){
    for (int i = 0; i < n; i++){
        scanf("%d", &v[i]);
    }
}

//========COMPARA DOIS VETORES========
int isEqual(int a[], int b[], int n){
    for (int i = 0; i < n; i++){
        if (a[i] != b[i])
            return 0;
    }

    return 1;
}
```

Figura 15: Primeiro exercício.

9.2 Exercício 2

```
1  #include <stdio.h>
2
3  #define VSIZE 100 /* tamanho maximo do vetor */
4  #define TRUE 1    /* constantes para opeacao booleana */
5  #define FALSE 0
6
7  /* le vetor */
8  void ReadVet(int v[VSIZE], int size){
9      int i;
10
11     for (i = 0; i < size; i++)
12         scanf("%d", &v[i]);
13
14 }
15
16 /* compara vetor */
17 int CompareVet(int va[VSIZE], int vb[VSIZE], int size){
18     int i, eq;
19
20     eq = TRUE;
21     for (i = 0; i < size && eq == 1; i++){
22         if (va[i] != vb[i])
23             eq = FALSE;
24     }
25
26     return eq;
27 }
28
29 int main(){
30     int va[VSIZE], vb[VSIZE], n;
31
32     scanf("%d", &n);
33     ReadVet(va, n);
34     ReadVet(vb, n);
35
36     if (CompareVet(va, vb, n) == FALSE)
37         printf("nao\n");
38     else
39         printf("sim\n");
40
41
42     return 0;
43 }
```

Figura 16: Segundo exercício.

9.3 Exercício 3

```c
#include <stdio.h>

void inic_vetor(int VET[],int n){
    int k;
    for(k=0;k<=n-1;k++){
        scanf("%d",&VET[k]);
    }
}

int main (){
    puts("digite a quantidade de elementos da sequencia de inteiros:");
    int n;
    scanf("%d",&n);
    int VET1[n];
    int VET2[n];
    int i;
    printf("digite a primeira sequencia de %d inteiros\n",n);
    inic_vetor(VET1,n);
    printf("digite a segunda sequencia de %d inteiros\n",n);
    inic_vetor(VET2,n);
    for(i=0;i<=n-1;i++){
        if (VET1[i] != VET2[i]){
            puts("nao");
            return 0;
        }
    }
    printf("sim\n");
    return 0;
}
```

Figura 17: Terceiro exercício.

9.4 Exercício 4

Transforme o código "apascalzado" que está mostrado na figura 18 para algo mais parecido com `C`.

```
1  int confere_igualdade (int *v1, int *v2, int tam){
2      int i = 0;
3      int igual = 1;
4      while( i < tam && igual == 1){
5          if (v1[i]==v2[i]){
6              i = i +1;
7          }
8          else {
9              igual = 0;
10             // printf ("O numero %d eh diferente de %d \n",v1[i],v2[i]);
11             return 0;
12         }
13     }
14     return 1;
15 }
```

Figura 18: Quarto exercício.

9.5 Exercício 5

```c
#include "stdio.h"

char * compara(int vetor1[], int vetor2[], int length) {
  int i;
  for (i = 0; i < length; i++) {
    if (vetor1[i] != vetor2[i]) return "nao";
  }
  return "sim";
}

void preenche(int vetor[], int length) {
  int i;
  for (i = 0; i < length; i++) {
    scanf("%d", &vetor[i]);
  }
}

int main() {
  int v1[30], v2[30];
  int n;

  printf("\nescreva o tamanho dos vetores: ");
  scanf("%d", &n);

  printf("\nescreva os %d valores do v1: ", n);
  preenche(v1, n);

  printf("\nescreva os %d valores do v2: ", n);
  preenche(v2, n);

  printf("\n%s\n", compara(v1, v2, n));

  return 0;
}
```

Figura 19: Quinto exercício.

9.6 Exercício 6

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int inserindo(int i,int f,int *seq){ // insere a sequencia dentro de um vetor
4      if (f==i)
5          return i;
6      inserindo(i+1,f,seq);
7      scanf("%d",&seq[i]);
8
9  }
10
11 int comp(int i,int f,int *seq1,int *seq2){ // funcao recursiva  que compara dois
       valores
12
13     if (f==i)
14         return 1;  // se a funcao chegar ao fim atribuimos o valor 1 como verdadeiro
15
16     if (seq1[i]!=seq2[i])
17         return 0;  // se a funcao chegar ao fim atribuimos o valor 0 como falso
18     comp(i+1,f,seq1,seq2);
19 }
20
21 int main(){
22     int x,tam_da_seq,conferindo;
23     printf("digite n:");
24     scanf("%d /n",&tam_da_seq);
25     int sequencia1[tam_da_seq],sequencia2[tam_da_seq];
26     inserindo(0,tam_da_seq,sequencia1);
27     inserindo(0,tam_da_seq,sequencia2);
28     conferindo=comp(0,tam_da_seq,sequencia1,sequencia2);
29
30     if (comp(0,tam_da_seq,sequencia1,sequencia2)==1)
31         printf("sim");
32     else
33         printf("nao");
34
35 }
```

Figura 20: Sexto exercício.

9.7 Exercício 7

```
1  #include<stdio.h>
2
3  int leia(int a[], int n) /*lendo as sequencias numericas*/
4  {
5      int t;
6      for (t = 0; t <= n; t++)
7          scanf("%d", &a[t]);
8  }
9
10 void compara(int v[], int f[], int n) /*comparando as sequencias*/
11 {
12     int t;
13     t = 0;
14     while (v[t] == f[t] && t < n)
15         t = t + 1;
16     if (t == n)
17         printf("sim\n");
18     else
19         printf("nao\n");
20 }
21
22 int main()
23 {
24     int m, t, l;
25     int v[m];
26     int f[m];
27     printf("digite um numero inteiro positivo\n");
28     scanf("%d", &m); /*lendo um n qualquer*/
29     l = m;
30     printf("digite a primeira sequencia de %d numeros inteiros quaisquer\n", m);
31     leia(v, l-1); /*lendo a primeira sequencia*/
32     printf("digite a segunda sequencia de %d numeros inteiros quaisquer\n", m);
33     leia(f, m-1); /*lendo a segunda sequencia*/
34     compara(v, f, m); /*comparando as sequencias*/
35 }
```

Figura 21: Sétimo exercício.

9.8 Desafio

O último exercício é um desafio. O ou a estudante teve honestidade e colocou no comentário que seu programa não funciona em determinadas situações, mas funciona em outras.

Baseado nas boas práticas e em mais algum conhecimento seu, explique o que está errado. O código é apresentado em duas figuras por causa do tamanho dele.

```
1  #include <stdio.h>
2
3  /* Por alguma razao, na minha maquina, esse programa nao funciona quando n = 4 ou
       multiplos de 4 (testei com 8, 12, 16, 20.
4  Nao descobri o motivo. Num compilador online de C (https://www.onlinegdb.com/
       online_c_compiler), porem, ele funciona perfeitamente.*/
5
6  // funcao que le um vetor de tamanho n digitado pelo usuario
7  int ler_vetor(int vetor[], int n) {
8      int i;
9      for (i = 0; i < n; i++) {
10         scanf ("%d", &vetor[i]);
11     }
12     return 0;
13 }
14
15 /* funcao que verifica se todos os elementos de dois vetores sao iguais ou nao
16 retorna 0 se sao iguais, retona 1 se houver ao menos um diferente*/
17 int eh_igual(int vetor1[], int vetor2[], int n) {
18     int i;
19
20     for (i = 0; i < n; i++) {
21     if (vetor1[i] != vetor2[i]) {
22         return 1; // se o codigo encontra uma diferenca, retorna 1 e a funcao nao
               prossegue
23     } else {
24         if (i == n-1) { // se o codigo chega no ultimo valor do vetor nao encontra
               uma diferenca, a funcao retorna 0
25         return 0;
26     }
27     }
28     }
29
30 }
```

Figura 22: Oitavo exercício, parte 1.

```
1  int main()
2  {
3      int n;
4      int i;
5      scanf ("%d", &n);
6      int vetor1[n], vetor2[n];
7
8      vetor1[n] = ler_vetor(vetor1, n);
9      vetor2[n] = ler_vetor(vetor2, n);
10
11     if (eh_igual(vetor1, vetor2, n) == 0) {
12         printf("sim");
13     } else {
14         printf("nao");
15     }
16
17     return 0;
18 }
```

Figura 23: Oitavo exercício, parte 2.