16. Documentação e Organização de Arquivos

16.1 A importância de um programa legível

Em geral, a maior parte do esforço envolvido durante o tempo de vida de um programa não é gasto em sua escrita ou depuração, mas sim em sua manutenção. É muito difícil se lembrar dos detalhes de um programa escrito há seis meses atrás. Além disso, muitas vezes a modificação de um programa não é feita pelo seu autor original, mas sim por outra pessoa.

O custo de manutenção chega a ser de duas a quatro vezes maior que o custo de desenvolvimento. (Pressman)

Mesmo se estas preocupações não estão presentes quando se é um estudante, é muito importante desenvolver bons hábitos de programação desde cedo.

- Use comentários;
- Coloque nomes significativos em variáveis e constantes;
- Utilize constantes quando for possível;
- Alinhe os comandos de forma clara e coerente;
- Use espaços em branco;
- Escreva somente uma instrução por linha;
- Use parênteses em expressões para torná-las mais legíveis.
- Declare as variáveis o mais próximo possível de sua utilização, criando-se blocos para definir onde as variáveis são utilizadas;
- Utilize de forma restrita alguns comandos obscuros do C

Comentários: O melhor momento de se comentar um programa é durante sua concepção, quando seus detalhes ainda estão “frescos'” na memória do programador.

Início do programa: O início de todo programa deve ser comentado:

- o que ele faz
- método utilizado para implementar o algoritmo
- como deve ser utilizado
- formato da entrada de dados
- formato da saída dos dados
- autor
- data

Declarações de variáveis e constantes: Nas declarações, comente o uso pretendido de cada variável e constante.

```
...
int numestudantes; // Numero de registros de estudantes
double media;      // Media da nota final dos estudantes
...
```

Grupo de comandos: Coloque um comentário antes de cada comando de seleção ou repetição para explicar sua função:

```
...
// Le os valores das notas e as acumula na variavel soma
scanf (“%d”, &nota);
while(nota > 0)
{
  soma = soma + nota;
  scanf (“%d”, &nota);
}
```

```
   ...
```

Blocos e Funções: Coloque comentários antes das definições de blocos e funções para explicar o que fazem, além da descrição dos parâmetros.

```
...
// Calcula a area de uma circunferencia, sendo fornecido o raio
double areaCircunf(double raio)
{
   static double pi = 3.14159;
   areaCircunf = pi * raio * raio;
}
...
```

Comentários Relevantes: Comentários devem acrescentar alguma coisa além daquilo que pode ser facilmente aprendido:

```
   ...
   x = 2;  // x recebe o valor 2
   …
   ….
    i = i +1  // incrementa valor de i
   …
   …
  /* usando scanf, obter valor de idade e multiplicar por 365 para
obter dias */
   scanf (“%d”, &idade);
   dias = idade *365;
…..
```

Esses comentários são desnecessários e podem até atrapalhar em vez de ajudar.

Nomes de variáveis significativos: O nome de uma variável deve lembrar qual a informação que está sendo armazenada (mnemônico):

```
double v = 0.042;
double x, y;
...
x = y * v;
….
```

As declarações acima nada dizem sobre o uso da constante e das variáveis.

```
double ValorTaxa = 0.042;
double Taxa, Preco;
...
Taxa = Preco * ValorTaxa;
...
```

No exemplo acima, os nomes das variáveis e da constante ajudam a compreender o que faz o algoritmo.

Alinhamento de comandos: O alinhamento dos comandos, uso de espaços em branco e a colocação de uma instrução por linha aumentam bastante a legibilidade:

```
…..
float x,y,z;
scanf(“%f”, &x);
scanf(“%f”, &x); if((x<=0)||(y<=0))
printf(``Erro"); else { z=sqrt(x*x+y*y);
printf(“%f”,z); }
}
```

O programa acima está correto, mas é difícil de ler e de entender. O programa abaixo é o mesmo programa, porém escrito de uma maneira mais legível:

```
/*
 * Este programa le os dois lados de um triangulo retangulo, calcula o comprimento da hipotenusa e imprime o resultado.
 * A formula usada e' o teorema de Pitagoras.
 * Autor: Eliana
 * Data:  14-08-2001
 */

#include <stdio.h>    // Rotinas de entrada e saida
#include <math.h> // Rotinas matematicas

void main ()
{
   float lado1, lado2; // Catetos do triangulo
   float hipotenusa;   // hipotenusa do triangulo
   scanf(“%f”, &lado1);
   scanf(“%f”, &lado2);
   if ((lado1 <= 0) || ( lado2 <= 0))
     printf(``Erro");
   else
   {
       z = sqrt(x*x + y*y);
      printf(“%f”,z);
    }
}
```

Não existe uma fórmula determinada para se fazer o alinhamento de comandos, mas ma vez que você tenha escolhido uma forma de fazer esse alinhamento, utilize-a sempre de maneira consistente. Por exemplo, o comando if do algoritmo acima poderia ter sido escrito de acordo com o seguinte alinhamento:

```
 if ((lado1 <= 0) || ( lado2 <= 0))
      printf(``Erro");
    else {
         z = sqrt(x*x + y*y);
        printf(“%f”,z);
     }
}
```

Nenhuma das formas acima é mais correta ou mais legível do que a outra. Entretanto, uma vez que uma forma é escolhida, ela deve ser consistentemente mantida na escrita do programa. Seja como for, no decorrer da disciplina daremos preferência à utilização de chaves de abertura e fechamento alinhadas conforme exemplo anterior.

Os editores de textos possuem vários modos de edição para programação que auxiliam na tarefa de tabulação.

Uso de parênteses em expressões: coloque parênteses e espaços em branco em expressões de forma a tornar mais legível a ordem de avaliação dos operadores:

```
2<5 | 15/3==5
```

é mais legível se escrita da seguinte forma:

```
(2 < 5) | ((15 / 3) == 5)
```

Da mesma forma, a expressão abaixo:

```
-i-5*j>==k+1
```

É equivalente à seguinte expressão (bem mais legível):

```
((-i) - (5 * j)) >= (k + 1)
```

Para cada parêntese aberto deve haver um parêntese de fechamento. O número de parênteses de abertura deve ser igual ao número de parênteses de fechamento. Verifique as correspondências entre os parênteses traçando linhas imaginárias conectando cada par de parênteses:

```
((3 + 4) / (5 * 2)) * (4 - 7 * (3 + 8))
||_____|  |___||  |     |_____||
|_________________|  |_______________|
```

Uso de Constantes: A utilização de constantes aumenta a legibilidade, a confiabilidade e a manutenibilidade de um programa. Considere o seguinte programa:

```
void main ( )
   {
      float raio, area;
      scanf(“%f”, &raio);
      area = 3.1416 * raio *  raio;
      printf (“%f”, &area);
   }
}
```

O próximo programa é mais fácil de ler e ser modificado:

```
#define PI 3.1416
void main ( )
   {
      float raio, area;
      scanf(“%f”, &raio);
      area = PI * raio *  raio;
      printf (“%f”, &area);
   }
}
```

Outro exemplo mais convincente:

```
void main ()
{
     int vetint[100];
     double vetreal[100];
      ...

     for (int i = 0; i < 100; i++)
     ...
     for (int cont = 0; cont < 100; cont++)
     ...
     media = soma / 100;
     ...
   }
}
```

Quando esse programa precisar ser modificado para lidar com vetores de tamanhos diferentes, todas as ocorrências de 100 devem ser encontradas e modificadas. Em um programa grande, isso pode ser tedioso e propenso a erros. Um método mais fácil e mais confiável é utilizar uma constante:

```
#define TAM 100

void main ()
{
     int vetint[TAM];
     double vetreal[TAM];
      ...
```

```
    for (int i = 0; i < TAM; i++)
    ...
    for (int cont = 0; cont < TAM; cont++)
    ...
    media = soma / TAM;
    ...
  }
}
```

Agora, quando o tamanho dos vetores precisar ser mudado, somente uma linha deverá ser mudada, independentemente do número de vezes que ela é usada no programa.

Declaraçãao de Variáveis Próxima ao seu Uso: Ao invés de se declarar todas as variáveis utilizadas em um programa no bloco mais externo, uma boa disciplina de programação é declarar as variáveis dentro dos blocos onde elas são utilizadas. Isto aumenta a segurança e a legibilidade do programa, pois a variável é declarada o mais próximo possível de sua utilização. Isto evita a utilização indevida de uma variável, o que poderia provocar um erro, e ajuda na compreensão do programa. Apesar de ser possível declarar variáveis em qualquer local de um programa, o melhor local para a declaração das variáveis é no início de um bloco, pois isto facilita a visualização de quais variáveis estão sendo utilizadas dentro daquele bloco.

```
void main()
{  int x, y;
   double w, z;
   int auxiliar;
   ...
   {
      .....

      {
        ......

        {
          ......
          // troca x e y
          auxiliar = x;
          x = y;
          y = auxiliar;
        }
      }
   }
}
```

No exemplo acima, a variável auxiliar é declarada no bloco mais externo, mas só utilizada em um bloco interno para auxiliar na troca de duas variáveis. O mais correto seria declará-la no bloco mais interno o mais próximo possível do seu uso:

```
void main()
{  int x, y;
   double w, z;
   ...
   {
      .....

      {
        ......

        {   // Declaracao de variaveis no inicio do bloco.
            int auxiliar;
            ......
            // troca x e y
            auxiliar = x;
            x = y;
            y = auxiliar;
        }
      }
   }
}
```

17 - Organização de arquivos no UNIX

- Programas muito extensos devem ser colocados em diferentes arquivos.
- Vantagens:
  - É mais fácil trabalhar com (entender) arquivos pequenos.
  - Permite reutilização.
  - Se um arquivo com código fonte é modificado, somente este é recompilado.
  - Facilita a manutenção e evolução
- Os programadores do Unix utilizam uma maneira padrão para particionar o código.

- Ao dividir o código-fonte em arquivos separados, alguns cuidados devem ser tomados:

  - Agrupe as funções e definições de dados relacionadas em um mesmo arquivo .c
  - Coloque os protótipos das funções públicas e as definições de dados necessárias às mesmas em um arquivo de cabeçalho .h (com o mesmo nome do arquivo .c correspondente), que será incluído pelos arquivos que usarem essas funções;
  - Somente efetue inclusões (#include) de arquivos de cabeçalho (.h). Não inclua arquivos de código .c
  - Não incluir código (comandos executáveis) em arquivos de header .h

17.1 - Extensões

- Arquivos de programas C podem ter as seguintes extensões:
  - .c ou .C ⇒ código C
  - .cc ⇒ código C++
  - .h ou .H ⇒ arquivos de cabeçalho (headers) em C ou C++
- Arquivos em código objeto têm a extensão .o
- Arquivos de bibliotecas (vários .o) têm extensão .a
- Arquivos executáveis têm extensão .exe (DOS) ou sem extensão.

17.2 - Arquivos fonte

17.2.1 - Disposição das informações

As informações de um arquivo fonte devem ser dispostas na seguinte ordem:

- Comentário inicial contendo:
  - descrição e objetivo geral do arquivo
  - nome dos autores
  - data e outras informações importantes
- Inclusão de headers de bibliotecas na seguinte ordem:
  - bibliotecas padrão
  - bibliotecas criadas pelo programador.
  - Exemplo:

```
#include <stdio.h>
#include <mybiblio.h>
```

- Definição de macros e tipos na seguinte ordem:
  - macros constantes (simples)

    - macros com argumentos
    - tipos (typedef) simples
    - tipos estruturados (typedef struct)
    - tipos enumerados (typedef enum)
- Declaração de variáveis na seguinte ordem:
    - externas
    - não estáticas
    - globais
    - estáticas globais
- Declaração de funções na seguinte ordem:
    - ordem alfabética se tiverem utilidade independente
    - agrupadas por utilidades afins
- Definição das funções

17.2.2 - Arquivos header (extensão .h)

- Os arquivos header são destinados a conter:
    - outros arquivos header (include #<outros.h>)
    - definições de macros e tipos, conforme o item 17.2.1
    - declaração de funções (protótipos), conforme o item 17.2.1

- Arquivos headers são incluídos, exatamente como estão escritos, no local onde é encontrada a palavra reservada #include ...

- Pode-se fazer a inclusão de um arquivo header em um outro arquivo de duas formas:

#include <nome.h>
nome.h será procurado no diretório padrão de bibliotecas
ou
#include “nome.h”
nome.h será procurado no diretório corrente
#include “include/nome.h”
nome.h será procurado no diretório include no corrente

- Arquivos header contêm informações para uma finalidade específica, por exemplo:
  - <string.h> para manipular strings
  - <stdio.h> para manipular E/S
  - <math.h> para funções matemáticas

- headers estão associados a uma determinada biblioteca, ou seja, contêm definições e declarações que serão utilizadas pela mesma

- Uma biblioteca pode conter mais de um header associado.
- Exemplos:
  - stdio.h, string.h e outros são headers da biblioteca libc.a
  - O tipo FILE está definido em stdio.h
  - As funções de manipulação de arquivos (e outras) estão declaradas em stdio.h
  - O tipo das funções de manipulação de aquivos que utilizam o tipo FILE, estão definidas em libs.a

- Podem ser vistos como a interface de uma biblioteca, isto é, através dos headers pode-se verificar quais são os parâmetros das funções definidas na biblioteca correspondente.

- Lista de protótipos das funções:
  - São os tipos dos parâmetros e do valor devolvido pela função
  - O padrão Ansi C, exige que esses tipos estejam explicitados.
  - Durante a compilação é feita a verificação de tipos, ou seja, é verificado se os parâmetros passados (quando a função é chamada) correspondem aos argumentos esperados (protótipos).

Exemplo: Seja o seguinte programa:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
  char s1[ ] = “Alo, “;
  char s2[ ] = “ estou aqui!”;
   int p;
   p = strcat(s1, s2);
   printf(“ %s \n “, p);
}
```

- No header *string.h* existe a seguinte declaração para a função strcat():

  char * strcat(char *str1, const char *str2);

- **char *** antes da função indica o tipo de valor retornado pela função, ou seja, ponteiro para caracter.

- No programa acima, o valor retornado é atribuído a **p**, que foi declarada como **int.**
- O compilador no padrão Ansi C vai acusar um erro ao fazer a verificação de tipos, pois **p** deveria ter sido declarado como **char *** ao invés de **int**.

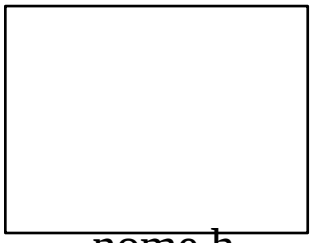
- Ao se criar headers, deve-se colocar no mesmo definições e declarações com uma finalidade comum.

- Alguns códigos são dependentes do hardware, como por exemplo, funções para manipular janelas, etc. Deve-se colocar código dependente do hardware em arquivos separados do código independente. Assim facilita a adaptação, só se mexe no arquivo que contém código dependente.

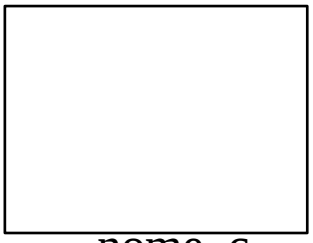
17.2.3 - Arquivos fonte (extensão .c)

- Contém as definições das funções feitas no header correspondente, bem como utiliza os tipos e variáveis especificados no mesmo.
- Outros tipos, variáveis e funções também podem ser declarados e/ou definidos nos arquivos .c
- Fontes e headers correspondentes possuem o mesmo nome, muda apenas a extensão.

Exemplo:



nome.h



nome. c

- Um arquivo fonte .c não necessariamente precisa ter um header associado.
- Um arquivo fonte .c pode incluir vários headers.

- Exemplo de partição do código:

```
#define TAM 20
#define .......
typedef struct  {
          char letra;
          int num;
          } TipoItem;
typedef struct  {
          TipoItem  item;
          Elemento *prox;
          } Elemento;
/*declarações de funções */
void         empilha( .........);
TipoItem  desempilha(.........);
void         FPVazia(.........);
```

pilha.h

```
#include <pilha.h>
void empilha(......)
{
.......
}

TipoItem desempilha(....)
{
.......
}

void FPVazia(......)
{
.......
}
etc ....
```

pilha.c

.....	.....	.....	.....	#include <stdio.h> #include <pilha.h> #include <fila.h> #include <lista.h> .....
lista.h	lista.c	fila.h	fila.c	main.c

Exemplo:

Considere funções que realizam operações com matrizes, estas funções podem ser separadas em dois arquivos distintos:

- operaMatriz.c: contém a definição das funções que realizam operações sobre uma matriz: leitura, impressão, determinante, etc.

- operaEntreMatrizes.c: contém a definição das funções que realizam operações entre duas matrizes: soma, multiplicação, etc.

- main.c: programa que utiliza as funções definidas.

- matriz.h: arquivo header (cabeçalho) contém includes para outros headers, definições de constantes e tipos utilizados, declarações para as funções com matrizes.

Poderiam ter sido criados dois arquivos de header, um para operaMatriz.c e outro para operaEntreMatrizes.c. Como o arquivo main.c não define funções (ou estruturas, tipos, etc) que serão usadas em outros programas, não é necessário criar um arquivo main.h.

Deve-se observar o uso das macros de pré-compilação `#ifdef` e `#define`, para evitar a repetição das definições, caso o mesmo arquivo de cabeçalho seja incluído múltiplas vezes em diferentes locais do código.

_____ operaMatriz.c _____

```
#include <stdio.h>
#include "matriz.h"
*********************************************
Neste arquivo estão definidas funções que realizam operações sobre uma matriz M
Funções: leitura, impressão e verificação se M é um quadrado mágico.
Usam o arquivo de cabeçalho definido pelo usuário matriz.h

Autor: Silvia
Data: 02/09/2017
*********************************************


void leitura (tpMatriz  M, int * n, int * m)
{ int i, j;
  printf("Digite o numero de linhas e colunas,  e depois cada elemento\n");
  scanf ("%d", n);
  scanf ("%d", m);
  for (i=0; i< *n; i++)
    for (j=0; j< *m; j++)
     scanf("%d", &M[i][j]);
}
```

```
void imprime (tpMatriz M, int  n, int  m)
{ int i, j;
   for (i=0; i<n; i++)
   {
     for (j=0; j<m; j++)
      printf("%d ", M[i][j]);
     printf("\n");
   }
}

int quadradoMagico (tpMatriz M, int n)
{
   int i, j, somads, somadp, somaLin[TAM], somaCol[TAM];

   for  (i=0; i<n; i++)  // inicializar vetores com 0, o gcc assume que todos os inteiros tem valor default de 0.
      somaLin[i] = somaCol[j] =0;
   somadp = somads = 0;
   for (i=0;  i<n; i++)
   {
      for (j=0;  j<n; j++)
      {
         somaLin[i] += M[i][j];
         somaCol[j] += M[i][j];
         if (i==j)
           somadp += M[i][j];
```

```
        if ((i+j == n-1))
          somads += M[i][j];
      }
    }

    if (somadp == somads)
    {
      // verifica se a soma das colunas e das linhas são iguais as das diagonais
      for (i=0; (i<n  && somaLin[i] == somads && somaCol[i] == somads); i++);

      if (i==n)  // se i = n todas foram iguais.
        return (1);
      else
       return (0);
    }
    else
      return (0);
}
```

_____ operaEntreMatrizes.c_____

```
#include "matriz.h"

/*************************************************
Neste arquivo estão definidas funções que realizam operações entre duas matrizes
entradas: M1 e M2, gerando uma terceira matriz saida: M3.

Funções: soma e multiplicação de matrizes
Usam o arquivo de cabeçalho definido pelo usuário matriz.h
Autor: Silvia
Data: 02/09/2017
*************************************************/
int somaMatrizes(int M1[][TAM], int n1, int m1, int M2[][TAM], int n2, int m2, int M3[][TAM])
{  int i, j;
   if (m1==m2 && n1==n2)
   {
     for (i=0; i<m1; i++)
       for (j=0; j<n1; j++)
         M3[i][j] = M1[i][j] + M2[i][j];
      return (1);
   }
   else
    return (0);
}
```

```
int multiplicaMatrizes(int  M1[][TAM], int n1, int m1, int M2[][TAM], int n2, int m2, int M3[][TAM])
{ int i, j;
   if (m1==n2 && n1==m2)
   {
     for (i=0;i<m1; i++)
       for (j=0; j<m1; j++)
         for (int k=0; k<n1; k++)
           M3[i][j] = M3[i][j] + (M1[i][k] * M2[k][j]);
     return (1);
   }
   else
    return (0);
}
```

_____ mainMatriz.c_____

```
#include "matriz.h"
/*************************************************************
Este programa lê uma matriz M, verifica se ela é um quadrado
mágico e logo em seguida realiza a soma desta matriz com ela
mesma gerando a matriz M2.

Usa o arquivo de cabeçalho definido pelo usuário matriz.h
Autor: Silvia
Data: 02/09/2017
************************************  /
void main()
{
  int i, j, m, n, M[TAM][TAM],  M2[TAM][TAM];

  leitura(M,&n,&m);

  if  (quadradoMagico (M,n))
   printf ("Quadrado Magico\n");
  else
    printf ("Não é Quadrado Magico\n");

  if (somaMatrizes(M,m,n,M,m,n,M2))
    imprime(M2,m,n);
}
```

Para compilar:

```
gcc main.c operaMatriz.c operaEntreMatrizes.c -oexe
```

gcc -c operaEntreMatrizes.c => gera o arquivo operaEntreMatrizes.o

gcc -c operaMatriz.c => gera o arquivo operaMatriz.o

gcc -c mainMatriz.c => gera o arquivo mainMatriz.o

isso pode ser útil para não precisar recompilar todos os arquivos.

gcc -oexe *.o => para linkar depois os arquivos objetos gerados.

_____matriz.h_____

```
/*
#ifndef _ __
#define __   __
*/


#include <stdio.h>

#define TAM 100

typedef  int tpMatriz[TAM][TAM];



void leitura (tpMatriz, int*, int*);
void imprime (tpMatriz, int, int);

int quadradoMagico (tpMatriz, int);

int somaMatrizes(tpMatriz, int, int,
tpMatriz, int, int, tpMatriz);

int multiplicaMatrizes(tpMatriz, int, int,
tpMatriz, int, int,tpMatriz);
```