Orientações para Escrita de Programas
(Mini Manual de Estilo de Programação)

1. Introdução

Este documento apresenta um resumo de regras de estilo que o aluno deve seguir ao escrever programas. Você também é aconselhado a consultar o **Capítulo 6** (*Legibilidade e Depuração de Programas*) do livro-texto recomendado e o guia de estilo *Indian Hill*, encontrado no site da disciplina e em vários outros sites na internet.

Em geral, é crucial que você apresente não apenas um programa que simplesmente funciona, mas leve ainda em consideração os critérios de ***legibilidade***, ***manutenibilidade*** e ***portabilidade***, que classifiquem o produto final como um ***programa de qualidade***. Um bom programa reflete um bom projeto de solução para o respectivo problema. Portanto, pense bastante sobre a linha de solução (algoritmo) a ser seguida antes de começar a codificar seu programa. Nessa disciplina, você não precisa preocupar-se em apresentar o programa mais eficiente possível para um dado exercício de programação. Apenas siga as recomendações de estilo contidas no presente documento.

2. Documentação

2.1 Comentários

Comentários devem ser utilizados para clarificar o programa e devem consistir de sentenças claras, escritas em bom e conciso Português. Idealmente, os comentários devem ser escritos simultaneamente com a construção do programa. Se você é daqueles que terminam de escrever o programa para depois comentá-lo, experimente o inverso: escreva os comentários primeiro e depois escreva o programa...

A seguir, serão apresentadas descrições resumidas dos comentários essenciais que devem constar em cada programa. (Maiores detalhes podem ser encontrados no **Capítulo 6** do livro-texto.)

2.1.1 Blocos de Comentário

Blocos de comentários devem ser inseridos no início do programa [i.e., no início do arquivo contendo a função `main()`], no início de cada arquivo de um programa constituído de vários arquivos e antes da definição de cada função.

O seguinte formato é recomendado para um bloco de comentário colocado no início de um programa:

```
/****
 *
 * Título:
 *
 * Autor:
 *
 * Data de Criação:
 * Última modificação:
 *
 * Descrição: [i.e., o que e como o programa faz]
 *
 * Entrada: [i.e., o tipo de entrada esperada; um exemplo pode ser recomendável]
 *
 * Saída: [i.e., a saída esperada; um exemplo pode ser recomendável]
 *
 ****/
```

Por exemplo:

```
/****
 *
 * Título: CalculaMedia
 *
 * Autor: José da Silva
 *
 * Data de Criação: 01/04/2001
 * Última modificação: 10/06/2001
 *
 * Descrição: Este programa lê um arranjo de, no máximo, 50 números
 *            reais e calcula a média dos elementos deste arranjo. A
 *            média é calculada por meio da fórmula:
 *            (soma dos valores)/(número de valores)
 *
 * Entrada: A entrada consiste de uma seqüência de números reais precedidos
 *          por um número inteiro positivo indicando o número de valores
 *          reais que serão introduzidos. Cada dado é introduzido numa
 *          linha separada. Por exemplo:
 *          3     (número de valores)
 *          2.0   (primeiro valor, etc.)
 *          3.0
 *          5.0
 *
 * Saída: O programa imprime na tela do computador o valor da média dos
 *        valores reais introduzidos.
 *
 ****/
```

O seguinte formato é recomendado para um bloco de comentário precedendo uma definição de função:

```
/****
 *
 * Título:
 *
 * Autor:
 *
 * Data de Criação:
 * Última modificação:
 *
 * Descrição: [i.e., o que e como o a função faz]
 *
 * Parâmetros: ... (entrada): [descrição do parâmetro]
 *             ... (saída): [descrição do parâmetro]
 *             ... (entrada/saída): [descrição do parâmetro]
 *
 * Valor de retorno: [descrição do valor retornado pela função]
 *
 ****/
```

A *Descrição* no bloco de comentário que precede uma definição de função deve descrever o algoritmo utilizado e pode fazer referência a variáveis locais ou parâmetros utilizados. Algoritmos clássicos (por exemplo, *quicksort*) não precisam ser descritos em detalhes, apenas mencionados. Algoritmos triviais também não precisam ser descritos em detalhes.

Nesse bloco de comentário, cada parâmetro deve ser descrito separadamente e seu modo (*entrada*, *saída* ou *entrada*/*saída*) deve ser citado. Deve-se ainda descrever aquilo que a função retorna (i.e., o *significado* do valor retornado). Se a função não possui parâmetros, escreve-se *nenhum* após *Parâmetros:*. Se a função não retorna nada (i.e., se seu tipo de retorno é **`void`**), escreve-se *nenhum* após *Valor de retorno:*. Um engano freqüentemente cometido pelos iniciantes é descrever o *tipo de retorno* da função (em vez do significado do valor retornado). Esta última informação não merece nenhum comentário, já que o tipo de retorno de uma função é claro no cabeçalho da mesma.

Exemplo de um comentário de bloco colocado antes de uma definição de função:

```
/****
 *
 * Título: LeValor()
 *
 * Autor: Ulysses de Oliveira
 *
 * Data de Criação: 21/10/2002
 * Última modificação: 25/10/2002
 *
 * Descrição: Lê um string no meio de entrada padrão e converte-o para o tipo double.
 *            [Descrição do algoritmo utilizado ...]
 *
 * Argumentos: Saída: valor - ponteiro para double que apontará para o
 *                           valor lido em caso de sucesso. Caso a função
 *                           não seja bem sucedida, *valor é indefinido
 *             Entrada: pfVerifica - ponteiro para uma função de verificação, que
 *                      recebe um valor double como entrada e retorna 1 se o valor
 *                      lido satisfaz os critérios de validade e 0, em caso contrário.
 *
 * Retorno: 1, se bem sucedida, ou 0, caso contrário.
 *
 ****/
```

Observe que, como demonstrado neste último exemplo, você não precisa seguir obrigatoriamente o *formato* do modelo sugerido. Isto é, você é livre para usar sua criatividade para criar seu próprio estilo de comentário. O que se espera é que o ***conteúdo*** seja aquele sugerido nos formatos de comentários apresentados acima.

Um arquivo contendo funções relacionadas por afinidade num programa com múltiplos arquivos, deve conter um bloco similar ao do exemplo acima contendo informações sobre o propósito geral (i.e., comum) das funções daquele arquivo. Cada função no arquivo deve conter seu próprio bloco de comentários específico para ela.

2.1.2 Comentários em Declarações de Variáveis

Cada variável que tem papel significativo na função ou programa deve ter um comentário associado à sua declaração que clarifica o papel da variável.

```
float   pesoDoAluno;    /* O peso do aluno em quilogramas */
```

Variáveis que não sejam significativas *não precisam ser comentadas*. O comentário a seguir, por exemplo, é absolutamente dispensável:

```
unsigned  int  i;    /* Variável utilizada como contador no laço for */
```

2.1.3 Comentários para Seções de Código

Cada seção (sequência de instruções) de uma função que executa uma dada tarefa, deve ser precedida por um bloco de comentário explicando o propósito da seção e o algoritmo utilizado pela mesma (se este não for trivial). Este comentário não dispensa outros comentários de linha (v. a seguir) quando estes se fizerem necessários para clarificar alguma instrução nesta seção. Por exemplo:

```
      /* O trecho de programa a seguir calcula tal coisa de acordo com  */
      /* o seguinte algoritmo: ...                                      */
      [inicie a seção aqui]
```

2.1.4 Comentários de Linha

Comentários de linha podem ser colocados à direita da instrução comentada (se houver espaço) ou antes da mesma. Exemplos:

```
x = y + 10*F(y);  /* Esta instrução calcula alguma coisa...  */
```

ou

```
            /* A instrução a seguir calcula tal coisa de tal e tal modo...  */
minhaVariavelTotal = minhaVariavelParcial + 10*MinhaFuncao(minhaVariavelParcial);
```

2.1.5 Comentários no Final de Blocos

Este tipo de comentário é necessário para indicar a quem pertence um dado fecha-chaves ("}") em blocos ***longos*** ou ***aninhados***. Estes comentários são dispensáveis quando um bloco é suficientemente curto. Por exemplo:

```
while (x){
     ...
     if  (y > 0){
          ...
     } /* Final do if */
     ...
}  /* Final do while */
```

2.1.6 Outros Comentários Necessários

Comentários são ainda recomendados nos seguintes casos:

- Uma instrução deliberadamente vazia.
- Um corpo de função deliberadamente vazio.
- No caso de uma instrução **`switch`** na qual instruções pertencentes a mais de um caso poderão ser executadas (i.e., na ausência deliberada de um **`break`**).
- Aonde o bom senso recomendar...

2.1.7 Comentários Desnecessários

Comentários devem ser escritos por programadores para programadores proficientes na linguagem utilizada no programa e têm o objetivo de documentar e esclarecer pontos que de outra forma seriam ambíguos ou difíceis de entender. Isto significa, entre outras coisas, que comentários não precisam serem didáticos ou escritos para leigos (como ocorre nos livros e manuais de programação). Portanto, não comente instruções que tenham significado óbvio. Por exemplo, o comentário a seguir seria condenável:

```
x = y + 2;  /* A variável x recebe o valor da variável y mais 2 */
```

2.2 Manual do Usuário

Além dos comentários dentro do programa, você também deve criar um **manual do usuário**; i.e., um arquivo-texto (e.g., do tipo *`readme`* ou *`leiame`*) contendo instruções para os usuários do programa. Este

manual não deve conter detalhes técnicos (i.e., ele deve ser dirigido para um usuário leigo em programação). Este requisito é necessário apenas para projetos de programação maiores, como o projeto final de curso.

3. Formatação do Programa

A seguir serão enumeradas algumas dicas de formatação de programas que ajudam a melhorar a legibilidade dos mesmos.

3.1 Endentação

Endentação deve ser utilizada para indicar que as instruções endentadas estão sob controle da instrução anterior não-endentada. O uso ***consistente*** de endentação é essencial para legibilidade do programa. Sugestões para endentação são apresentadas a seguir. Você não precisa seguir *exatamente* essas sugestões, mas, qualquer que seja sua escolha, seja consistente!

3.1.1 Quando Endentar

Use sempre endentação para instruções dentro de blocos (i.e., entre um par de colchetes), ou para instruções que fazem parte de alguma estrutura de controle ou seguindo um rótulo. Exemplos:

```
{
    <instrução 1>;
    ...
    <instrução N>;
}
```

```
while  (x) {
    <instrução 1>;
    ...
    <instrução N>;
}
```

3.1.2 Espaço de Tabulação

Fixe o espaço de tabulação em quatro para endentação (esta medida não é aleatória, mas sim resultado de pesquisa). Espaços menores do que este podem não ser muito legíveis, enquanto que espaços maiores o farão chegar rapidamente ao final da linha na tela.

3.1.3 Rótulos

As instruções seguindo um rótulo utilizado em conjunção com um **goto**, devem ser endentadas com relação ao rótulo:

```
istoEhUmRotulo:
    [Instruções a serem executadas quando o programa for desviado para cá]
```

3.1.4 Estruturas de Controle

A seguir, estão enumeradas algumas sugestões para formatação de estruturas de controle:

if-else

```
if (x) {
     ...
}
else  if (y){
     ...
}
else {
     ...
}  /* Final do if */
```

switch

```
switch (x) {
   case 'A' :
      [Escreva aqui as instruções correspondentes à opção A]
      break;
   ...
   default:
      [Escreva aqui as instruções correspondentes à opção default]
      break;
}   /* Final do  switch */
```

while

```
while (x) {
     ...
}  /* Final do while */
```

do-while

```
do {
     ...
}  while (x);
```

for

```
for (i = 0; i < 9; i++){
     ...
}  /* Final do for  */
```

O uso de abre-chaves na mesma linha inicial de uma estrutura de controle tem com vantagem reduzir o risco de se encerrar acidentalmente a estrutura de controle colocando um ponto-e-vírgula nesta posição (um erro muito freqüente em programação em C). Por exemplo, é muito mais fácil cometer o erro:

```
while (x);  /* O ponto-e-vírgula encerra a instrução while */
{
     ...
```

do que o erro:

```
while (x);{       /* Aqui é mais fácil perceber o erro */
     ...
```

3.2 Uso de Espaços em Branco

O uso judicioso de espaços em branco é essencial para a legibilidade do programa. Isto inclui espaços horizontais em torno de operandos e operadores em expressões, em torno de comentários, para alinhar identificadores sendo declarados, etc. Espaços verticais também são importantes para separar funções, blocos, conjuntos de instruções com alguma afinidade lógica dentro de um bloco, etc. Entretanto, ***não exagere*** no uso de espaços em branco, pois o uso exagerado de espaços em branco prejudica a legibilidade, ao invés de melhorá-la.

3.3.1 Espaços Horizontais

Além de servirem como endentação, espaços horizontais devem ainda ser utilizado em torno de identificadores e operadores para melhorar a legibilidade de expressões.

• Use espaços horizontais para enfatizar precedência de operadores. Por exemplo,

```
5*3 + 4
```

é melhor do que:

```
5 * 3 + 4
```

• Use espaços horizontais para alinhar identificadores numa seção de declaração, e.g.,

```
short                   var1;
register  long  int     var2;
```

3.2.2 Espaços Verticais

Use espaços verticais em branco para separar seções logicamente diferentes do programa (ou função). Mas, não separe ***todas*** as instruções com espaços em branco entre as mesmas; instruções que têm alguma afinidade lógica (i.e., instruções que fazem mais sentido quando vistas em conjunto) devem ser mantidas juntas para indicar esta afinidade.

4. Estilo de Codificação

4.1 Instruções e Expressões

• Escreva instruções e expressões que sejam razoavelmente curtas.

• Evite instruções que ocupam mais de uma linha. Quando isto for realmente inevitável endente as linhas seguintes com relação a linha inicial da instrução. Por exemplo, escreva:

```
resultado = (minhaVariavel1 + minhaVariavel2)*minhaVariavel3 -
              minhaVariavel4 + minhaVariavel5;
```

ao invés de:

```
resultado = (minhaVariavel1 + minhaVariavel2)*minhaVariavel3 -
minhaVariavel4 + minhaVariavel5;
```

• Evite o uso de constantes numéricas em instruções ou expressões. Em vez disto, substitua valores que têm algum significado por constantes simbólicas. Por exemplo:

```
perimetro = 2*3.14*r;
```

deve ser rescrito como:

```
#define PI 3.14
...
perimetro = 2*PI*r;
```

Note que o valor `3.14`, que tem um significado, foi substituído pela constante simbólica `PI` na instrução, mas o valor `2` permaneceu, visto que este valor não tem nenhum significado, simplesmente faz parte de uma fórmula (você seria capaz de dar um nome para o valor `2`?).

• Evite a escrita de blocos muito longos. Tente escrever blocos que caibam inteiramente na tela, pois isto facilita a leitura dos mesmos (para depuração, por exemplo).

• Evite o uso de expressões relacionais muito longas e complexas. Uma forma de se verificar se uma expressão relacional é muito complexa é utilizar o chamado *teste do telefone*: leia a expressão em voz alta; se você conseguir entender a expressão à medida em que a lê, a expressão passa no teste; caso contrário, se você se deixar de acompanhá-la, é melhor dividir a expressão em subexpressões aninhadas.

4.2 Escolha de Identificadores

4.2.1 Convenções

O uso consistente de convenções para a criação de identificadores para as diversas entidades que compõem um programa facilita bastante leitura do programa. A seguir, são resumidas algumas sugestões gerais para criação de identificadores.

• **Nomes de variáveis**. Comece com letra minúscula; se o nome da variável for composto utilize letra maiúscula no início de cada palavra seguinte, inclusive palavras de ligação (e.g., preposições e conjunções); não utilize sublinha. Comece identificadores de variáveis globais com a letra `g`. Por exemplo, `gMinhaVariavelGlobal`.

• **Nome de tipos**. Siga a regra acima para nomes de variáveis, mas comece com a letra "`t`" ou termine com "`_t`". Exemplo, `tListaEncadeada` (ou `listaEncadeada_t`)

• **Nomes de funções**. Utilize a mesma regra para nomes de variáveis, mas comece com letra maiúscula.

• **Nomes de macros**. Utilize sempre letras maiúsculas; se o nome for composto, utilize sublinha para separar os componentes.

Como no caso de outros elementos de estilo apresentados anteriormente, existem outras convenções para criação de identificadores (uma convenção famosa, mas de gosto duvidoso, é o chamado *estilo húngaro*). Novamente, como recomendado antes, você tem liberdade de escolher outra convenção, mas, qualquer que seja sua escolha, seja consistente!

4.2.2 Representatividade

• Identificadores que exercem papeis importantes no programa devem ter nomes que sejam significativos com relação aos respectivos papeis exercidos (e.g., `nomeDoAluno` é muito melhor do que simplesmente `x`).

• Identificadores com importância menor, não precisam ter nomes significativos. Por exemplo, uma variável utilizada apenas como variável de controle num laço for pode ser denominada `i` (i.e., não precisa ser denominada `contador`, por exemplo).

• Não siga como exemplos as nomenclaturas de funções de biblioteca de C, pois elas são horríveis em termos de legibilidade e coerência.

• Não utilize identificadores nem muito longos nem muito abreviados: encontre um meio-termo que seja sensato e legível.

5. Interface do Usuário

O estudo daquilo em que consiste uma boa interface do usuário constitui por si só uma disciplina à parte. Entretanto, pode-se ter um conjunto mínimo de recomendações básicas, ditadas principalmente pelo bom senso, que devem ser seguidas por todos os programas solicitados ao longo do curso. Essas recomendações básicas são as seguintes:

- Todo programa interativo deve iniciar sua execução apresentando ao usuário informações sobre o que o programa faz, seu autor, versão e qualquer outra informação pertinente ***para o usuário*** do programa.

- Toda entrada de dados deve ser precedida por uma indicação (*prompt*) informando ao usuário o tipo de entrada que o programa espera que o usuário introduza e o formato destes dados.

- Se for o caso, o programa deve ainda informar ao usuário que ação ele deve executar numa certa entrada de dados, se esta ação não for óbvia. Por exemplo, você não precisa dizer ao usuário para *pressionar uma tecla* para introduzir um caracter, basta dizer *digite um caracter*. Mas, você precisa informar ao usuário como executar uma operação não usual, tal como *pressione simultaneamente as teclas* `ALT`, `SHIFT` *e* `A`, ao invés de *digite* `ALT-SHIFT-A`.

- Toda saída de dados deve ser compreensível para o usuário.

- O programa deve informar ao usuário o que ele deve fazer para encerrar o programa ou uma dada iteração.

- Tenha sempre em mente que o programa deverá ser usado por ***usuários comuns*** e não por profissionais de computação. Portanto, não faça suposições sobre o nível intelectual dos usuários (a não ser que isto seja explicitamente pressuposto no enunciado do problema).

6. Portabilidade

Sempre que escrever um programa em C, certifique-se que o compilador utilizado segue o padrão ANSI/ISO. Portanto, não utilize extensões da linguagem C (por exemplo, extensões da Borland) específicas de alguma implementação de compilador. Você pode utilizar funções de biblioteca específicas de alguma plataforma (por exemplo, a função `clrscr()` do módulo `conio`), mas, neste caso, chame atenção por meio de comentários para o fato de estar usando instruções que não são portáveis.

7. Forma de Apresentação dos Programas

A resolução de qualquer Lista de Exercícios de Programação deve conter o seguinte:

• **Listagem** impressa do programa (capa bonita não é necessária!)

• **Exemplo** de uma ou mais sessões de execução do programa contendo os dados de entrada e as saídas resultantes. (Leia o documento intitulado *Redirecionamento de Entrada e Saída no DOS* e aprenda a redirecionar entrada e saída para arquivos-texto, pois este conhecimento facilita esta tarefa de apresentação de exemplos de sessões dos seus programas.)

Além disso, você pode ser solicitado a apresentar uma seção de execução de seu programa. Esteja, portanto, preparado para ser argüido.

Nas apresentações dos trabalhos utilize as seguintes recomendações:

- **Apresentação dos programas:**

  1. Selecione todo o conteúdo do programa no BC++
  2. Copie o conteúdo selecionado
  3. Cole no documento do Word que será utilizado para apresentar o trabalho
  4. Selecione todo texto no novo documento e utilize a fonte *Courier New 9*.
  5. Idealmente, você também deve colocar as palavras-chave do programa em negrito e corrigir as endentações.

- **Apresentação de sessões de execução de programas:**

  1. Selecione o conteúdo da sessão usando o laço do DOS (console), que é um botão tracejado no topo da janela do console.
  2. Copie o conteúdo selecionado usando barra de ferramentas.
  3. Cole no documento do Word que será utilizado para apresentar o trabalho
  4. Selecione todo texto no novo documento e utilize a fonte *Courier New 9*.

8. Metodologia de Avaliação de Programas

Os programas serão avaliados, numa escala de 0 a 10 pontos, divididos em dois critérios gerais:

(1) **funcionalidade do programa**, e
(2) **estilo de programação**.

A avaliação em cada um destes critérios corresponde a, no máximo, cinco pontos.

8.1 Funcionalidade do Programa

A tabela a seguir apresenta algumas prováveis características apresentadas por programas classificados com os respectivos pontos no critério de ***funcionalidade do programa***.

Pontuação	Características do Programa
5	Absolutamente (ou convincentemente) correto e robusto.
4	Quase correto; contém alguns ***bugs*** triviais que não comprometem o funcionamento do programa como um todo; não prevê todas as entradas possíveis e pode quebrar em situações não usuais.
3	Basicamente correto, mas contém alguns erros comprometedores; *quebra* com relativa facilidade.
2	Abordagem de solução do problema é razoável, mas contém muitos erros; *quebra* com muita facilidade.
1	Basicamente, apenas sintaticamente correto; o programa funciona, mas os resultados são errados.
0	Não consegue sequer ser compilado devido a erros de sintaxe.

8.2 Estilo de Programação

A tabela a seguir apresenta algumas prováveis características apresentadas por programas classificados com os respectivos pontos no critério de ***estilo de programação***.

Pontuação	Características do Programa
5	Estilo perfeito; uma obra de arte.
4	Algoritmos e estruturas de dados adequados; boa interface do usuário (em termos de apresentação e facilidade de uso); bem documentado; tão simples quanto possível; identificadores bem escolhidos; fácil de ler e entender.
3	Algoritmos e estruturas de dados não foram bem escolhidos; interface do usuário rudimentar; documentação e clareza descuidadas; mais complexo do que deveria; identificadores não são representativos; razoavelmente fácil de ler e entender.
2	Algoritmos e estruturas de dados ruins; interface do usuário que apenas o próprio programador sabe usar; documentação ruim ou ausente; difícil de ser entendido.
1	Entende-se com muito sacrifício.
0	Ilegível; não é um trabalho digno de um aluno de Computação.

9. Programas Fraudulentos

Programas em que forem constatados algum tipo de fraude receberão nota 0,0 (zero). Na prática, isto significa que, se o seu programa for semelhante a algum outro além da casualidade, ambos receberão nota zero, independentemente do fato de você ter copiado ou ter sido copiado. Você é responsável por sua proteção. Portanto, proteja-se!

Você pode incluir em seus exercícios de programação, porções de código (mas, não programas inteiros!) encontradas em livros e em outras fontes. Se este for o caso, você deve incluir na documentação (i.e., nos comentários) do programa sua *fonte de inspiração*. Se você não proceder desta maneira e sua fonte for descoberta, seu programa será considerado fraudulento e receberá a devida avaliação...

10. Prazos de Apresentação dos Programas

O prazo legal de apresentação de cada lista de exercícios de programação é aquele indicado no documento contendo os enunciados dos exercícios respectivos. ***Cada dia (ou fração de dia) de atraso valerá o desconto de um ponto na nota atribuída ao programa***. Por exemplo, em termos práticos, se sua lista de exercícios de programação valer 10, você receberá apenas 1 se esta lista for entregue nove dias além do prazo.

Última alteração: 14/03/2011