Regras de estilo para código fonte em C/C++

No processo de escrita de código fonte em uma linguagem de alto nível é interessante que o programador se preocupe não somente com a correção sintática e lógica do código (ou seja, se o programa está funcionando corretamente de acordo com o seu objetivo), mas também com a organização do texto do programa em si.

A vantagem mais evidente de um texto de programa melhor organizado é a maior facilidade de entendimento e manutenção posteriores; não são raros os casos em que o próprio programador tem dificuldade em entender, após algum tempo, o código por ele escrito devido ao fato de este não estar organizado de maneira que facilite a sua compreensão.

O exemplo a seguir ilustra como um código-fonte em linguagem C pode ser escrito de forma desorganizada, dificultando o seu entendimento:

Exemplo 1 - Código escrito sem regras de organização

```
#include <stdio.h>

void main(void)
{
unsigned long int a, b, c, d, e, f;
printf("Digite o número total de elementos:");
scanf("%lu", &a);
printf("Digite o número de elementos de cada grupo: ");
scanf("%lu", &b);
if (a < b) printf("A deve ser maior ou igual a B");
else
{
c=1;
d=a;
while (d>1)
{ c=c*d;
d--; }
e=1;
d=b;
while (d>1)
{ e = e*d;
d--; }
f=1;
d=a-b;
while (d>1)
{ f=f*d;
d--; }
printf("Número de combinações possíveis: %lu", c/(e*f));
}
}
```

O código apresentado calcula o número de combinações possíveis de *n* elementos, *p* a *p* (onde *n* é o número total de elementos e *p* é o número de elementos de cada grupo).

Embora seja um problema de solução simples, uma simples leitura do texto do programa não deixa clara a sua finalidade devido a problemas de organização.

O mesmo programa pode ser escrito utilizando algumas regras simples de organização:

Exemplo 2 - Código escrito utilizando-se regras de organização

```
#include <stdio.h>

/*--------------------------------
 Função: fatorial
 Parâmetros: n - numero do qual se calculara o fatorial
 Retorno: valor do fatorial de n
 Descrição: calcula e retorna o fatorial de n
--------------------------------*/
unsigned long int fatorial (unsigned long int n)
{
      unsigned lont int fat = 1; /*valor inicial do fatorial*/
      while (n > 1)
      {
            fat = fat * n;
            n--;
      }
      return fat;
}

void main(void)
{
      unsigned long int elem_total; /*numero total de elementos*/
      unsigned long int elem_grupo; /*numero de cada grupo*/
      unsigned long int num_comb;   /*numero de combinacoes*/
      printf("Digite o número total de elementos:");
      scanf("%lu", &elem_total);
      printf("Digite o número de elementos de cada grupo: ");
      scanf("%lu", &elem_grupo);
      /*Testa se o numero total eh maior ou igual que o numero de
cada grupo*/
      if (elem_total < elem_grupo)
            printf("Total deve ser maior ou igual a grupo");
      else
      {
            num_comb = fatorial(elem_total) /
            (fatorial(elem_grupo) * fatorial(elem_total -
            elem_grupo));
      }
      printf("Número de combinações possíveis: %lu", num_comb);


}
```

Neste caso, a aplicação de algumas regras tornou o código mais facilmente inteligível (e, por consequência, mais facilmente alterável). Vamos analisar cada uma destas regras em detalhes.

1. Identação

No Exemplo 1, percebe-se que todas as linhas do código-fonte possuem o mesmo alinhamento à esquerda. Com este tipo de organização não fica claro, a princípio, quais são as estruturas que compõem o programa, onde elas se iniciam e onde terminam, dificultando o entendimento do código e também sua manutenção.

O uso de **identação** determina que cada bloco lógico do programa seja marcado por um espaçamento à direita em relação ao bloco anterior, tornando mais fácil identificar a estruturação do programa e a sua lógica no que diz respeito às estruturas de controle e aos módulos funcionais. No Exemplo 2 o espaçamento utilizado foi de 6 caracteres, porém espaçamentos de 2 a 4 caracteres são mais adequados para programas que utilizam um grande número de blocos lógicos encadeados e, portanto, um espaçamento máximo elevado em relação à margem esquerda.

Podemos considerar como blocos lógicos:

- corpo de função
- corpo de estrutura condicional
- corpo de estrutura de repetição
- corpo de definição de estrutura de dados (*struct*)

Em linguagem C, os blocos lógicos devem ser delimitados por chaves (à exceção dos blocos formados somente por uma instrução e que não são corpo de função). No Exemplo 2 as chaves foram alinhadas com a identação do bloco anterior ao bloco lógico sendo iniciado:

```
while (n > 1)
{
      fat = fat * n;
      n--;
}
```

alinhadas com o bloco anterior

Outra regra utiliza a chave de fechamento alinhada com a identação do bloco anterior, porém a chave de abertura posicionada no final da última instrução (ou cabeçalho) do bloco anterior:

```
while (n > 1) {
      fat = fat * n;
      n--;
}
```

fim da última instrução

alinhada com o bloco anterior

Nos blocos formados somente por uma instrução o uso de chaves é recomendado, porém não obrigatório. Caso não se queira utilizar as chaves, uma boa prática consiste em identar a instrução normalmente, posicionando-a na linha seguinte à última instrução (ou cabeçalho) do bloco anterior. Esta prática facilita o trabalho de depuração, já que a grande maioria dos depuradores se orienta pelas linhas do código-fonte para a execução passo a passo.

```
if (elem_total < elem_grupo)
    printf("Total deve ser maior ou igual a grupo");
```

identada e posicionada na linha seguinte

2. Nomes de variáveis

O identificador (nome) atribuído a uma variável deve expressar, na medida do possível, a finalidade daquela variável. No Exemplo 1, ao nomear-se as variáveis como *a, b, c, d, e* e *f* não se conseguiu expressar claramente a sua finalidade; nomes compostos por um único caracter não diminuem o tamanho do código executável gerado e devem ser evitados, a menos que sejam utilizados somente como auxiliares ou que exprimam claramente o objetivo para o qual foram declaradas as variáveis (por exemplo, coeficientes de uma equação).

Existem várias boas práticas para atribuição de nomes de variáveis. A regra utilizada no Exemplo 2 foi a de obter uma sentença que exprimisse o objetivo da variável (por exemplo, "número de combinações"), abreviar as palavras da sentença e gerar um identificador com as abreviaturas unidas por sublinhado (por exemplo, "num_comb").

Muitos programadores utilizam regras de nomeação derivadas da notação húngara, criada pelo Dr. Charles Simonyi, arquiteto-chefe da Microsoft. A notação húngara é uma convenção que adiciona um prefixo a um identificador para indicar o seu tipo funcional, e que por sua utilidade passou a ser amplamente utilizada, com pequenas adaptações, dentro da própria Microsoft para desenvolvimento de código de forma mais eficiente. No Exemplo 2, caso se desejasse utilizar a notação húngara as variáveis poderiam ser nomeadas da seguinte maneira (onde *ul* seria o prefixo convencionado para uso com variáveis do tipo *unsigned long int*):

```
unsigned long int ulElemTotal; /*numero total de elementos*/
unsigned long int ulElemGrupo; /*numero de cada grupo*/
unsigned long int ulNumComb;   /*numero de combinacoes*/
```

3. Comentários

Um programa bem documentado deve conter comentários, principalmente nos pontos onde uma simples leitura do código não é suficiente para se entender a sua finalidade. Em adição, declarações de variáveis também devem ser documentadas já que

os seus próprios identificadores, mesmo quando são adequadamente escolhidos, nem sempre são suficientes para uma compreensão da finalidade a que elas se destinam.

Em implementações de funções, uma boa prática consiste em se introduzir um comentário antes do cabeçalho contendo as seguintes informações:

- nome (identificador) da função
- descrição dos parâmetros (dados de entrada) da função
- descrição do retorno da função (se houver)
- descrição do funcionamento

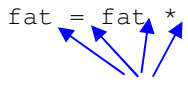
No Exemplo 2:

```
/*--------------------------------
 Função: fatorial
 Parâmetros: n - numero do qual se calculara o fatorial
 Retorno: valor do fatorial de n
 Descrição: calcula e retorna o fatorial de n
---------------------------------*/
unsigned long int fatorial (unsigned long int n)
{
      unsigned lont int fat = 1; /*valor inicial do fatorial*/
      while (n > 1)
      {
            fat = fat * n;
            n--;
      }
      return fat;
}
```

Caso exista um protótipo da função inserido em um arquivo de cabeçalho (*header file*, com a extensão .h), também é boa prática introduzir o mesmo comentário antes do protótipo, facilitando a compreensão daquela função mesmo antes de se ter acesso ao código da implementação.

4. Espaçamentos e quebras de linha

Expressões envolvendo operadores, tanto lógicos quanto aritméticos, devem conter espaços separando os operandos dos operadores para facilitar a sua leitura.



espaços entre operandos e operadores

Em casos nos quais seja necessário quebrar uma expressão ou chamada de função em mais do que uma linha, utilizar uma das seguintes regras:

- quebrar após uma vírgula

- quebrar após um operador

No caso de impressão de strings (utilizando *printf* ou equivalente) com uma grande quantidade de caracteres é interessante utilizar várias chamadas de função em sequência, evitando que uma linha de código contenha um número excessivo de caracteres. Esta providência facilita a navegação no código-fonte e também a sua impressão.