Regras e Recomendações para Programas em Linguagem C

1 Introdução

Este documento é uma coletânea de regras e recomendações para programas escritos em linguagem C. Um boa parte deste documento é baseado no manual de estilo desenvolvido nos laboratórios da AT&T em Indian Hill, EUA. Este documento não cobre todos os casos. Portanto experiência e bom-senso devem ser utilizados nas situações imprevistas.

Neste documento **módulo** é uma unidade de programa. O módulo é composto por um arquivo fonte (ou módulo de implementação ou *source file*) e por um arquivo de interface (ou arquivo cabeçalho ou módulo de interface ou *header file*).

2 Objetivo

O objetivo deste documento é definir um estilo de programação para a linguagem C a ser seguido pelos diversos autores de maneira que todos os artefatos criados tenham formato e aparência uniformes, independentemente de quem os produziu. Em última instância, este documento espera contribuir positivamente na qualidade do produto gerado, aumentar a sua portabilidade, reduzir a sua manutenção e melhorar a claridade do código gerado.

3 Arquivos

Os arquivos devem ser organizados em seções claramente marcadas. Embora não exista um tamanho máximo para arquivos, evite aqueles com mais de

1000 linhas porque eles são incômodos de manusear. De maneira similar, limite o comprimento das linhas a 100 colunas. Isto permite que uma linha possa ser impressa ou vista na tela sem transbordar para a próxima. Além disto linhas excessivamente longas resultantes de vários níveis de indentação são normalmente sinal de um programa pobremente organizado.

3.1 Nomeclatura de Arquivos

Os nomes dos arquivos compõem-se por um nome base, um ponto e um sufixo (ou extensão) na forma `base.ext`. O primeiro caracter do nome deve ser uma letra. Todos os caracteres (exceto o ponto) devem ser ser letras minúsculas ou números. O nome base deve ter até 8 caracteres e deve dar uma idéia da funcionalidade do arquivo. O sufixo deve ter até 3 caracteres e seguir a seguinte convenção:

- `c` para arquivos fonte em linguagem C;
- `cc` ou `cpp` para arquivos fonte em linguagem C++;
- `s` para arquivos fonte em linguagem Assembly;
- `h` para arquivos de interface em linguagem C; e
- `h` ou `hpp` para arquivos de interface em linguagem C++.

Não utilize nomes que já existam na biblioteca do compilador (por exemplo, `stdio.h`).

3.2 Arquivo Fonte

Organize os arquivos fontes nas seções detalhadas a seguir.

- **Cabeçalho do arquivo.** O primeiro ítem do arquivo fonte deve ser um bloco de comentários com a mensagem dos direitos autorais, a identificação do nome do arquivo, a descrição da funcionalidade que o código fornece (com referências caso apropriado) e uma lista ordenada das alterações que o arquivo sofreu.
- **Comandos do pré-processador.** Esta seção apresenta uma lista dos arquivos que são incluídos neste arquivo fonte (utilizando o comando

`#include`). A ordem dos arquivos deve ser: *(i)* arquivos de interface do sistema (da biblioteca do compilador), *(ii)* arquivo de interface deste módulo e *(iii)* demais arquivos. Caso uma inclusão aconteça por uma razão especial (e não óbvia), coloque esta razão num comentário. Em seguida coloque as definições de constantes e macros (utilizando o comando `#define`) que são utilizados **somente neste módulo**. Constantes e macros com uso global devem ser definidos no arquivo de interface deste módulo.

- **Definição de tipos.** Esta seção contém definições de tipos de dados (utilizando o comando `#typedef`) que são utilizados **somente neste módulo**. Para definições com uso global, utilize o arquivo de interface deste módulo.

- **Declaração de variáveis.** Esta seção contém declarações de variáveis que são locais a este módulo e que (em geral) são compartilhadas por uma ou mais funções. O uso de variáveis globais (compartilhadas por vários módulos) não é permitido.

- **Protótipos das funções.** Protótipos para todas as funções que são locais a este módulo devem ser incluidas nesta seção. Os argumentos devem conter tanto o tipo de dado como o nome. Funções globais devem ter seus protótipos colocados no arquivo de interface deste módulo.

- **Funções.** As funções devem ser colocadas por último no arquivo. A seqüência das funções fica a critério do programador; entretanto a sugestão é colocar funções na ordem em que elas são chamadas a partir da principal função do módulo, isto é, agrupando-se funções com o mesmo nível de abstração. Caso o módulo contenha a função `main` coloque-a obrigatóriamente primeiro. Para módulos com funções essencialmente independentes considere colocá-las em ordem alfabética. Cada função deve ser precedida de um cabeçalho onde detalhes do seu funcionamento são colocados.

A ordem das seções colocada acima deve ser seguida (veja o molde no Apêndice A). Nem todas as partes apresentadas precisam ser utilizadas num arquivo (por exemplo, um arquivo que não contenha definições de tipos deve ter esta seção vazia).

Todo arquivo fonte deve ter a ele associado um arquivo de interface com o mesmo nome base. Por exemplo o arquivo `foo.c` deve obrigatoriamente incluir o seu arquivo associado `foo.h`.

3.3 Arquivo de Interface

Arquivos de interface são aqueles incluídos em outros arquivos pelo pré-processador antes da compilação. Alguns, como por exemplo `stdio.h`, são definidos a nível de sistema e devem ser incluídos em qualquer programa que utilize, no caso deste exemplo, a biblioteca padrão de entrada/saída. Arquivos de interface contém declarações que são utilizadas por mais de um módulo. Estes arquivos devem ser organizados a nível de funcionalidade, isto é, declarações para cada sub-sistema devem ser agrupados em um ou mais arquivos. De maneira similar, declarações que são dependentes do ambiente de execução (*hardware*, sistema operacional, etc.) devem ser colocadas juntas em um conjunto de arquivos.

Não especifique toda a árvore de diretórios dos arquivos de interface incluídos pelo comando `#include`. É preferível agrupar estes arquivos em sub-diretórios e indicá-los ao compilador através da opção de *include-path* (normalmente `-I`). Isto evita que os arquivos fonte e de interface precisem ser modificados no caso de reorganização da estrutura dos diretórios.

Todos os arquivos de interface necessários para a compilação de um módulo devem ser diretamente incluídos. Evite, portanto, inclusões indiretas.

Todo arquivo de interface deve estar encapsulado pelos seguintes comandos:

```
#ifndef FOO_H
#define FOO_H
demais linhas do arquivo
#endif
```

onde `FOO_H` é o mome do arquivo em letras maísculas com o ponto substituido por um *underscore*. Isto evita que um arquivo seja incluído multiplas vezes.

É proibido declarar variáveis ou qualquer bloco de código em arquivos de interface.

Organize os arquivos de interface nas seções detalhadas a seguir.

- **Cabeçalho do arquivo.** Semelhante ao cabeçalho do arquivo fonte (veja Seção 3.2).
- **Comandos do pré-processador.** Esta seção apresenta uma lista dos arquivos que são incluídos neste arquivo (utilizando o comando `#include`). A ordem dos arquivos deve ser: *(i)* arquivos de interface do sistema (da biblioteca do compilador), *(ii)* demais arquivos. Caso uma inclusão aconteça por uma razão especial (e não óbvia), coloque esta razão num comentário. Em seguida são colocadas as definições de

constantes e macros (utilizando o comando `#define`) que são utilizados por um ou mais módulos.

- **Definição de tipos.** Esta seção contém definições de tipos de dados (utilizando o comando `#typedef`) que são utilizados por um ou mais módulos.

- **Protótipos das funções.** Protótipos para todas as funções utilizadas por um ou mais módulos devem ser incluidas nesta seção. Os argumentos devem conter tanto o tipo de dado como o nome.

A ordem das seções colocada acima deve ser mantida (veja o molde Apêndice B). Nem todas as partes apresentadas precisam ser utilizadas num arquivo (seções não utilizadas devem ser deixadas vazias).

4 Comentários

Utilize cometários para descrever o que está acontecendo, como está sendo feito, o que os parâmetros significam, as restrições e problemas da implementação, etc. Evite, entretanto, adicionar comentários quando a leitura do código já forneça informação suficiente. Comentários que estão em desacordo com o código tem um valor negativo. Atenção especial deve, portanto, ser tomada para a atualização dos comentários quando da mudança do código. Comentários curtos devem ser do tipo "calcula valor medio", ao invés de "soma dos valores dividido por n". Não há necessidade de colocar-se comentários em todas as linhas. Um comentário a cada bloco de 3 a 10 linhas de código explicando o seu funcionamento é normalmente suficiente.

Comentários de uma linha devem estar alinhados com o código a que eles se referem.

```
if( argc > 1 ) {
    /* Obtem arquivo de entrada a partir da linha de comando */
    if( freopen( argv[1], "r", stdin ) == NULL ) {
        perror( argv[1] );
    }
}
```

Blocos de comentários devem seguir o formato dado abaixo e devem também alinhar-se ao código a eles associados.

```
 /* Aqui comeca um bloco de comentario
  * O comentario deve estar alinhado com codigo
```

```
 * a que ele se relaciona
 */
```

Comentários mais curtos podem aparecer na mesma linha do código que eles descrevem (comentários *in-linha*). Eles devem ser separados do código por espaços e caso vários destes comentários aparecerem em um bloco de código eles devem ser alinhados.

```
if( alvo == EXCEPTION ) {
    flag = TRUE;              /* caso de excecao */
}
else {
    flag = isprime( alvo );   /* somente se alvo for impar */
}
```

5 Declarações

Coloque uma declaração por linha e adicione um comentário *in-linha* caso o seu nome não forneça informação suficiente. Os tipos, nomes, valores e comentários devem estar alinhados.

Para tipos `struct`, `enum` e `union`, coloque um elemento por linha, com um comentário *in-linha* se apropriado. A chave de abertura ({) deve estar na primeira linha enquanto que a de fechamento (}) deve estar sozinha na última linha.

```
/* Tipos de barcos */
enum bt_t {
    KETCH = 1,
    YAWL,
    SLOOP,
    SQRIG,
    MOTOR
};

struct boat {
    int        wllength;   /* comprimento da linha de agua em m */
    enum bt_t  type;       /* tipo do barco */
    long       sailarea;   /* area da vela em mm2 */
};
```

Coloque o qualificador de ponteiro (*) sempre com o nome da variável e nunca com o tipo, isto é,

```
float *x_axis;
```

em vez de

```
float* x_axis;
```

Qualquer variável cujo valor inicial é importante deve ser explicitamente inicializada ou, se este for o caso, deve-se adicionar um comentário indicando que a inicialização automática para zero é esperada. Nunca use o inicializador vazio {}. Na inicialização de estruturas, os valores devem ser separados apropriadamente por chaves. No caso de inicialização de inteiros longos, coloque um `L` depois do valor, conforme ilustrado no exemplo abaixo.

```
int          x = 1;
char         *msg = "message";
struct boat  winner[] = {
    { 40, YAWL, 6000000L },
    { 28, MOTOR, 0L },
    { 0 }
};
```

O uso de variáveis globais é proibido. Variáveis compartilhadas por várias funções devem ser evitados mas podem ser usadas desde que todas as funções estejam contidas num único módulo. Estas variáveis devem ser declaradas como estáticas.

Os tipos mais importantes (e somente os mais importantes) devem ser colocados em evidência pela criação de tipos especiais. Isto facilita a leitura do código.

6 Declaração de Funções

Cada função deve ser precedida por um cabeçalho (um bloco de comentários) que forneça detalhes sobre o que a função faz e (caso não seja claro) como usá-la. Algorítimos e implementações não triviais também devem ser discutidos, assim como resultados esperados para o uso indevido da função. Entretanto evite a duplicação de informação — não repita o que possa ser facilmente obtido do código.

O tipo retornado pela função deve ser sempre declarado e colocado sozinho na linha. Use o tipo `void` para aquelas funções que não retornem nenhum valor.

O nome da função e a lista de parâmentros devem ser colocados sozinhos numa linha. A chave de abertura ({) deve estar na coluna 1 e também sozinha na linha. A lista de declarações deve estar um nível acima de indentação.

Variáveis que têm o mesmo significado em várias funções devem ter o mesmo nome. Por outro lado evite usar o mesmo nome para variáveis que têm aplicação distinta.

7 Espaços Brancos

Use espaços verticais e horizontais de maneira a facilitar a leitura do código. Indentação e espaçamento devem refletir a estrutura do programa. Por exemplo, deve existir pelo menos uma linha em branco entre blocos de código do programa. O passo da indentação é de 4 espaços brancos. É proibido o uso de tabulação.

Linhas longas, em particular com operadores condicionais, devem ser quebradas em linhas separadas, como ilustrado abaixo.

```
if( foo->next == NULL
    && totalcount < needed
    && needed <= MAX_ALLOT
    && server_active( current_input )) {
```

De maneira similar, *loops* mais complicados devem ser divididos em em linhas diferentes.

```
for( curr = *listp, trail = listp;
     curr != NULL;
     trail = &( curr->next ), curr = curr->next ) {
```

De modo geral quebre qualquer expressão mais complexa em várias linhas.

```
var_c = ( var_a == var_b )
    ?  var_d + func( var_a )
    :  func( var_b ) - var_d;
```

Não deve haver espaço branco entre palavras da linguagem (`if`, `else`, `for`,

etc.) e parenteses ou entre nomes de funções e parenteses. Deve-se, entretanto separar os parenteses e o texto por eles contidos com um espaço branco. Espaços brancos não podem ser utilizados entre o nome e o parenteses na definição de macros com parâmetros porque o pré-processador não conseguiria reconhecer a lista de argumentos.

8 Instrução Simples

Deve existir uma e somente uma instrução por linha.

O corpo vazio de uma instrução de *loop* deve estar sozinho numa linha com um comentário indicando esta situação.

```
while( *dest++ == *src++ )
    ;    /* Vazio */
```

Não use comparações implicitas. É melhor usar

```
if( func() != FAIL )
```

do que

```
if( func() )
```

mesmo que `FAIL` tenha valor 0 o que em C significa falso. Um teste explicito evitará problemas caso alguém resolva alterar o valor de `FAIL` de zero para, por exemplo, -1.

Embora inexistente em C, é bastante comum a criação do tipo booleano:

```
typedef int   bool;
#define FALSE   0
#define TRUE    1
```

ou

```
typedef enum {
    NO=0,
    YES
} bool;
```

Isto facilita enormemente a leitura do programa.

Mesmo com estas declarações não se deve testar um valor booleano com 1 (ou `TRUE`, etc.) porque a maioria das funções garante o retorno do valor zero para falso mas somente um valor diferente de zero para verdadeiro. Assim

```
if( func() == TRUE ) {
```

deve ser escrito

```
if( func() != FALSE ) {
```

Evite instruções embutidas. Por exemplo

```
var_a = var_b + var_c;
var_d = var_a + var_r;
```

não deve ser substituido por

```
var_d = ( var_a = var_b + var_c ) + var_r;
```

embora este último economize um ciclo.

A instrução `goto` deve ser usada raramente, como em qualquer código bem estruturado. Seu principal uso é para sair de vários níveis de `switch`, `for`, etc. (embora isto indique que os blocos mais internos estariam melhores como funções separadas com os seus vereditos de sucesso/falha).

```
for( ... )  {
    while( ... )  {
     ...
    if( disaster != FALSE ) {
        goto error;
    }
}
...
error:
    trate condição de erro
```

9 Instrução Composta

Uma instrução composta é formada por uma lista de instruções simples envoltas entre chaves. A chave de abertura ({) deve ser colocada na primeira linha da instrução enquanto que a de fechamento (}) deve estar sozinha na última linha.

```
controle {
   instrução
   instrução
}
```

Para blocos com vários rótulos, cada um deve ser colocado em uma linha separada. A situação de *fall through* da instrução `switch` (isto é, quando não existe a instrução `break` entre um segmento do código e o próximo) deve ser comentada.

```
switch( expr ) {
case ABC:
case DEF:
    instruções
    break;
case UVW:
    instruções
    /* Fall through */
case XYZ:
    instruções
    break;
}
```

A última instrução `break` embora desnecessária deve ser colocada para evitar uma situação de *fall through* caso um novo `case` seja adicionado no futuro. O rótulo `default`, se presente, deve ser o último da instrução. Este rótulo não requer `break`.

Instruções `if-else` devem sempre usar as chaves para delimitar seus blocos de código, mesmo que o bloco contenha apenas uma única instrução.

```
if( expr ) {
    instrução
}
else {
    instrução
    instrução
}
```

Instruções `if-else` com continuação na forma de `else if` devem ser agrupadas conforme colocado abaixo.

```
if( STREQ( reply, "yes")) {
    instruções para yes
}
else if( STREQ( reply, "no")) {
    ...
}
else if( STREQ( reply, "maybe")) {
    ...
}
else {
    instruções para nda
    ...
}
```

Instruções `do-while` devem, de maneira similar, sempre usar as chaves para delimitar seus blocos de código, mesmo que o bloco contenha apenas uma única instrução.

10 Operadores

Separe operadores unários e seus operandos por pelo menos um espaço branco. Devido à que regras de precedência não serem sempres evidentes, deve-se envolver as operações em parenteses. Entretanto um número muito grande de parenteses dificulta a leitura e portanto bom senso deve ser usado. Evite o operador binário virgula. Quando testar uma variável com ponto flutuante, use sempre `>=` ou `<=` e nunca `==` ou `!=`.

11 Nomeclatura

Nomes com um *underscore* como primeiro ou último caracter são comumente utilizados pelo sistema e devem ser evitados pelo programador. Constantes (definidas por `#define` e `enum`) devem conter somente letras maiúsculas. Por outro lado nomes de funções, variáveis, novos tipos (usando `typedef`), estruturas (`struct`), uniões (`union`) e enumerações (`enum`) devem conter somente letras minusculas. Nomes de macros deve conter somente letras maiúsculas.

Evite nomes que diferem apenas na caixa da letra (maiúsculo ou minusculo), por exemplo `foo` e `Foo`. Evite também nomes que se paracem, como `foobar` e `foo_bar` ou aqueles que diferem apenas pelos caracteres `l` (letra ele), `1` (número um) e `I` (letra i). Estes casos tem um grande potencial para confusão.

Nomes de variáveis **nunca** devem ter apenas uma letra, com exceção dos populares contadores de *loop* `i`, `j` e `k`, desde que eles não desempenhem nenhuma função adicional.

12 Constantes

Constantes numéricas não devem ser usadas diretamente. O comando do pré-processador `#define` deve ser empregado para dar às constantes nomes

com sentido. Isto facilita a leitura do código além de simplificar no caso do valor de alguma constante mudar, uma vez que apenas o respectivo `#define` necessita ser alterado.

Para variáveis que adquirem um número finito (e de certo modo pequeno) de valores, é aconselhável usar o tipo enumeração.

13 Macros

Expressões complexas podem ser usadas como parâmetros em macros e problemas de precedência podem ocorrer caso todas as ocorrências dos parâmetros na definição da macro não estejam envoltas por parenteses.

Macros economizam o processamento adicional causado pela chamada da função. Entretanto quando a macro fica muito longa esta desvantagem torna-se desprezível e preferência deve ser dada para o uso de uma função.

14 Depuração

A primeira constante de enumerações deve ter o seu valor explicitamente declarado.

Verifique sempre o veredito de sucesso/falha das funções chamadas. Isto permite que o programa seja abortado de maneira ordenada e limpa caso algum erro aconteça.

Código inserido para depuração e teste do módulo deve ser mantido para fins de manutenção e compilado condicionalmente.

15 Compilação Condicional

Compilação condicional é útil para código dependente do ambiente de execução, para manutenção, etc. Limite preferencialmente a compilação condicional aos arquivos de interface. Isto mantém os arquivos fontes mais limpos e facilita a sua leitura. Por exemplo, a função de alocação de memória pode ser definida como

```
#ifdef DEBUG
    extern void *mm_malloc();
```

```
        #define MALLOC( size )   ( mm_malloc( size ))
    #else
        extern void *malloc();
        #define MALLOC( size )   ( malloc( size ))
    #endif
```

de maneira que no código apenas `MALLOC` é usado.

16 ANSI C

Sempre que possível escreva código seguindo o padrão ANSI C. Isto melhora a eficiência e beneficia a portabilidade.

A Molde para Arquivos Fonte

```
/*********************************************************************************************
 *
 * Departamento de Infomatica
 * Centro de Tecnologia
 * Universidade Estadual de Maringa
 *
 * Copyright (c) 2006 Universidade Estadual de Maringa
 * Todos os direitos reservados. Nenhuma parte deste arquivo pode ser usada ou
 * reproduzida sem permissao escrita da Universidade Estadual de Maringa.
 *
 * Nome do arquivo: foo.c
 * Descricao: descrição em até 2 linhas
 */

/********************************** LISTA DE MODIFICACOES ************************************
 *
 * aaaa-mm-dd    Nome             Descrição da modificação mais recente.
 * 2003-12-19   Joao Autor       Corrigido problema com alavanca azul.
 * 2003-12-01   Joao Autor       Arquivo criado.
 */

/*********************************************** INCLUDES ***********************************************/

#include cheader2.h
#include foo.h
#include other.h

/*********************************************** CONSTANTES *********************************************/

#define CONSTANTE_1        10
#define CONSTANTE_2        20

/************************************************ MACROS ************************************************/

#define MACRO_1            (instruções)
#define MACRO_2            (instruções)

/********************************************* TIPOS DE DADOS *******************************************/
```

```
#typedef oldname_1         newname_1
#typedef oldname_2         newname_2

/************************************* VARIAVEIS LOCAIS ************************************/

static tipo_1  var_1;
static tipo_2  var_2;

/********************************** PROTOTIPOS DAS FUNCOES **********************************/

static return_t function1( t11 name11, t12 name12);
static return_t function2( t21 name21);

/*************************************************************************************************
 *
 * nome da função
 *
 * parametros: lista de parâmetros com explicação do uso e faixa de valores esperados+
 *
 * valor retornado: significado do valor retornado
 *
 * descricao: descrição da função
 *
 * notas: explicação de particularidades da função, por exemplo limitacoes e comportamento
 *         em caso de parâmetros fora da faixa, etc.+
 */

int
fct1( int var1 )
{
    instruções
}
```

B Molde para Arquivos de Interface

```
#infdef FOO_H
#define FOO_H
```

```
/*****************************************************************************************************
 *
 * Departamento de Infomatica
 * Centro de Tecnologia
 * Universidade Estadual de Maringa
 *
 * Copyright (c) 2006 Universidade Estadual de Maringa
 * Todos os direitos reservados. Nenhuma parte deste arquivo pode ser usada ou
 * reproduzida sem permissao escrita da Universidade Estadual de Maringa.
 *
 * Nome do arquivo: foo.h
 * Descricao: descrição em até 2 linhas
 */

/************************************ LISTA DE MODIFICACOES ************************************
 *
 * aaaa-mm-dd    Nome             Descrição da modificação mais recente.
 * 2003-12-19    Joao Autor       Corrigido problema com alavanca azul.
 * 2003-12-01    Joao Autor       Arquivo criado.
 */

/*************************************** INCLUDES ***************************************/

#include cheader1.h

/*************************************** CONSTANTES ***************************************/

#define CONSTANTE_1        10
#define CONSTANTE_2        20

/*************************************** MACROS ***************************************/

#define MACRO_1            (instruções)
#define MACRO_2            (instruções)

/************************************ TIPOS DE DADOS ************************************/

#typedef oldname_1         newname_1
#typedef oldname_2         newname_2

/************************************ PROTOTIPOS DAS FUNCOES ************************************/
```

```
static return_t function1( t11 name11, t12 name12);

#endif
```