PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 26/96

PG-03 Regras e Recomendações para a Programação em C e C++

- Versão 1.01 -

Arndt von Staa
(Editor)

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 26/96
Editor: Carlos J. P. Lucena Setembro, 1996

PG-03 Regras e Recomendações para a Programação em C e C++ *

- Versão 1.01 -

Arndt von Staa
(Editor)

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca Documentação e Informação
PUC-Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 - Rio de Janeiro, RJ
Brasil
Tel +55-21-529 9386 Fax +55-21-511 5645
E-mail: biblio@inf.puc-rio.br www: http://www.inf.puc-rio.br

PG-03 Regras e recomendações para a programação em C e C++

Versão 1.01

editor: A.v. Staa[1]
arndt@inf.puc-rio.br

Laboratório de Engenharia de Software
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro,
Brasil

Setembro 1996

PUC-RioInf.MCC 26/96

Resumo

Neste documento estabelecemos regras e recomendações para o estilo de redação de programas redigidos em C. Estas mesmas regras de estilo se aplicam a programas redigidos em C++, considerando a sub-linguagem desta e que se assemelha a C. O estilo procura evitar uso de construções de difícil compreensão, bem como visa uniformizar a apresentação dos programas.

Palavras chave: estilo de programação, formatação de programas, manutenibilidade.

Abstract

In this document we establish style and formatting rules for programs written in C. These rules apply also to the sub-language of C++ similar to C.

Keywords: maintainability, programming style, source code formatting.

---

[1] Trabalho apoiado por: CNPq, Bolsa de Pesquisador 300029/92-6, CENPES/Petrobrás, Itautec/ Philco

Histórico de evolução

PG-03 Regras e recomendações para a programação em C e C++

Gestor: Laboratório de Engenharia de Software
Departamento de Informática, PUC-Rio

Arquivo: Nrc01
Editado: 16 setembro, 1996
Impresso: 16 setembro, 1996

Documentos correlatos

PG-01 Regras e recomendações para a escolha de nomes de elementos em programas C e C++

PG-02 Regras e recomendações para o uso de constantes simbólicas em C e C++

PG-04 Regras e recomendações específicas para a programação em C++

PG-05 Regras e recomendações para a inclusão de especificações

Versão V1.01

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/jul/1996
Data entrada em vigor: 01/jul/1996

Data de início da próxima revisão 02/jan/1997

Descrição de evolução
acertos de ortografia e sintaxe
eliminação de pequenas ambigüidades

Descrição da retroação

Versão V1.00

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/mar/1996
Data entrada em vigor: 01/mar/1996

Descrição de evolução

Descrição da retroação

Créditos

Revisores

André Derraik	(TeCGraf PUC-Rio)	Versão 1.0
Claudio de Oliveira	(PUC-PR)	Versão 1.0
Geraldo Machado Costa	(LES PUC-Rio)	Versão 1.0
Lincoln Nobumiti Kanamori	(Itautec Philco)	Versão 1.0
Pedro Alexandre O. Giovani	(Itautec Philco)	Versão 1.0
Pedro Jorge E. Hübscher	(LES PUC-Rio)	Versão 1.0
Renan Martins Baptista	(CENPES)	Versão 1.0
Rosa Maria Ramalho Correia	(Itautec Philco)	Versão 1.0

Apoio

CENPES Petrobrás
CNPq
Itautec Philco

Marcas registradas e nomes de produtos

MS-DOS, Windows, Visual C++ são marcas registradas da Microsoft Corp.

Sumário

1. Objetivo 1
2. Motivação 2
3. Definição da norma 3
    3.1 Uso da linguagem padrão 3
    3.2 Composição de um módulo 4
    3.3 Comentários 7
    3.4 Margem esquerda 9
    3.5 Estilo de expressões 10
    3.6 Estilo de declarações 11
    3.7 Dados globais 12
    3.8 Ponteiros 13
    3.9 Estilo de estruturas de controle 14
    3.10 Estilo de redação de comandos sucessivos 16
Bibliografia 18

1. Objetivo

O presente conjunto de regras e recomendações tem como objetivos:

- definir um estilo de programação válido para programas redigidos em C ou em C++.
- conduzir a programas que facilitem verificar se estão corretos.
- conduzir a programas de fácil manutenção.

Como resultado da adoção desta norma espera-se que:

- todos os programas tenham um estilo consistente e independente de quem redigiu o programa.
- seja fácil entender corretamente o código, sem precisar recorrer a documentação complementar.
- sejam evitada a ocorrência de erros comuns ao programar em C ou C++.
- programadores possam rápida e facilmente entender e corretamente modificar programas redigidos por outros.

A presente norma é válida para programas escritos em C ou em C++. Na norma *PG-04 Regras e recomendações específicas para a programação em C++* encontra-se um outro conjunto de regras visando especificamente a linguagem C++. Esta divisão foi feita para simplificar o texto da presente norma que se dirige também a programas redigidos em C.

2. Motivação

C e C++ formam um par de linguagens em que C++ contém quase integralmente todas as construções e respectiva semântica da linguagem C. Originalmente, inclusive, a linguagem C++ era traduzida para C por meio de um conjunto de macros do pré-processador C. Ambas as linguagens C e C++ permitem construções que, se usadas de forma desavisada, podem trazer dificuldades ao desenvolvimento, à garantia de qualidade, e à manutenção do programa.

Como conseqüência da adoção da presente norma, espera-se uma redução significativa da ocorrência dos erros de programação mais comuns cometidos ao se programar em C ou em C++ sem uma disciplina bem definida. Além de procurarem atingir o ideal de programas corretos por construção, as regras e recomendações aqui apresentadas estabelecem um estilo de programação uniforme. Sendo as linguagens C e C++ muito ricas e versáteis, este conjunto de regras visa evitar que programas sejam implementados de uma forma críptica, dificultando a sua compreensão e manutenção. Mais especificamente, as presentes regras visam eliminar personalismos danosos à qualidade de programas, facilitando o desenvolvimento e a manutenção de um mesmo programas por várias pessoas e em épocas bem distintas.

Módulos de programas devem poder ter vida longa. Além disso, módulos devem poder ser reutilizados em diversos projetos, reduzindo substantivamente o volume de código a ser redigido e mantido. Embora a meta de reúso requeira, em primeiro lugar, especificações e projetos cuidadosamente elaborados, para que módulos sejam reutilizáveis é necessário que sejam portáteis. Ou seja, é necessário que possam ser compilados, sem necessitarem de maiores modificações, com uma variedade de compiladores e executando em uma variedade de plataformas, produzindo exatamente o mesmo comportamento.

3. Definição da norma

Neste documento, quando não for explicitamente dito ao contrário, as regras, exceções e recomendações valerão tanto para programas escritos em C, como para programas escritos em C++. Se uma regra se aplica somente a uma destas linguagens, este fato será explicitamente mencionado.

3.1 Uso da linguagem padrão

Regra 1: Ao programar em C utilize somente construções válidas no padrão ANSI.

Exceção 2: Sendo necessário o uso de uma palavra chave não padronizada, utilize uma constante simbólica no seu lugar. Estas constantes simbólicas devem estar contidas no arquivo PDR<id compilador>.INC.

Recom. 3: Ao programar em C++ utilize somente construções válidas no padrão ANSI em elaboração.

Recom. 4: Todas as construções não padronizadas devem ser agregadas em um único arquivo de definições.

A linguagem C é padronizada. Para assegurar portatilidade entre diferentes compiladores, em particular, com relação a compiladores futuros, evite o uso de construções que não sejam do padrão ANSI. Utilize sempre as chaves de compilação que assegurem a conformidade do programa com o padrão ANSI.

Embora C++ ainda não possua um padrão aprovado, já existe um padrão ANSI sendo desenvolvido. Procure ater-se a este padrão. Ou seja, compile programas C++ com as chaves de conformidade com o padrão ANSI ligadas e siga as observações do parágrafo anterior com relação a possíveis necessidades de não conformidade com o padrão ANSI.

Infelizmente, pode ser necessário o uso de construções não padronizadas, entre outras pelas razões a seguir:

- para estabelecer comunicação com alguns ambientes – por exemplo *Windows* adota a convenção *Pascal* de passagem de parâmetros obrigando a inclusão da palavra chave *pascal* não existente em outras plataformas;
- para assegurar o correto uso das propriedades de endereçamento da máquina objetivo – por exemplo as arquiteturas *Intel*, quando utilizadas em modo de 16 bits, diferenciam ponteiros `far` de ponteiros `near`.

Nos casos em que uma construção não padronizada for necessária, utilize um comando `#define` para definir uma constante simbólica ou uma macro resultando na construção desejada. Os comandos `#define` que estabelecem padrões próprios, devem ser agregados em um único arquivo específico. Dependendo do compilador e da plataforma utilizada, o conteúdo deste arquivo pode variar. Utilize um nome de arquivo na forma `PDR<id compilador>.INC`, criados para cada compilador – `<id compilador>` – a ser utilizado. Através deste artifício o texto do programa permanece o mesmo, independentemente do compilador empregado. O que varia é o arquivo de definições que assegura a uniformidade entre compiladores.

Regra 5: Utilize a chave de controle de advertências mais restritiva disponível no compilador e corrija o código até que não sejam mais geradas advertências ao compilar.

Exceção 6: Caso a advertência seja de otimização ou seja gerada por falha do compilador, e após um cuidadoso exame, algumas advertências poderão ser toleradas.

Recom. 7: Mantenha a última listagem de mensagens de erro junto com os respectivos módulos de implementação e de definição.

C e C++ permitem a realização de diversas operações com resultados inesperados. Por exemplo, em ambas é permitido atribuir um valor do tipo ponteiro a uma variável do tipo inteiro e vice-versa. Na maioria das vezes o compilador adverte quando uma construção levando a resultados duvidosos for utilizada. Assegurando-se que tais advertências sejam eliminadas, reduz-se em muito o número de ocasiões em que resultados inesperados poderão ser produzidos durante a execução do programa.

Infelizmente, alguns compiladores emitem advertências relativas a código correto. Evidentemente, tais advertências nem sempre poderão ser eliminadas. Muitas vezes compiladores emitem advertências relativas a condi-

ções de otimização. Por exemplo, ao otimizar o código de um `case` contendo um comando `break` ao final de um `switch` que, por sua vez, se encontra ao final da função. Neste caso recebe-se tipicamente uma mensagem do gênero "código inatingível". Alguns compiladores também emitem advertências para construções do tipo `while ( TRUE )` – repetir para sempre –, informando que a "expressão de controle do `while` é constante". Todos estes exemplos ilustram casos em que mensagens de advertência são geradas relativas a código, em princípio, correto. No entanto, não se pode aceitar tacitamente mensagens de advertência, uma vez que podem refletir mal uso da linguagem. Por exemplo mensagens do gênero "variável pode estar sendo utilizada antes de ser inicializada" é freqüentemente um sinal de erro de programação. Para permitir aos inspetores de qualidade, examinar se as mensagens geradas são permitidas, mantenha sempre a última listagem de mensagens de erros de compilação junto com o respectivo código. Para tal crie um arquivo com o nome *`xxx.err`*, onde *`xxx`* é o nome do arquivo contendo o código do módulo e nele coloque as mensagens de erro da última compilação deste módulo.

3.2 Composição de um módulo

Regra 8: Cada módulo será composto por um módulo de definição e um módulo de implementação.

Uma das propriedades de C e C++ é permitir o controle de consistência das interfaces entre módulos. Para assegurar que este controle *garanta* a consistência entre módulos cliente[2] e módulos servidores, é fundamental que se utilize exatamente a mesma definição de interface tanto ao compilar o módulo servidor, como ao compilar os diversos módulos cliente deste módulo servidor. O código de interface será redigido no *módulo de definição*. Este módulo será utilizado – `#include` – ao compilar o *módulo de implementação* do servidor – *módulo de definição próprio* –, bem como ao compilar outros módulos de implementação de módulos cliente. Surgem problemas devido a propriedades sintáticas das linguagens C e C++, por exemplo, variáveis externas devem aparecer exatamente uma vez sem o declarador `extern`, e todas as outras vezes com este declarador. Regras estabelecidas mais adiante descrevem como resolver estas dificuldades.

Regra 9: Utilize os seguintes nomes de extensão:

1. `C` para módulos de implementação escritos em C
2. `H` para módulos de definição escritos em C
3. `CPP` para módulos de implementação escritos em C++
4. `HPP` para módulos de definição escritos em C++
5. `INC` para arquivos contendo tabelas de constantes simbólicas
6. `HH` para arquivos contendo módulos de definição visíveis a programas externos ao projeto. Estes arquivos de definição contém, tipicamente, as definições de APIs tornadas disponíveis.

Para facilitar a criação de ferramentas, e para tornar uniforme a nomenclatura de arquivos, é recomendável que se adote um padrão de nomes de extensão de arquivos. Desta forma, conhecendo o nome de extensão, é conhecido também a natureza do conteúdo do arquivo.

Regra 10: Módulos definição e arquivos de tabelas devem conter controles para evitar a inclusão duplicada ao compilar um módulo, utilizando o esquema de código a seguir:

```
<Cabeçalho do arquivo>

#if ! defined( <Nome arquivo>_ )
#define        <Nome arquivo>_

<Corpo do arquivo>

#endif
```

onde:

---

[2] *Módulos cliente* utilizam dados, tipos, classes e funções que outros módulos – *módulos servidores* – tornam disponíveis.

<Cabeçalho de arquivo> é um comentário padrão identificando o arquivo, ver norma *PG-05 Regras e recomendações para a inclusão de especificações*

<Nome arquivo> é o nome do arquivo contendo o módulo de definição ou a tabela de constantes. Note que o nome do arquivo será seguido de um caractere *sublinha* para assegurar a unicidade do nome. Este nome será definido e sua definição nunca será excluída.

<Corpo do arquivo> é o código do arquivo de definição ou da tabela de constantes. Caso seja necessário, inicie o texto do corpo do módulo de definição com comandos de inclusão. Para tal utilize a mesma ordem que a descrita na regra 16.

Regra 11: Módulos de definição devem poder ser utilizados para compilar tanto o próprio módulo cuja interface definem, como para compilar módulos cliente deste.

Regra 12: Inicie o código do corpo do módulo de definição com o seguinte esquema de código:

```
// Controle de escopo do arquivo de definição[3]

#if defined( <Nome arquivo>_PROPRIO )
   #define <Nome arquivo>_CLASS
#else
   #define <Nome arquivo>_CLASS extern
#endif
```

A variável de compilação terminada em `_PROPRIO` é utilizada para sinalizar se o módulo de definição está sendo compilado junto com o próprio módulo de implementação ou com algum dos seus clientes. A variável de compilação terminada em `_CLASS` controla a presença do declarador `extern`. Este será gerado sempre que o módulo de definição for incluído ao compilar um módulo cliente. Todas as inclusões de arquivos devem ser colocadas após este código de controle.

Regra 13: Declare da seguinte forma cada variável global externa:

```
<Nome arquivo>_CLASS <declaração de variável externa>
```

Regra 14: Declare a inicialização de variáveis externas, utilizando código com a organização a seguir:

```
#if defined( <Nome arquivo>_CLASS )
          = <Inicialização> ;
#else

#endif
```

Regra 15: Ao final do módulo de definição coloque o código:

```
#undef <Nome arquivo>_CLASS
```

As linhas finais eliminam os controles de geração de código de módulos de definição próprio. A Figura 1 ilustra um módulo de definição fictício redigido em C e observando estas regras.

---

[3] Nesta norma adotaremos a convenção C++ para comentários. Para comentários C substitua o par de caracteres `//` pelo par `/*` e inclua o par `*/` ao final da última linha de comentário de um mesmo grupo de linhas de comentário.

```
//***************************************************************
//
//  Módulo de definição: Módulo exemplo
//
//  Nome do arquivo:     EXEMP.HPP
//
//***************************************************************

#if !defined( EXEMP_ )
#define EXEMP_

// Controle de escopo do arquivo de definição

#if defined( EXEMP_PROPRIO )
   #define EXEMP_CLASS
#else
   #define EXEMP_CLASS extern
#endif

//**** Estruturas de dados exportada pelo módulo *****

   // Estrutura de dados : Vetor de números

      EXEMP_CLASS int EX_vtNum[ 5 ]
                #if defined( EXEMP_PROPRIO )
                   = { 1 , 2 , 3 , 4 , 5 } ;
                #else
                   ;
                #endif

#undef EXEMP_CLASS

#endif

//********* Fim da definição: modulo **********
```

Figura 1. Ilustração de módulo de definição

Regra 16: Organize o módulo de implementação e o de definição na seguinte forma:

1. cabeçalho do módulo
2. inclusão, se necessário, do arquivo `PDR<id compilador>.INC`. Este arquivo contém as declarações não padronizadas segundo o padrão ANSI.
3. inclusões do compilador
4. inclusão do módulo de definição próprio. Em programas de um único módulo e que externam exclusivamente a função `main`, o módulo de definição não é requerido.
5. inclusões, se necessário, dos módulos de definição dos módulos servidores
6. inclusões, se necessário, dos arquivos de tabelas de constantes
7. corpo do módulo de implementação ou de definição

Regra 17: Redija o código de inclusão do módulo de definição próprio na forma a seguir:

```
#define <Nome arquivo>_PROPRIO
#include "ARQUIVO.H"
#undef  <Nome arquivo>_PROPRIO
```

Esta regra assegura que um módulo de definição redigido em conformidade com as regras anteriores, gere código correto ao ser compilado no módulo de implementação próprio.

Regra 18: Módulos de definição devem conter exclusivamente as declarações e o código executável (tipicamente `inline`) efetivamente necessários para que módulos cliente possam ser compilados.

Regra 19: Todas as variáveis e funções declaradas em módulos de implementação devem estar precedidas do declarador `static`.

O módulo de definição deve ser o menor possível. Para isto deve conter somente as declarações de tipos, de constantes, de variáveis, de classes, de protótipos de funções e, no caso de C++, de código `inline` que precise ser tornado visível para potenciais módulos cliente. Todas as demais declarações e todo código não `inline` devem ser encapsuladas, devendo estar contidos exclusivamente no módulo de implementação. Este, por sua vez, não deve declarar nenhuma variável ou função global externa. Para tal, todas as declarações de variáveis globais encapsuladas e de protótipos de funções encapsulados devem ser `static`.

3.3 Comentários

Regra 20: Ao programar em C, utilize somente comentários permanentes do tipo `/* ... */`.

Regra 21: Ao programar em C++ utilize somente `//` para comentários permanentes.

Regra 22: Ao programar em C ou em C++, utilize o esquema de código a seguir para tornar temporariamente não compilável uma parte de um programa:

```
#ifdef 0
   ...
#endif
```

O padrão de comentários C é da forma `/* ... */`, não existindo outra forma válida para comentários. Em C++ pode-se adotar tanto a forma `//` ... como a forma `/* ... */`, sugere-se adotar uma única destas formas, no caso a forma mais comumente encontrada.

Como comentários na forma `/* ... */` não podem ser aninhados, precisa-se utilizar uma construção de compilação condicional como a definida na Regra 22 para temporariamente eliminar código durante o desenvolvimento, teste ou integração. Uma vez completado e aprovado o módulo, este não deverá mais conter construções de compilação condicional como as da Regra 22.

Regra 23: Utilize a estrutura a seguir para delimitar porções de código condicionalmente compiláveis:

```
#ifdef <Identificação da janela de visita>
   ...
#endif
```

Em muitas ocasiões é interessante inserir código permanente que somente será compilado quando o programa estiver sendo depurado. Este código deve permanecer como código de compilação condicional. Utiliza-se para tal a construção `#ifdef` apresentada na Regra 23. Esta construção vale tanto para C como para C++.

A *<Identificação da janela de visita>* é uma variável de pré-processamento, usualmente `_DEBUG`. Definindo-se várias variáveis de pré-processamento, pode-se estabelecer diversos graus de granularidade nos instrumentos de apoio à depuração. A seguir ilustramos o uso desta construção:

```
#ifdef   _DEBUG
   #undef THIS_FILE
   static char THIS_FILE[] = __FILE__ ;
#endif
```

Este código condicional cria a constante *string* `THIS_FILE` contendo o nome do arquivo de código fonte sendo compilado. Isto permite que os instrumentos de apoio à depuração emitam mensagens identificando o arquivo contendo o código do instrumento de tempo de execução acionado.

```
1  // Extrair palavras chave
2     // Criar lista de palavras chave
         fragmento de código

3     // ptOrgLista - aponta para lista vazia

4     // Gerar lista de palavras chave
5        // Ler primeiro caractere
            fragmento de código

6        while ( tem caractere )
7        {
8           // Processar caractere corrente
9              if ( Caractere delimitador
                 && Palavra chave vazia   )
10             {
11                // Registrar início de nova palavra chave
                     fragmento de código
12             } else if ( Caractere de palavra )
13             {
14                // Adicionar caractere à palavra
                     fragmento de código
15             } else if ( Caractere delimitador
                        && Palavra chave não vazia )
16             {
17                // Registrar a palavra chave
                     fragmento de código
18             } // if

19          // Ler próximo caractere
               fragmento de código
20       } // while

21    // ptOrgLista - aponta para o primeiro elemento da lista
22    //      - os elementos da lista são duplamente encadeados
23    //      - o primeiro elemento possui antecessor nulo
24    //      - o último elemento possui sucessor nulo

25    // Ordenar a lista de palavras chave
         fragmento de código

26    // Exibir a lista de palavras chave
         fragmento de código
```

Figura 2. Exemplo de comentários, utilizando as convenções de C++

Recom. 24: Utilize comentários para:

- prover informação gerencial.
- documentar especificações.
- definir a intenção do código a seguir. Comentários desta natureza são também chamados de *pseudo-código*.
- definir as condições que devem ser satisfeitas pelos valores manipulados pelo programa, para que este esteja operando corretamente no ponto onde se encontra este comentário. Comentários desta natureza são chamados de *assertivas*.
- definir marcas de conclusão de estruturas de controle de programas.

Recom. 25: Comentários devem ser compactos e fáceis de encontrar e distinguir do código.

Recom. 26: Comentários que não acrescentem informação devem ser eliminados.

Comentários devem acrescentar informação que não se conseguiria extrair facilmente do código a seguir. Comentários óbvios ou não elucidativos de nada adiantam e, portanto, devem ser eliminados. A norma *PG-05 Regras e recomendações para a inclusão de especificações* detalha o formato, a natureza do conteúdo e a aplicação dos diversos comentários.

Em trechos de código fonte pequenos – tipicamente entre 5 e 10 linhas de código – comentários são geralmente supérfluos. A escolha criteriosa de nomes de elementos é muito mais útil nestes casos. A norma *PG-01 Regras e recomendações para a escolha dos nomes de elementos em programas C e C++* estabelece regras e recomendações para a escolha de bons nomes. Trechos maiores devem ser segmentados, sendo cada segmento precedido por um comentário – *pseudo instrução* – que identifica a intenção do trecho de código a seguir. Pseudo-instruções devem sempre ser redigidas iniciando com um verbo no infinitivo, uma vez que elas correspondem, em linhas gerais, a funções. As pseudo-instruções formam uma estrutura de endentação. Na Figura 2 apresentamos um esqueleto de programa. Neste esqueleto os textos redigidos em *itálico* devem ser substituídos por código fonte em acordo com a especificação contida no comentário que antecede o texto. Na figura as linhas 1, 2, 4, 5, 8, 11, 14, 17, 19, 25 e 26 ilustram pseudo-instruções. Note que a margem esquerda de pseudo-instruções de nível de abstração menor está 3 caracteres para dentro a medida que se vai descendo no nível de abstração, exemplos linhas 1, 2, 4, 25 e 26, onde as quatro últimas formam um nível de abstração menor do que a linha 1. Operações de controle concretas e que estão no mesmo nível de abstração que uma pseudo-instrução são alinhadas na mesma margem esquerda que as correspondentes pseudo-instruções. As linhas 5, e 6 ilustram isto. Todos fragmentos de código são alinhados **3 caracteres para a direita** com relação à margem da correspondente pseudo-instrução. As linhas 8, 9 e 10 ilustram isto. Finalmente, pseudo instruções contidas em uma estrutura de controle estarão 3 caracteres para a direita da margem de alinhamento da correspondente estrutura de controle. As linhas 6, 7 e 8, e linhas 9, 10 e 11 ilustram isto.

O término de estruturas de controle deve ser claramente marcado. As linhas 18 e 20 ilustram comentários de término de estruturas de controle. Nas linguagens C e C++ o caractere delimitador fecha chave "}" é utilizado universalmente para terminar uma função, uma repetição, uma seleção, ou, como acontece em C++, um bloco contendo uma declaração local própria. É comum perder-se a contagem de fecha chave. Embora existam ferramentas para determinar o correto nivelamento de parêntesis, colchetes e chaves, estas ferramentas não informam o significado do fecha chave. Em adição o caractere fecha chave é excessivamente discreto, tornando-o difícil de distinguir do caractere fecha parêntesis, podendo ser facilmente ignorado em uma leitura apressada.

Assertivas devem ser alinhadas na mesma margem em que ficaria a correspondente pseudo-instrução. As linhas 3 e 21 a 24 ilustram assertivas.

Recomenda-se, ainda, que pseudo-instruções e assertivas sejam separadas do código que as antecede de pelo menos uma linha em branco. Similarmente o código que as sucede deve ser precedido de um branco. Desta forma os componentes lógicos de um algoritmo ficam mais destacados. Linhas em branco devem ser incluídas também nos fragmentos de código de modo que fiquem mais evidentes os grupos de ações que formam unidades lógicas.

3.4 Margem esquerda

Regra 27: Ao endentar para a direita, avance sempre 3 caracteres.

É habitual registrar a estrutura de controle e de abstração do código fonte através do alinhamento da margem esquerda. Endentações de menos do que 3 caracteres tendem a passar despercebidas. Já endentações com mais do que 3 caracteres tendem a rapidamente esgotar o espaço útil de uma linha de código na tela.

Regra 28: Linhas de continuação devem estar 10 caracteres para a direita da linha inicial do comando, sendo que a nova linha deve iniciar sempre com um operando.

Exceção 29: Parâmetros de função redigidos em linhas sucessivas, devem alinhar-se na margem esquerda do primeiro parâmetro da função. Caso a expressão do parâmetro precise ser continuada, a continuação estará 4 caracteres à direita da margem dos parâmetros.

Ao utilizar expressões complexas envolvendo elementos com nomes longos, é comum esgotar-se a linha antes de se conseguir completar a redação da expressão. As linhas continuadas devem terminar com um operador sinalizando para o leitor que a linha a seguir é uma continuação. A escolha de 10 caracteres visa criar uma margem esquerda de alinhamento da continuação evidentemente diferente da margem esquerda da linha inicial sendo continuada e de possíveis blocos endentados logo a seguir. Caso a expressão requeira mais de uma linha de continuação todas as linhas continuadas deverão ser alinhadas na mesma margem esquerda.

Exemplo:

```
VariavelMuitoLonga = OutraVariavelLonga->CampoLongo[
          IndiceLongo ].OutroCampoLongo ;
```

Expressões de parâmetros de funções podem ser longas obrigando seqüências de parâmetros a serem colocados em linhas sucessivas. Nestes casos o texto fica mais legível se cada parâmetro for colocado em uma linha e estas alinhadas com o primeiro parâmetro. Caso o espaço útil na linha fique reduzido, alinhe o primeiro parâmetro em linha de continuação.

Exemplos

```
FuncaoLongaComTresParametros( EsteEhOPrimeiroParametro ,
                              EsteEhOSegundo ,
                              VariavelLonga->CampoLongo[
                                      IndiceLongo ].OutroCampo ) ;
```

Forma alternativa de redigir a mesma expressão:

```
FuncaoLongaComTresParametros(
          EsteEhOPrimeiroParametro ,
          EsteEhOSegundo ,
          VariavelLonga->CampoLongo[
                    IndiceLongo ].OutroCampo ) ;
```

Regra 30: Utilize caracteres espaço em branco e não tabulações para alinhar a margem esquerda.

Nem todos os editores entendem tabuladores, isto pode provocar inconsistências se um código fonte é intercambiado entre diferentes editores. Além disso, cada editor tem as suas convenção de estabelecer paradas de tabulação, tornando difícil assegurar a uniformidade do tratamento de margens esquerdas. Pior, como paradas de tabulação podem ser alteradas pelo usuário, um código com a organização de margens perfeita pode perder esta propriedade por ser editado com um editor com paradas de tabulação diferentes.

3.5 Estilo de expressões

Regra 31: Separe todos os operadores, sinais de pontuação e operandos do elemento precedente por um espaço em branco. No entanto, não separe:

1. os operadores '.' e '->',
2. vírgulas do operando que a antecede,
3. indicadores de tipo '*' e '&' do nome do tipo,
4. operadores unários dos respectivos operandos,
5. abre parêntesis de funções,
6. abre colchetes de indexações.

Da mesma forma como em português se espera ver as palavras nitidamente separadas, também em programas é fortemente recomendável que os elementos que formam uma expressão sejam bem destacados um do outro.

Tanto em C como em C++ a propriedade de ser um ponteiro ou uma referência é uma propriedade do tipo e não da variável. Conseqüentemente o caractere "*" que denota que uma variável é um ponteiro para um determinado tipo, é propriedade deste tipo e não da variável. Note que esta regra contraria o costume, comum ao redigir programas C, de escrever o indicador de tipo ponteiro associado ao nome da variável.

Exemplos

```
Var0 = Var1 + Var2 * (( Var3 + Var4 ) * 5 ) ;
Funcao( -Vet[ Var6->Var7.Campo, &OutroParm ] ) ;
Valor = OutraFunc( Parm, OutroParm + 1, TerceiroParm ) ;
tpObjeto* pObjeto ;
int*      pInteiro ;
```

Confronte as duas últimas declarações com a prática desrecomendada e ilustrada a seguir:

```
tpObjeto *pObjeto ;
int      *pInteiro ;
```

3.6 Estilo de declarações

Regra 32: Declare cada variável independentemente, exceto quando o tipo da variável for um tipo primitivo não ponteiro ou referência.

Regra 33: Liste as declarações em linhas sucessivas.

Regra 34: Alinhe o nome de variáveis declaradas em comandos sucessivos na mesma margem esquerda após a definição do tipo.

Regra 35: Alinhe inicializações de modo que os nome de variáveis declaradas permaneçam destacados no texto.

Exemplos

```
// Não declare assim:
   int i , j , k ;

// Declare assim:
   int i ,
       j ,
       k ;

// Não declare assim:
   int* pA ,
      * pB ,
      * pC ;

// Declare assim:
   int* pA ;
   int* pB ;
   int* pC ;

// Não declare assim:
   char *NomeUsuario = 0 ;
   int NumLivros=42;
   int & IntRef = NumLivros ;

// Declare assim - observe o alinhamento do operador "=":
   char*  NomeUsuario = 0 ;
   int    NumLivros   = 42 ;
   int&   IntRef      = NumLivros ;
```

Regra 36: Declare o tipo retornado e o nome da função na mesma linha.

Regra 37: Não insira espaço em branco entre o nome da função e o correspondente abre parêntesis.

Regra 38: Declare parâmetros formais de uma função em linhas sucessivas alinhadas com o primeiro parâmetro e obedecendo à regra de alinhamento do nome após ao declarador. Alinhe as vírgulas separadoras de parâmetros formais na mesma margem, e coloque o fecha parêntesis da função um caractere para a direita do alinhamento das vírgulas.

Regra 39: Declare a lista de parâmetros em protótipos de funções exatamente igual à lista nos cabeçalhos correspondentes.

Regra 40: Ao programar em C use `void` como tipo de função que retorne nada, e/ou caso a lista de parâmetros formais for vazia. Ao programar em C++ utilize `void` para denotar um valor retornado de tipo indefinido.

Exemplos

```
// Não declare assim:
   char*
   Objeto::sString( )
   Outro( int Parm, char* Str )
   MaisUm( int )
   main( )
```

```
// Declare assim:
   char*  Objeto::sString( void ) ;
   void   Outro( int   Parm ,
                 char* Str    )
   void   MaisUm( int Nome )
   void   main( void )
```

Regra 41: Em C++ variáveis devem ter o menor escopo possível.

Em C++ variáveis locais podem ser declaradas no interior de blocos. Utilize esta possibilidade para declarar variáveis no menor bloco possível. Isto reduz possíveis interferências entre usos de variáveis.

Recom. 42: Evite obliterar o nome de variáveis globais com variáveis locais.

Procure dar nomes diferentes a variáveis globais, a variáveis membro de classes e a variáveis locais. Muitas vezes, ao encontrar o nome de uma variável global em uma função, assume-se tacitamente que a função está acessando esta variável global e não a variável local de mesmo nome. Esta recomendação não se aplica quando o nome local for o nome de um parâmetro, uma vez que neste caso o protótipo conterá este nome documentando explicitamente a obliteração de nomes. Adotando a norma *PG-01 Regras e recomendações para a escolha dos nomes de elementos em programas C e C++* a possibilidade de obliteração de nomes é virtualmente eliminada.

Regra 43: Cada variável declarada deve receber um valor inicial antes de ser utilizada.

Recom. 44: Sempre que for possível, utilize inicialização e não atribuição de valor para inicializar variáveis declaradas.

Exceção 45: Quando o valor inicial de uma variável for uma expressão complexa ou o resultado de um processamento extenso, a atribuição inicial é desnecessária.

Um dos erros freqüentes é o uso de variáveis que ainda não foram devidamente inicializadas. Assegure-se, por inspeção do código, que todas as variáveis tenham sido devidamente inicializadas. Pode-se inicializá-las para um valor neutro, por exemplo zero ou nulo, ou para uma constante com o significado de *valor ilegal*. Neste último caso, antes de se utilizar a variável inclui-se um instrumento de depuração, por exemplo:

```
ASSERT Variavel != VALOR_ILEGAL ;
```

verificando se a variável contém este valor ilegal. Ao compilar o programa para produção este instrumento é automaticamente ignorado.

A inicialização na declaração tende a ser mais eficiente do que a inicialização por sucessivas atribuições.

Recom. 46: Use tipos `unsigned` somente para declarar variáveis que jamais poderão ter valores negativos.

Evite operações de subtração envolvendo valores de tipo *unsigned*. Um valor *unsigned* "negativo" é sempre maior do que os valores inteiros positivos. Muitas vezes isto provoca resultados inesperados. O seguinte `for` entra em *loop* infinito caso a variável de controle tenha um tipo *unsigned*:

```
for ( UnSig = 10 ; UnSig >= 0 ; UnSig--)
```

Identificadores são exemplos de valores que nunca terão valores negativos, uma vez que são tipicamente gerados por meio de operações de adição. Já coordenadas, mesmo quando confinadas a uma janela podem, temporariamente, ter valores negativos quando a expressão de cálculo da coordenada for complexa.

3.7 Dados globais

Recom. 47: Evite o uso de dados globais externos.

Recom. 48: Ao programar em C++, evite o uso de dados globais encapsulados.

Recom. 49: Sendo necessário declarar dados globais, agregue-os em uma estrutura e declare uma única variável com o tipo desta estrutura.

Dados globais devem ser utilizados com parcimônia, principalmente ao programar em C++. Algumas vezes, no entanto, pode-se tornar necessário utilizar dados globais. Dados globais devem ser agregados em uma ou mais estruturas dependendo do seu significado, se são coletâneas de dados de interface ou se são coletâneas de parâmetros de controle da estação de trabalho. O uso de dados agregados facilita a cópia dos dados no caso de se precisar preservá-los por alguma razão. Por exemplo, ao salvar ou recuperar os parâmetros de controle da estação de trabalho.

O exemplo a seguir ilustra a definição de dados externos globais em C:

```
/* Declaração do tipo estrutura de dados interface */
   typedef struct
   {
      int  Var1 ;
      char Var2 ;
   } MOD_tpInterface ;

/* Declaração da variável de interface */
   MOD_tpInterface MOD ;
```

Note que o nome da variável de interface é somente o nome do produto ou domínio – ver norma *PG-01 Regras e recomendações para a escolha de nomes de elementos em programas C e C++*. Desta forma um uso da variável global se assemelha ao nome que esta variável teria se fosse redigido normalmente, por exemplo `MOD.Var1`, ao invés de `MOD_Var1` como normalmente seria redigida.

O exemplo a seguir ilustra a definição de dados globais encapsulados em C:

```
/* Declaração do tipo estrutura de dados encapsulada */
   typedef struct
   {
      int  Var1 ;
      char Var2 ;
   } st_tpCapsula ;

/* Declaração da variável de interface encapsulada */
   st_tpCapsula st ;
```

3.8 Ponteiros

Regra 50: Ao programar em C, utilize a constante `NULL` para denotar ponteiros nulos.

Regra 51: Ao programar em C++, utilize a constante `P_NIL`, definido igual a (void *) 0, para denotar ponteiros nulos .

Em C++ ainda não está definido como será denominado e declarado o valor de ponteiro nulo, poderá ser definido `(void *) 0` ou simplesmente `0`. Por esta razão não deve ser utilizado o valor `NULL` em C++. No entanto, recomenda-se o uso do valor 0 para ponteiros nulos em C++. O valor `P_NIL` deve ser definido, no arquivo `PDR<id compilador>.INC`. Este arquivo já havia sido definido na seção *Uso da linguagem padrão*. A definição será na forma:

```
#define   P_NIL  (( void * ) 0 )
```

Recom. 52: Evite a atribuição de ponteiro a outro ponteiro.

Nem C nem C++ possuem *garbage collection*. Quando um espaço de dados for liberado, não é perguntado se existe algum outro ponteiro ativo apontando para ele. Isto pode levar a acessos a espaços de dados já liberados e realocados para outro uso, bem como a desalocações múltiplas de um mesmo espaço de dados, provocando erros de execução de difícil localização.

Recom. 53: Evite o uso de ponteiros para ponteiros.

Recom. 54: Em C++ parâmetros de funções que alterem o valor de um parâmetro do tipo ponteiro devem ser declarados como referência para ponteiro (ex. char* &).

Em caso de necessidade de ponteiros para ponteiros em C++, deve ser declarada uma classe que possua uma variável membro do tipo ponteiro. Isto aumenta a inteligibilidade do código e, conseqüentemente, a facilidade de avaliar se está ou não correto. Em alguns casos não é possível observar esta recomendação. Por exemplo em ambas as linguagens, o segundo parâmetro da função `main( )` deve ser declarado `char* [ ]`, ou seja é, na realidade um valor do tipo `char**`, ponteiro para ponteiro.

Recom. 55: Evite o uso de aritmética envolvendo ponteiros.

Ponteiros dependem do modelo de endereçamento da máquina para a qual o programa está sendo compilado. O comportamento de aritmética envolvendo ponteiros pode se tornar dependente da máquina e portanto não ser portátil, nem entre diferentes modelos de endereçamento de uma mesma máquina. Por outro lado, operações que adicionem (ou subtraiam) valores inteiros de ponteiros podem ser utilizados uma vez que possuem uma semântica bem definida e independente de plataforma.

Exemplo:

```
   char* pChr1 ;
   char* pChr2 ;
   char  vtChr3[ 100 ] ;
   int   Desloc1 ;
   int   Desloc2 = 3 ;
   pChr1 = vtChr3 ;
   pChr2 = &vtChr3[ 10 ] ;
/* código a evitar */
   Desloc1 = pChr2 - pChr1 ;
/* Código permitido */
   *( pChr1 + Desloc2 ) = 'x' ;
```

Regra 56: Não utilize ponteiros como valores lógicos.

Embora ponteiros nulos tenham normalmente o valor 0, não redija expressões tais como

```
if ( pVal )          ou   if ( !pVal )
```

Ao invés disso redija, em C:

```
if ( pVal != NULL )     e   if ( pVal == NULL );
```

em C++:

```
if ( pVal != P_NIL )    e   if ( pVal == P_NIL ).
```

Recom. 57: Ao desalocar um espaço de dados, sempre atribua um novo valor a todos os ponteiros que apontam para este espaço de dados ou para o seu interior.

Recom. 58: Evite ter mais de um ponteiro para um mesmo espaço de dados.

Recom. 59: Evite o uso de ponteiros globais, procure utilizar somente ponteiros locais.

Ponteiros que apontam para áreas desalocadas devem receber valor `NULL` ou `P_NIL`, ou então um novo valor, para evitar o acesso futuro a esta área. Evidentemente, isto pode ser muito difícil de ser realizado, uma vez que nem sempre é possível saber quais são todos os ponteiros que apontam para um mesmo espaço de dados. Procurando fazer com que o escopo de vida de um ponteiro seja o menor possível, reduz-se em muito os problemas relacionados ao uso de ponteiros.

3.9 Estilo de estruturas de controle

Regra 60: Os caracteres que delimitam um bloco devem ser colocados na mesma coluna e em linhas separadas antes e depois do bloco, alinhados na margem esquerda do correspondente comando de controle.

Exemplo:

```
while ( ... )
```

```
{
   // código do bloco
} // while
```

Regra 61: Todos os comandos de controle devem ser seguidos de um bloco, mesmo que o código deste bloco seja vazio ou tenha uma única linha.

Exemplo

```
// Não escreva assim
   while ( ... ) ;

// Escreva assim
   while ( ... )
   {
      // corpo vazio
   } // while

// Não escreva assim
   if ( ... ) return ;

// Nem assim
   if ( ... )
      return ;

// Escreva assim
   if ( ... )
   {
      return ;
   } // if
```

Código é lido inúmeras vezes e por diferentes pessoas. Além disso, muitas vezes a leitura tende a ser realizada de forma dinâmica, na busca de um ponto contendo determinado bloco. Ao ler, mesmo que superficialmente, deve-se ter sempre certeza de estar entendendo o texto. É recomendável, então, que todos os blocos corpo de alguma estrutura de controle sejam redigidos da mesma forma. Blocos corpo de uma única linha, ou até vazios devem estar visíveis. Em particular, blocos corpo vazios devem deixar claro que estão intencionalmente e não acidentalmente vazios.

Regra 62: Separe as palavras reservadas de estruturas de controle do abre parêntesis da correspondente expressão por pelo menos um espaço em branco.

Os comandos `if`, `while`, `for` e `switch` não são funções, portanto não devem seguir a notação comum a funções.

Exemplos:

```
if ( A != 0 )
for ( i = 0 ; i < n ; i++ )
```

Regra 63: Sempre termine o bloco de comandos que segue um `case` com um comando `break`.

Utilize cases sucessivos sem break somente para selecionar um mesmo fragmento de código. Cada seleção deve terminar com um `break` ou `return` para evitar a execução do código do fragmento a seguir.

Exemplo:

```
case 'd' :
case 't' :
   printf( "Consoantes palatais" ) ;
   break ;
```

Regra 64: Sempre inclua uma opção `default` nas estruturas de `switch`.

Regra 65: Em seqüências de seleção múltipla criadas com sucessivos comandos `if else if`, assegure que o último `else` exista e que capture todas as condições imprevistas.

switches com corpos longos, ou seqüências complexas de comandos `if else if`, são difíceis de entender, tornando freqüente a ocorrência de erros de omissão de condições. Isto ocorre particularmente quando se realiza manutenção em um módulo. Capturando e notificando como erros de execução todos os casos e condições que não deveriam ocorrer, reduz a chance destes erros passarem despercebidos por inspeções e testes.

Recom. 66: Não crie variáveis temporárias apenas para controle de término de um ciclo, use `break` ou `return` para sair de ciclos antes de processar todos os elementos.

Recom. 67: Evite o uso de `continue`.

Regra 68: É proibido o uso de `goto`.

Muitos livros texto que abordam programação estruturada de forma ortodoxa proíbem qualquer forma velada ou explícita de `goto`, entre eles o `break` e o `return`. Para tal, torna-se necessário criar variáveis temporárias do gênero `Terminou`. De maneira geral o código se torna tão pouco legível e certamente menos eficiente do que o código contendo `break` e `return`. Já a necessidade de se utilizar `goto` é sempre fruto de um projeto inadequado, conseqüentemente pode ser evitado sempre.

Regra 69: Expressões lógicas envolvendo operadores `||` e `&&` devem ser quebradas em linhas sucessivas a cada operador lógico. As linhas sucessivas devem ter a margem esquerda alinhada nos operadores lógicos. Cada sub-expressão deve ser inserida em um par de parêntesis, exceto quando a expressão for unária ou de um único termo. Os caracteres abre e fecha parêntesis de um mesmo nível de operação devem estar alinhados na mesma margem.

Recom. 70: Evite expressões lógicas complexas. Procure particioná-las em vários `if`'s aninhados.

Exemplo:

```
while ( ( ( a < b    )
        || ( x != 'x' ))
     && !Acabou        )
```

3.10 Estilo de redação de comandos sucessivos

Recom. 71: Ponha legibilidade e compreensibilidade antes das demonstrações de domínio de nuanças da linguagem.

Recom. 72: Somente otimize código através de artifícios de programação se for estritamente necessário e, ainda assim, somente depois de efetivamente medir se este fragmento de código é o responsável pelo desempenho inaceitável.

Programas extensos, independentemente de quão bem forem escritos, são difíceis de compreender. O uso liberal da linguagem somente aumenta esta dificuldade. Raras vezes código sub-ótimo poderá ser melhorado significativamente através de floreios de uso da linguagem. Além disso, os modernos compiladores otimizadores tendem a gerar código de excelente qualidade. De maneira geral código com desempenho inaceitável é fruto de um projeto inadequado, ou da escolha de estruturas de dados inapropriadas ao problema, ou da escolha de algoritmos de baixo desempenho. Antes de tentar otimizar através do uso de técnicas de programação desrecomendáveis ou de difícil compreensão, meça e verifique onde se encontram os segmentos de código responsáveis pelo consumo de recursos.

Regra 73: Cada comando deve iniciar em uma nova linha.

Regra 74: Expressões não devem conter atribuições[4].

[4] C e C++ trata atribuição ('=') como um operador binário que retorna o valor atribuído. Sendo assim, uma atribuição não pode ser classificada como um comando. No entanto, conforme estabelecido por este conjunto de regras e recomendações, a atribuição é restringida de tal modo que se passa a ter as características de um comando. O objetivo é evitar que sejam redigidas atribuições escondidas no meio de uma expressão complexa. Tais atribuições têm o mesmo caráter nocivo de um efeito colateral.

Da mesma forma como cada variável deve ter um somente significado, cada instrução deve servir a um só propósito. Através do uso de variáveis temporárias, pode-se eliminar usos de atribuições no meio de expressões.

Exemplos:

```
// Evite
   vtAbc[ i++ ] = 0 ;
// Redija
   vtAbc[ i ] = 0 ;
   i++ ;

// Evite
   if ( i -= X )
// Redija
   i -= X
   if ( i != 0 )
```

Recom. 75: Redija comandos de atribuição sucessivos alinhando o operador de atribuição na mesma margem.

Recom. 76: Sempre que possível agrupe comandos de atribuição sucessivos segundo o seu significado.

Procure sempre destacar os elementos de uma expressão de modo a tornar mais legível o programa. Um forte aliado para o aumento da legibilidade é a diagramação do texto de código.

Exemplos

```
// Não redija assim
   Var1 = 0 ;
   Var3 = 'x' ;
   vtVar2[ Var1 ].Campo2 = 1 ;


// Redija assim
   Var1                  = 0 ;
   vtVar2[ Var1 ].Campo2 = 1 ;
   Var3                  = 'x' ;
```

Regra 77: Redija atribuições de um mesmo valor a diversas variáveis em forma de uma lista de linhas, cada qual contendo uma atribuição.

Lembre-se que um programa se torna mais legível se cada linha contiver apenas um comando. Exemplo.

```
// Não redija assim
   Var1 = Var2 = Var3 = 0 ;

// Redija assim
   Var1 =
   Var2 =
   Var3 = 0 ;
```

Bibliografia

Diversas das regras aqui apresentadas, bem como a estrutura das presentes regras, seguem de perto o texto:

Henricson, M; Nyquist, E; *Programming in C++: Rules and Recomendations*; Relatório técnico; Ellemtel Telecommunications Systems Laboratories; Älvsjö, Suécia; 1992

A referência padrão para a linguagem C é:

Kernighan, B.; Ritchie, D.; *The C Programming Language, Second Edition*; Prentice Hall 1988

A referência padrão para a linguagem C++ é:

Stroustrup, B.; *The C++ Programming Language*, Second Edition; Addison Wesley; 1991

Uma boa referência para orientação a objetos utilizando C++ é

Montenegro, F.; Pacheco, R.; *Orientação a Objetos em C++*; Editora Ciência Moderna; 1994