Apêndice 1. Padrão geral

O presente padrão tem por objetivo:

- reduzir a freqüência das faltas e defeitos mais corriqueiros.

1.1 Uso da linguagem padrão

Recom. 1: Ponha legibilidade e compreensibilidade antes das demonstrações de domínio de nuanças da linguagem.

Recom. 2: Somente otimize código através de artifícios de programação se for estritamente necessário e, ainda assim, somente depois de ter medido o desempenho dos elementos do programa.

Regra 3: Ao programar em C utilize somente construções válidas no padrão ISO [C 1990].

Regra 4: Ao programar em C++ utilize somente construções válidas no padrão ISO [C++ 1998, Stroustrup 1997].

Recom. 5: Sendo necessário o uso de uma palavra chave não padronizada em C ou C++ e requerida pelo ambiente de desenvolvimento, utilize uma constante simbólica no seu lugar. Estas constantes simbólicas devem estar contidas no arquivo `PDR<`*id compilador*`>.INC`.

Recom. 6: Todas as palavras não padronizadas devem ser agregadas em um único arquivo de definições.

Recom. 7: Ao programar em Java utilize a versão 1.2 ou mais recente [HC 1999a, HC 1999b][*].

Regra 8: Utilize a chave de controle de mensagens de advertência mais restritiva disponível no compilador e corrija o código até que não sejam mais geradas advertências ao compilar.

Exceção 9: Caso a advertência seja decorrente de um uso justificado de uma construção problemática, ou decorrente de otimização aceitável, ou gerada por falha conhecida do compilador, inclua um comentário no código fonte, no local indicado pela mensagem de advertência, informando que a mensagem está justificada e a razão da justificativa.

[*] Caso surja uma nova versão após março de 1999, procure utilizar esta versão.

Recom. 10: Ao desenvolver um programa utilizando um Ambiente Integrado de Programação (IDE – Integrated Dvelopment Environment, ex.: Turbo C/C++, Visual C/C++, Visual Cafe), realize o conjunto completo de testes com relação ao programa executável gerado no final. Os testes de aceitação devem ser realizados fora do controle do ambiente de desenvolvimento.

Recom. 11: Mantenha a última listagem de mensagens de erro e advertências junto com os respectivos módulos de implementação e de definição.

1.2 Declaração de protótipos e de cabeçalhos de funções

Regra 12: Declare a lista de parâmetros de um protótipo de função contendo os mesmos nomes que os contidos na lista de parâmetros do correspondente cabeçalho.

Regra 13: Ao programar em C use `void` como tipo de função que retorne nada, e quando a lista de parâmetros formais for vazia.

Regra 14: Ao programar em C++ utilize `void` para denotar um valor retornado de tipo indefinido.

1.3 Declarações

Recom. 15: Em C++ e Java utilize blocos aninhados para declarar variáveis locais de modo que tenham o menor escopo possível.

Recom. 16: Evite obliterar com nomes de variáveis locais os nomes de elementos globais ou de membros de classes.

Recom. 17: Sempre que for possível, declare e inicialize as variáveis em um mesmo comando.

Exceção 18: Em C e C++ não é necessário inicializar variáveis globais se o valor inicial for 0.

Recom. 19: Caso uma variável deva receber um valor calculado em uma função antes de ser utilizada, inicialize-a com um valor constante não permitido no conjunto de valores legais e que denote *variável não inicializada*.

Exemplo

```
#define   VALOR_ILEGAL   0xFFFF
...
tpAlgumTipo ptAlgumaArea = VALOR_ILEGAL ;
```

Recom. 20: Inclua imediatamente antes do primeiro uso de uma variável inicializada com a constante valor ilegal um dos fragmentos de código:

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

ou

```
if ( Variavel == VALOR_ILEGAL )
{
   ativar exceção de erro de uso da variável
} // if
```

Recom. 21: Use tipos `unsigned` somente para declarar variáveis que jamais poderão ter valores negativos, mesmo durante a avaliação de uma expressão envolvendo valores deste tipo.

Exemplo:

```
// Evite:
   unsigned Conta;
   for ( Conta = 10; Conta >= 0; Conta--)  // nunca termina

// Redija assim:
   int Conta;
   for ( Conta = 10; Conta >= 0; Conta--)
```

1.4 Dados globais

Recom. 22: Evite o uso de dados globais externos.

Recom. 23: Sendo necessário declarar dados globais, agregue-os em um tipo estrutura e declare uma única variável com este tipo.

Exemplo

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

```
/* Declaração da interface */
   MD_tpInterface MD ;       /* O acesso aos elementos terá a
                             /* forma MD.Var1 */

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

/* Declaração da cápsula */
   st_tpCapsula st ;       /* O acesso aos elementos terá a
                           /* forma st.Var1 */
```

1.5 Seqüência de execução

Recom. 24: Somente ative código capaz de alterar o conteúdo de estruturas de dados:

- Depois que todos os dados necessários tiverem sido adquiridos e devidamente validados.
- Depois que todos os espaços de dados necessários tiverem sido alocados.
- Sem interferência do usuário ou da plataforma de execução.

Exemplo

Considere o problema: substituir o elemento corrente existente em uma lista de elementos por um novo elemento `<chave, valor>` onde `valor` é um ponteiro para uma área de dados.

```
// Evite uma estrutura semelhante a:
   alocar espaço para cabeça (parte chave) do elemento novo
   obter chave do elemento novo
   copiar chave para cabeça de elemento novo
   validar chave do elemento novo
   eliminar cabeça e valor do elemento antigo
   inserir cabeça do elemento novo na lista no lugar do
             elemento antigo
   alocar espaço para valor do elemento novo
   obter valor do elemento novo
   copiar valor para espaço alocado
   validar valor do elemento novo
   encadear cabeça com valor do elemento novo
```

Esta estrutura mistura as ações. Ao realizar uma manutenção podem surgir problemas, pois as instruções não formam grupos coesos. A estrutura a seguir é bem melhor, pois organiza o código de acordo com a semântica dos grupos de instruções:

```
// Criar nó do valor do elemento novo a inserir
   obter valor do elemento novo
   validar valor do elemento novo
   alocar nó com espaço para valor do elemento novo
   copiar valor para nó
// Criar cabeça do elemento novo
   obter chave do elemento novo
   validar chave do elemento novo
   alocar espaço para cabeça do elemento novo
   copiar chave para cabeça
   encadear cabeça com valor do elemento novo
// Substituir elemento antigo pelo novo
   inserir cabeça do elemento novo na lista no lugar do
             elemento antigo
   eliminar cabeça do elemento antigo
   eliminar valor do elemento antigo
```

Recom. 25: Agregue e ordene seqüências de comandos ou pseudo-instruções de acordo com o seu significado.

Exemplo:

```
// Seqüência a evitar
   Gravar BufferA em Saída
   Gravar BufferB em Saída
   Ler registro de ArquivoB para BufferB
   Ler registro de ArquivoA para BufferA

// Prefira a seguinte seqüência
   // Transferir A para Saída
      Gravar BufferA em Saída
      Ler registro de ArquivoA para BufferA
   // Transferir B para Saída
      Gravar BufferB em Saída
      Ler registro de ArquivoB para BufferB
```

A possibilidade de identificar novas pseudo-instruções é particularmente interessante pois facilita encontrar agregados que possam ser reutilizados. No exemplo acima, pode-se criar uma função que grava o *buffer* e lê um novo valor. Após esta transformação, o texto fica assim:

```
// Transferir A e B para Saída
      Transferir( ArquivoA, BufferA, ArquivoSaida) ;
      Transferir( ArquivoB, BufferB, ArquivoSaida) ;

// A função Transferir seria semelhante a:
   void Transferir( FILE * ArquivoEntra ,
                    char * BufferEntra  ,
                    FILE * ArquivoSai    )
{
   Gravar BufferEntra em ArquivoSai
   Ler registro de ArquivoEntra para BufferEntra
```

```
}
```

Recom. 26: Seqüências de comandos parentetisantes devem ser redigidos obedecendo ao correspondente aninhamento.

Exemplo:

```
// Ordenação de código a evitar
   Abrir arquivo A
   Abrir arquivo B
   Abrir arquivo C
      ...
   Fechar arquivo A
   Fechar arquivo B
   Fechar arquivo C

// Ordenação de código segundo a regra
   Abrir arquivo A
   Abrir arquivo B
   Abrir arquivo C
      ...
   Fechar arquivo C
   Fechar arquivo B
   Fechar arquivo A
```

📖 Operações **parentetisantes** são pares de operações onde uma desfaz o efeito da outra. São exemplos:

- *abrir* e *fechar* arquivos;
- *criar* e *destruir* arquivos;
- *alocar* e *desalocar* espaços de dados;
- *inserir* e *excluir* elementos em uma estrutura de dados;
- *construir* e *destruir* classes;

1.6 Expressões

Recom. 27: Evite o uso do operador ternário "`?`" quando pelo menos uma das expressões contiver mais de um operador, ao invés utilize o comando `if`.

Recom. 28: Sempre use `sizeof( NomeVariavel )` para determinar o tamanho ocupado por uma variável.

Recom. 29: Sempre use `sizeof( NomeDoTipo )` para determinar o tamanho a ser ocupado por um elemento a alocar (ex.: `malloc`),

ou para saber o tamanho do espaço de dados apontado por um ponteiro ou referência.

Recom. 30: Evite o uso de *type cast*. Use-o somente para determinar o tipo específico ao alocar um espaço de dados, ou ao copiar um valor contido em estrutura de persistência (ex.: arquivo) ou em estrutura genérica (ex.: lista).

Regra 31: Sempre verifique o retorno de uma função ou método que possa gerar uma exceção ou retornar uma condição de retorno.

Exemplo

```
// Evite:
   pStr = ( char * ) malloc( strlen( Str ) + 1 );
   strcpy( pStr, Str ) ;
   pArq = fopen( NomeArq, "w" );
   fputs( pStr, pArq );

// Redija:
   pStr = ( char * ) malloc( strlen( Str ) + 1 );
   if ( pStr == NULL )
   {
      printf( "\nFaltou memória" ) ;
      exit( 4 ) ;                    /* ou tratamento de erro */
   } /* if */
   strcpy( pStr, Str ) ;

   pArq = fopen( NomeArq, "w" );
   if ( pArq == NULL )
   {
      printf( "\nNão abriu: %", NomeArq ) ;
      exit( 4 ) ;                    /* ou tratamento de erro */
   } /* if */
   fputs( pStr, pArq );
```

```
// Evite:
   CFile Arq( NomeArq, CFile::modeCreate | CFile::modeWrite ) ;
   Arq.Write( Buffer, sizeof( Buffer )) ;

// Redija:
   CFile Arq ;
   if ( !Arq.Open( NomeArq, CFile::modeCreate | CFile::modeWrite ))
   {
      out << "\nNão abriu " << NomeArq << " em virtude de "
                << expArq.m_cause ;
      tratamento de erro
   } // if
   Arq.Write( Buffer, sizeof( Buffer )) ;
```

1.7 Comandos

Regra 32: É proibido o uso de `goto`.

1.7.1. switch e case

Recom. 33: Mantenha curto o código associado a cada `case` (em torno de 5 linhas). Se o código ficar longo, converta-o em uma chamada de função.

Regra 34: Sempre termine o bloco de comandos que segue um `case` com um comando `break`.

Exemplo:

```
// Evite, pois torna o código sensível à arrumação
   case 1 :
      algum código sem break no final
   case 2 :
      mais código sem break no final

// Redija:
   case 1 :
      algum código
      break ;
   case 2 :
      mais código
      break ;

// Outro exemplo, redija assim
   case 'd' :
   case 't' :
      printf( "Consoantes palatais" ) ;
      break ;
```

Regra 35: Sempre inclua uma opção `default` nas estruturas de `switch`.

Recom 36 O `default` do `switch` deve capturar somente as condições não previstas.

Exemplo

```
// Evite:
   switch( menuItem )
   {
      case ITEM_COPY:
         Copiar( );
         break;
      case ITEM_CUT:
```

```
         Cortar( ) ;
         break;
      default:            // Captura tudo: colar e todos os erros.
         Colar( );
         break;
   } // switch

// Redija assim:
   switch( menuItem )
   {
      case ITEM_COPY:
         Copiar( );
         break;
      case ITEM_CUT:
         Cortar( ) ;
         break;
      case ITEM_PASTE:
         Colar( ) ;
         break;
      default:
         ASSERT( FALSE );     // Sempre corresponde a um erro.
         break;
      } // switch
```

```
// Exemplo de default que efetivamente corresponde a "demais casos"
   switch( NivelAcesso )
   {
      case 0:
         InserirAdministrador( );
         break;
      case 1:
         InserirConsultor( );
         break;
      case 4:
         InserirInstalador( );
         break;
      default:                //compreende 2, 3, 5, 6, ..., 100
         InserirUsuarioNormal( );
         break;
   } // switch
```

1.7.2. if else

Regra 37: Em seleções múltiplas heterogêneas criadas com sucessivos comandos `if ... else if`, assegure que o último `else` exista.

Recom 38 O `else` final de uma seleção heterogênea deve capturar somente as condições não previstas.

Recom 39 Assegure que o conjunto das condições de uma seleção heterogênea, exceto o `else` final, cobre a totalidade de condições possíveis.

Exemplo

```
if        ( cond1 )
{
   // fragmento de código selecionado se cond1 for verdadeira
} else if ( cond2 )
{
   // fragmento de código selecionado se cond1 for falsa
   //      e cond2 for verdadeira
} else if ( cond3 )
{...
} else if ( condn )
{
   // fragmento de código selecionado se cond1 até condn-1 forem
   //      todas falsas e condn for verdadeira
} else {
   // fragmento de código selecionado se cond1 até condn forem
   //      todas falsas, condição default
} // if
```

No esquema de código acima o i-ésimo fragmento de código é selecionado de acordo com a seguinte condição:

$$\left(\&\&_{j=1}^{i-1} \; !cond_j \right) \;\&\&\; cond_i \quad , \quad i \text{ é mínimo}$$

Cada elemento selecionável da seleção heterogênea redigida desta forma depende da posição em que ocorre. A condição "*i é mínimo*" é necessária, uma vez que diversas condições poderiam resultar em verdadeiro e será sempre escolhida a primeira delas na seqüência de execução.

Recom 40 Procure utilizar sempre a seleção independente de posição.

Recom 41 Procure assegurar que cada condição de uma seleção heterogênea, exceto o `else` final, seja mutuamente exclusiva com todas as demais condições.

Em uma seleção independente de posição vale a seguinte condição:

$$\left(\&\&_{j=1}^{n,\, j \neq i} \; !cond_j \right) \;\&\&\; cond_i$$

Esta condição valerá se e somente se todas as condições forem mutuamente exclusivas. Neste caso o resultado da seleção será independente da ordem com que as condições forem redigidos.

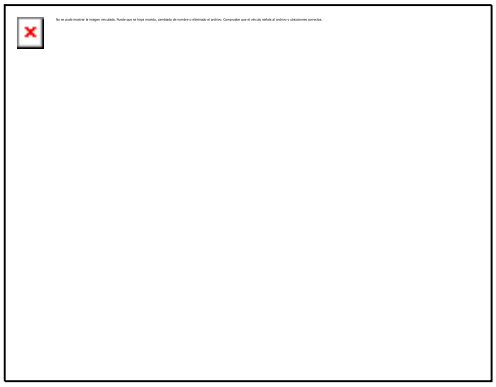

Figura 1. Ilustração de escopo de efeito, escopo de controle

O *escopo de controle* de uma condição (resultado da avaliação de uma expressão condicional) é a porção de código controlada por uma seleção (ou repetição) que avalia uma expressão resultando nesta condição. Na Figura 1 o escopo de controle da condição *Cond1* é a subestrutura *X* quando computada em *Controlar ação B*, e é a subestrutura *Z* quando calculada em *Controlar ação A*. O escopo de controle da condição *Cond2* é a subestrutura *Y* quando calculada em *Controlar ação B*.

O *escopo de efeito* de uma condição é o conjunto de porções de código ativáveis quando esta condição for verdadeira. O escopo de efeito da condição *Cond1* é formado pelas duas subestruturas *X* e *Z*. Já o escopo de efeito da condição *Cond2* é a subestrutura *Y*.

O escopo de controle será diferente do escopo de efeito quando a mesma condição for avaliada repetidas vezes. Na Figura 1 a condição é avaliada no bloco *Controlar ação B* e é avaliada de novo após o retorno da execução do ramo que contém este bloco no bloco *Controlar ação A*.

Recom. 42: Para cada condição procure assegurar que o escopo de efeito seja igual ao escopo de controle.

1.7.3. Repetições

Recom. 43: Em Java utilize `break` com rótulo para sair de estruturas aninhadas.

Recom. 44: Não crie variáveis temporárias apenas para controle de término de um ciclo, use `break` ou `return` para sair de repetições antes de processar todos os elementos.

Recom. 45: Evite o uso de `continue`.

Figura 2. Fluxo dos dados em uma repetição

Recom. 46: Antes de ativar o corpo da repetição, o estado corrente deve estar completamente definido.

Recom. 47: Ao retornar do corpo para o controle da repetição, o estado corrente deve corresponder ao próximo estado a ser processado;

Recom. 48: Durante a execução da repetição cada iteração deve corresponder a um estado deve ser diferente dos demais.

Recom. 49: O número de estados de uma repetição deve ser finito.

Assegurar que o número de estados de uma repetição seja finito pode requerer truques de programação. Exemplo: caminhamento em grafo.

```
void VisitarFilhos( tpQQ * pNo )
{
   tpQQ* pNoCorr ;
   if ( pNo == NULL )
   {
      return ;
   } /* if */
   pNoCorr = pNo->pOrgFil ;
   while ( pNoCorr )   // entra em loop se existirem ciclos
                       // no grafo
   {
      VisitarFilhos( pNoCorr ) ;
      pNoCorr = pNoCorr->pProxIrmao ;
   } /* while */
} /* VisitarFilhos */
```

Código modificado

```
void VisitarFilhos( tpQQ * pNo )
{
   tpQQ* pNoCorr ;
   if ( pNo == NULL )
   {
      return ;
   } /* if */
   if ( FoiVisitado( pNo ))
   {
      return ;
   } /* if */
   MarcarVisitado( pNo ) ;
```

```
   pNoCorr = pNo->pOrgFil ;
   while ( pNoCorr )
   {
      VisitarFilhos( pNoCorr ) ;
      pNoCorr = pNoCorr->pProxIrmao ;
   } /* while */
} /* VisitarFilhos */
```

Para que este algoritmo funcione, precisa-se de uma estrutura auxiliar `Visitados` na qual se registram todos os nós que já foram visitados. Antes de iniciar o caminhamento esta estrutura auxiliar deve ser esvaziada. Durante o caminhamento `Visitados` recebe uma marca para cada um dos nós visitados. A função `FoiVisitado` retorna `TRUE` caso a estrutura `Visitados` contenha uma marca correspondente ao nó passado por parâmetro.