UNIVERSIDADE DE BRASÍLIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMAÇÃO SISTEMÁTICA
Prof. Francisco A. C. Pinheiro

Padrão de Codificação e Estilo para a Linguagem C

(adotado para os trabalhos da disciplina)

VERSÃO 2007-2

Sumário

Capítulo 1

Linguagem

1.1 Declarações e definições

1.1.1 Ponteiros e vetores

Regra 1. *Aritmética de ponteiros deve ser aplicada apenas a ponteiros que referenciam vetores ou elementos de vetores.*

RAZÃO: a adição e subtração de ponteiros que não apontem para vetores resulta em um comportamento indefinido.
EXEMPLO:

`Desconforme:` `a` não aponta para vetor

```
void funNAO (const int *a) {
  ...
  int *p1 = a + 1;
  ...
}
```

`Conforme:` `a` aponta para vetor

```
void funSIM (const int a[]) {
  ...
  int *p1 = a + 1;
  ...
}
```

FONTE: [1, regra 2-05-00-16], [2], [7, Ap. 7, regra 14] exem01.c

---

Regra 2. *Um ponteiro só deve ser usado para acessar elementos de um vetor se tiver sido declarado como ponteiro para este vetor.*

RAZÃO: Evita acesso indevido a posições de memória que não sejam elementos de vetores.
EXEMPLO:

`Desconforme`

```
void funNAO (const int *a) {
  printf("%d ", *(a + 3));
  printf("%d ", *(a++));
  printf("%d \n", a[2]);
}
```

`Conforme`

```
void funSIM (const int a[]) {
  printf("%d ", *(a + 3));
  printf("%d ", *(a++));
  printf("%d \n", a[2]);
}
```

FONTE: [1, regra 2-05-00-16], [2] exem02.c

Regra 3. *Operações aritméticas ou relacionais (>, <, >=, <=) entre ponteiros são permitidas apenas se os ponteiros apontarem para o mesmo vetor.*

RAZÃO: As operações envolvendo ponteiros apontando para diferentes vetores produzem valores indefinidos, que dependem do local da memória onde os vetores são armazenados.
EXEMPLO:

```
int fun(int origem[], int destino[]) {
  int res;
  int *p1 = (origem + 1);
  int *p2 = (destino + 1);
  res = p1 - origem; // Conforme: res = 1
  res = origem - p1; // Conforme: res = -1
  res = origem - origem; // Conforme: res = 0
  res = p2 - p1;     // Desconforme: res indefinido
  res = p1 > origem; // Conforme: res = 1 (verdadeira)
  res = origem >= p1; // Conforme: res = 0 (falsa)
  res = origem >= origem; // Conforme: res = 1 (verdadeira)
  res = p2 > p1;     // Desconforme: res indefinido (0 ou 1)
}
```

FONTE: [1, regra 2-05-00-17], [2] exem04.c

Regra 4. *Deve-se assegurar que o ponteiro não seja nulo, antes de sua utilização.*

EXEMPLO:

```
void fun(int *vet) {
  if (vet != NULL) {
     printf("%d\n", *(vet + 2)); // Conforme
  }
  printf("%d\n", *(vet + 5)); // Desconforme: vet pode ser nulo
}
```

FONTE: [2] exem05.c

Regra 5. *Declare vetores definindo seu tamanho explicitamente, ou implicitamente por inicialização.*

RAZÃO: Embora seja possível declarar um array com tipo incompleto, é mais seguro que seu tamanho seja explicitamente determinado; torna clara a intenção do programador.
EXEMPLO:

```
      int array1[ 10 ];
extern int array2[ ];        // Desconforme
      int array3[ ] = { 0, 10, 15 };
```

FONTE: [1] exem19.c

Recomendação 6. *Deve-se usar índices inteiros, ao invés de ponteiros, para acessar os elementos de um vetor.*

RAZÃO: Facilita a compreensão do programa.
EXEMPLO:

Desconforme

```
void funNAO (const int a[]) {
  printf("%d ", *a);
  printf("%d ", *(++a));
  printf("%d \n", *(a + 1));
}
```

Conforme

```
void funSIM (const int a[]) {
  printf("%d ", a[0]);
  printf("%d ", a[1]);
  printf("%d \n", a[2]);
}
```

FONTE: [1, regra 2-05-00-15] exem03.c

Recomendação 7. *Inicialize vetores multidimensionais e vetores de estruturas usando chaves para delimitar seus elementos.*

RAZÃO: O uso de chaves torna mais claro o formato dos vetores e estruturas.
EXEMPLO: Os vetores `vetorNAO` e `contasNAO` estão desconformes.

```
typedef struct {
  int cod;
  char tipo;} tipoC;
tipoC contasNAO[] = {1, 'a', 2, 'b', 3, 'a'};
tipoC contasSIM[] = {{1, 'a'}, {2, 'b'}, {3, 'a'}};
int vetorNAO[2][3] = {1, 2, 3, 4, 5, 6};
int vetorSIM[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

FONTE: [2] exem20.c

1.1.2 Funções e procedimentos

Regra 8. *Toda função deve possuir um protótipo que especifique explicitamente um tipo de retorno (ou void) e um número definido de parâmetros, especificando-se o tipo de cada um, sem, entretanto, incluir nomes.*

RAZÃO: O protótipo evita chamadas incompatíveis à função, enquanto a ausência de nomes evita a colisão com macros previamente definidas.
EXEMPLO:

Desconforme

```
int f1 ( ) ;
int f1 (int a) ;
int f2 (char * nome)
```

Conforme

```
int f2 (int) ;
int f4 (char *) ;
int f6 (void) ;
```

FONTE: [4] [2] exem06.c
OBSERVAÇÕES: O padrão da Exida [2] recomenda o uso de nomes em protótipos de função.

1.1.3 Tipos, estruturas e uniões

Recomendação 9. *Use `typedef` para todas as estruturas `struct` e `union`, colocando as definições de tipo antes das declarações.*

RAZÃO: Elimina o inconveniente da palavra extra `struct` ou `union` em referências. As definições antes das declarações possibilitam o uso de tipos circulares.
EXEMPLO:

```
typedef struct foo TipoFoo;
typedef struct boo TipoBoo;

struct foo {
  TipoBoo *bElem;
};
struct boo {
  TipoFoo *fElem;
};
```

FONTE: [4] exem13.c

Recomendação 10. *Não misture declarações com definições de tipos.*

RAZÃO: Todas as declarações de tipos devem ser visíveis e bem delimitadas, para facilitar a leitura e o entendimento.
EXEMPLO: A coluna esquerda mistura declarações e definições de variáveis e tipos. A coluna direita mostra como as mesmas declarações e definições devem ser feitas.

Desconforme	Conforme

`struct foo {`
`  int x;`
`} obj;`
	`typedef struct foo Tfoo;`
	`struct foo {`
	`  int x;`
	`};`
	`Tfoo obj;`

`typedef struct {`
`  int x;`
`} Ts;`
	`typedef struct anom Ts;`
	`struct anom {`
	`  int x;`
	`};`

FONTE: [4] exem14.c

Recomendação 11. *Um identificador não deve referir-se a um tipo e a um objeto ou função simultaneamente.*

RAZÃO: Esta situação dificulta a compreensão, podendo causar confusão com o nome que está sobrecarregado.
EXEMPLO: As declarações abaixo estão desconformes:

```
typedef struct vetor { int x; } vetor;
typedef struct fun { int x; } (*fun)();
      struct vet { int x; } vet;
```

FONTE: [1] exem15.c

1.1.4 Constantes e enumerações

Regra 12. *As variáveis que não são modificadas, incluindo as variáveis paramétricas e os ponteiros para variáveis que não são modificadas, devem ser declaradas como* `const`.

RAZÃO: Se uma variável não deve ser modificada deve ser declarada de forma a assegurar este comportamento.
EXEMPLO: Neste exemplo as variáveis `p2`, `p3` e `i` estão desconformes, pois não são modificadas. Observe que apenas o conteúdo de `p4` é modificado; compare com `p5`.

```
void fun(const int p1,
       int p2,       // Desconforme
       int * p3,     // Desconforme
       const int * p4,
       int * const p5) {
  int i = 30;        // Desconforme
  const j = 37;
  p4 = p3;
  *p5 = *p3;
  printf("%d %d %d %d %d\n",p1,p2,*p3,*p4,*p5);
  printf("%d %d\n",i,j);
}
```

FONTE: [1], [5], [2] exem17.c

Regra 13. *Não use atribuição na inicialização dos itens de uma enumeração, exceto o primeiro ou se todos forem explicitamente inicializados.*

RAZÃO: Quebra a seqüência de enumeração, dificultando a compreensão.
EXEMPLO: Na declaração abaixo, `B` e `C` representam os mesmos valores que `D` e `E`.

```
enum {A, B, C, D=1, E, F};    // Desconforme
enum {A=0, B=1, C=2, D=1, E=2, F=3 } // Conforme
```

FONTE: [2] exem18.c

Recomendação 14. *Declare constantes usando enumeração e variáveis* `static const`*, ao invés de* `#define` *ou literais.*

RAZÃO: O uso de literais dificulta a manutenção, enquanto o uso de `#define` pode causar problemas com alguns depuradores [4].
EXEMPLO:

```
#define RED 0xF00     // Desconforme: #define
void f(const float valor) {
  enum { Red = 0xF00, Blue = 0x0F0, Green = 0x00F };
  static const float pi = 3.14159265358;
  const int TAXA = 2;
  float res;
  res = TAXA * valor;
  res = 2 * valor;     // Desconforme: literal 2
}
```

FONTE: [4], [5] exem16.c

1.1.5 Geral

Regra 15. *Todo tipo, objeto ou função deve ser declarado uma única vez.*

RAZÃO: Uma única declaração possibilita que o compilador detecte tipos incompatíveis para o mesmo identificador. Normalmente, o identificador deve ser declarado em um arquivo cabeçalho que será incluído em todo arquivo que o defina ou use.
EXEMPLO: No programa abaixo a variável `b` é declarada como `float` no arquivo `exem08.c` e como `int` no arquivo `exem08cab2.h`. Como, pela estrutura do programa, o compilador não tem acesso às duas declarações, a incompatibilidade de tipos não é detectada. Já a variável `a` teria detectada uma possível incompatibilidade de tipos.

EXEM08CAB1.H ⇒

```
extern int a;
extern void fun8b();
```

EXEM08.C

```
#include <stdio.h>
#include "exem08cab1.h"
extern float b;
int a = 156;
int main() {
  fun8b();
  b = 57;
  printf("main : %d %f\n",a,b);
  fun8b();
}
```

⇓ EXEM08CAB2.H ⇒

```
#include "exem08cab1.h"
extern int b;
```

EXEM08B.C

```
#include <stdio.h>
#include "exem08cab2.h"
int b = 432;
void fun8b() {
  printf("fun08b: %d %d\n",a,b);
}
```

FONTE: [1], [2] exem08.c , exem08b.c, exem08cab1.h, exem08cab2.h

---

Regra 16. *Os arquivos de cabeçalho devem conter apenas declarações que não reservem espaço em memória (isto é, não devem conter definições).*

RAZÃO: Contribui para que não haja várias definições de um mesmo objeto, além de ocultar a implementação da interface.
EXEMPLO:

```
// arquivo cabeçalho arq.h
   void f1 ( );
   void f2 ( ) { }   // Desconforme: define a função f2
   int a;            // Desconforme: define a variável a;
```

FONTE: [1], [4]

Regra 17. *Funções e variáveis externas usadas em apenas um arquivo devem ser declaradas como `static` neste arquivo.*

RAZÃO: A declaração `static` faz com que o elemento só possa ser referenciado no arquivo onde está declarado, evitando a exportação indesejada de nomes.
EXEMPLO: A referência a variável `a` em `exem.c` é inválida porque ela está declarada como `static` em `exem_a.c`.

EXEM.C

```
extern int a;
extern void fun(void);
int main() {
  a = 23;
  fun();
}
```

EXEM_A.C

```
#include <stdio.h>
static int a;
void fun() {
  printf("%d\n",a);
}
```

FONTE: [4], [5] exem11.c, exem11a.c

Recomendação 18. *As funções e variáveis globais devem ser declaradas em arquivos cabeçalhos usando `extern` explícito.*

RAZÃO: O uso de `extern` reafirma o caráter declaratório desses elementos. Particularmente, não se deve declarar objetos externos no corpo de funções. Se o elemento, função ou variável, será referenciado em outros arquivos, sua declaração deve estar em um arquivo cabeçalho; caso contrário, não deve ser declarado como `extern`.
EXEMPLO: Considerando que o trecho abaixo esteja em um arquivo de implementação, as variáveis `a` e `b` não estão em conformidade com esta regra.

```
#include <stdio.h>
extern int a;
char fun() {
  extern int b;
  printf("%d \n",b);
}
```

FONTE: [4] exem10.c

Recomendação 19. *A ocultação de nomes deve ser evitada. Em particular, os identificadores globais, incluindo os nomes definidos por `typedef` e os nomes de variáveis automáticas (locais) declaradas como estáticas devem ser únicos.*

RAZÃO: A ocultação de nomes torna o código confuso. Os nomes globais são referenciados em várias partes de um programa; redefiní-los pode causar confusão. O mesmo vale para variáveis estáticas automáticas.
EXEMPLO: No código abaixo temas as seguintes situações de desconformidade:

O nome `ul32` é usado como identificador de tipo e nome de variável.

A variável `a` é declarada como global e redefinida em `fun1`.

O nome da variável local estática `c` é usado também em `fun2`.

A variável local `d` é ocultada em um bloco interno de `fun1`.

A variável paramétrica `b` é ocultada em um bloco interno de `fun1`.

```
typedef unsigned long ul32;
int a = 12;
void fun1(int b) {
  static int c;
  int d = 788;
  int ul32 = 234;
  int a;
  {
    int d = 921;
    float b = 501;
  }
void fun2() {
   char c;
 }
}
```

FONTE: [1], [4] exem12.c

1.2 Expressões e controle

1.2.1 Elementos não utilizados

Regra 20. *O valor retornado por uma função cujo tipo não seja declarado* `void` *deve ser utilizado.*

RAZÃO: Se o valor produzido por uma expressão não for usado, a expressão não precisa existir. No caso de funções, se a chamada é feita apenas por conta de efeitos colaterais, o correto é codificar esses efeitos como um procedimento (função sem valor de retorno).
EXEMPLO:

```
int fun(int);

int main() {
  int res = fun(3);
  fun(3);           // Desconforme
}
```

FONTE: [1], [2] exem07.c
OBSERVAÇÕES: Pode-se, alternativamente, descartar o valor de retorno explicitamente, impondo-lhe o valor `void` (como, por exemplo, em `(void)fun(3)`).

Regra 21. *Um programa não deve conter elementos (variáveis, declaração de tipos, funções e parâmetros) não utilizados.*

RAZÃO: Se um elemento não é utilizado, não é necessário e não deve ser declarado.
EXEMPLO: Neste exemplo o tipo `int32`, a variável paramétrica `p2`, a variável local `b`, a variável externa `c` e a função `semuso` não são usadas; portanto, devem ser excluídas.

```
typedef int int16;
typedef long int32;

int comuso(int16 p1, char p2) {
  static int a;
  int b;
```

```
    if (p1 != a)
      a = a + p1;
    return a;
  }
  int c;
  void semuso() {
    printf("funcao sem uso");
  }
  int main() {
    printf("%d\n", comuso(2, 'c'));
  }
```

FONTE: [1] exem21.c

1.2.2 Fluxo de controle

Regra 22. *Um programa não deve conter código não realizável (unreachable code).*

RAZÃO: O código é não realizável quando, por construção, nunca será executado. Não é necessário e pode indicar erro de lógica ou distração.
EXEMPLO:

```
  void funNAO(int para) {
    int local;
    local = 0;
    switch (para) {
      local = para; // Não realizável
      case 1: {
        printf("caso um\n");
      }
      case 2: {
        printf("caso dois\n");
      }
      default: {
        printf("caso padrao\n");
      }
    }
    return local;
    local++;        // Não realizável
  }
```

FONTE: [1] exem27.c

Regra 23. *Um programa não deve conter código morto.*

RAZÃO: Código morto é o código cuja execução não afeta o processamento. É desnecessário e pode indicar distração ou erro de lógica.
EXEMPLO: No código abaixo a função retorna o valor do argumento recebido acrescido de 99. As linhas marcadas não afetam este comportamento.

```
  int possuiCodMorto(int para) {
    int local = 99;
    para = para + local;
    local = para;     // desconforme
```

```
    if (0 == local)   // desconforme
      local++;        // desconforme
    return para;
  }
```

FONTE: [1] exem28.c

Regra 24. *Um programa não deve conter caminhos intransitáveis.*

RAZÃO: Um caminho intransitável existe sintaticamente, mas a semântica da aplicação assegura que nunca será executado. É desnecessário e pode indicar distração ou erro de lógica.
EXEMPLO: No código abaixo, considerando que `val` e `valU` são armazenados em 16 bits, as expressões condicionais, por serem sempre verdadeiras ou sempre falsas, levam a códigos intransitáveis.

```
    enum ec { VERMELHO, AZUL, VERDE } cor;
    unsigned short valU;   // 16 bits, sem sinal
    short val;             // 16 bits

    if (valU < 0U)         // sempre falso
      printf("teste um\n");
    if (valU <= 0xffffU)   // sempre verdadeiro
      printf("teste dois\n");
    if (val < 130)
      printf("teste tres\n");
    if ((val < 10) && (val > 20)) // sempre falso
      printf("teste quatro\n");
    if ((val < 10) || (val > 5)) // sempre verdade
      printf("teste cinco\n");
    if (cor <= VERDE)
      printf("teste seis\n");
    if (val > 10) {
      if (val > 5) {       // sempre verdade
        printf("teste sete\n");
      }
    }
```

FONTE: [1] exem29.c

1.2.3 Avaliação de expressões

Regra 25. *O lado direito dos operadores lógicos não deve conter efeitos colaterais.*

RAZÃO: A avaliação dos operandos do lado direito é condicional e pode não ocorrer.
EXEMPLO: No trecho abaixo o primeiro `if` está conforme a regra e o segundo não está. Se a função `funComEfeito` tiver efeitos colaterais e a função `funSemEfeito` não tiver, temos que o terceiro `if` é desconforme e o quarto está em conformidade com esta regra.

```
    if ((y == 0) || (y == x + 1))
      printf("condicao em conformidade\n");
    if ((y == 0) || (y == x++))
      printf("condicao desconforme\n");
    if ((y == 0) || (y == funComEfeito()))
      printf("condicao desconforme\n");
    if ((y == 0) || (y == funSemEfeito()))
      printf("condicao em conformidade\n");
```

FONTE: [1], [2] exem23.c

Recomendação 26. *As expressões condicionais devem ser do tipo booliano e especificadas explicitamente através de operadores relacionais ou lógicos.*

RAZÃO: Torna mais claro o significado da expressão.
EXEMPLO:

Desconforme	Conforme
`(i)`	`(i == 0)`
`(!x)`	`(x != 0)`
`(c)`	`(c == '\0')`

FONTE: [4], [1], [3], [2] exem22.c

1.2.4 Estruturas de controle

Recomendação 27. *No comando `switch` todas as cláusulas devem terminar com `break`, `continue`, `return`, ou comentário do tipo `segue o fluxo`. Um `break` deve ser colocado na última cláusula.*

RAZÃO: Torna explícita a intenção do programador.
EXEMPLO: O trecho de código abaixo está em conformidade com esta regra.

```
switch (x) {
  case 0:
        printf("opcao 0\n");
        break;
  case 1:
  case 2:
        printf("opcao 1 ou 2\n");
        // segue o fluxo
  default:
        printf("todo o resto\n");
        break;
  case 3:
        printf("opcao 3\n");
        break;
}
```

FONTE: [4], [2] exem24.c

Recomendação 28. *As cláusulas das estruturas condicionais e iterativas devem ser delimitadas por chaves, mesmo se forem vazias ou contiverem apenas um elemento. Admite-se não usar chaves para comandos `if` não aninhados, quando suas cláusulas contiverem apenas um elemento.*

RAZÃO: Melhora a legibilidade, definindo de forma clara o escopo de cada cláusula. definindo de forma clara o escopo de cada cláusula. Observe que alguns padrões especificam o uso de chaves em todas as situações.
EXEMPLO:

Desconforme

```
if ( teste ) ;
```

```
for ( i = 0; i < 10; i++ )
  x++ ;
```

```
while ( teste )
  x++;
```

Conforme

```
if ( teste ) {
}
```

```
if ( teste )
  x = 0;
else
  y = 1;
```

```
while ( teste ) {
  x++;
}
```

FONTE: [1], [3] exem25.c

Recomendação 29. *Cada `if` em uma estrutura de `if`s aninhados deve ter ambas as cláusulas (então e senão) delimitadas por chaves.*

RAZÃO: Comandos `if`s aninhados são mais difíceis de entender; o uso das chaves facilita a identificação de cada cláusula. O `else` final, se não contiver ações, deve conter um comentário explicando a falta de ação.
EXEMPLO: Os dois trechos a seguir estão em conformidade com esta regra.

```
if ( x < 0 ) {
  if ( y > 5) {
    log_error( 3 );
    x = 0;
  } else {
    // sempre y > 5
  }
} else {
 log_error( 4 );
}
```

```
if ( x < 0 ) {
  log_error ( 3 );
  x = 0;
} else {
  if ( y < 0 ) {
    x = 3;
  } else {
   // sempre: y < 0
  }
 }
```

FONTE: [1] exem26.c
OBSERVAÇÕES: Este estilo é conhecido como programação defensiva.

1.2.5 Imposição de tipos e alocação de memória

Recomendação 30. *Evite usar imposição de tipos (type cast).*

RAZÃO: Erros produzidos por imposição de tipos são difíceis identificar e tratar. Temos também que a imposição de tipos pode não funcionar quando o programa é compilado com otimização, em virtude das regras de identificação (aliasing rules) utilizadas pelo compilador.
EXEMPLO: No trecho a seguir a primeira impressão produz `1111 1111` ou `2222 2222`, dependendo do nível de otimização utilizado na compilação. A última impressão também resulta em 49 ou 50, dependendo da otimização.

```
short vetor[2];

vetor[0]=0x1111;
```

```
vetor[1]=0x1111;
*(int *)vetor = 0x22222222;
printf("%x %x\n", vetor[0], vetor[1]);

double a = 0.5;
double b = 0.01;
double c = a / b;
printf("%f / %f = %f\n",a,b,c);
printf("%f / %f = %d\n",a,b,(int)(c));
```

FONTE: [5] exem30.c

Recomendação 31. *Não use alocação dinâmica de memória (malloc, realloc, free).*

RAZÃO: Erros referentes a alocação dinâmica de memória são difíceis de identificar e tratar. O programador deve tentar reservar previamente o espaço de memória necessário à execução do programa.

FONTE: [5]

Capítulo 2

Organização, documentação e estilo

2.1 Estrutura dos programas

Regra 32. *Deve-se evitar a inclusão múltipla de arquivos usando-se uma das formas abaixo:*

```
// Início do arquivo
#if !defined ( idModulo )
#define idModulo
  // Conteúdo do arquivo
#endif
// Fim do arquivo
```

```
// Início do arquivo
#ifndef ( idModulo )
#define idModulo
  // Conteúdo do arquivo
#endif
// Fim do arquivo
```

RAZÃO: A inclusão múltipla pode gerar confusão e, no pior dos casos, definições múltiplas e inconsistentes.
FONTE: [1], [2], [5] exem32.c

Recomendação 33. *Use a seguinte organização para os arquivos fontes: inclusão de cabeçalhos do sistema e cabeçalhos locais, definições de tipos, definição de constantes, variáveis globais e funções. É razoável ter-se várias repetições das últimas três seções. Observe que a diretiva* `#include` *só deve ser precedida por comentários e outras diretivas de pré-processamento.*

RAZÃO: O uso de um formato definido facilita a leitura e compreensão dos programas.
EXEMPLO: Observe no esquema abaixo a falta de declarações. Elas devem ser colocadas na interface (arquivos cabeçalhos).

cabeçalhos do sistema	`#include <stdio.h>`
cabeçalhos locais	`#include "exem33.h"`
definição de tipos	`typedef unsigned int intU;`
definição de constantes	`const float G = 9.8;`
	`enum {A=12, B, C} TIPOS;`
definição de variáveis globais	`float taxaMensal;`
definição de funções	`void impProcTeste() { }`

FONTE: [4] exem33.c

Recomendação 34. *Use a seguinte organização para arquivos de cabeçalho: definição de tipos, definição de constantes, declaração de objetos externos e declaração de funções externas. Pode-se ter várias repetições dessa estrutura.*

RAZÃO: A existência de um formato bem definido ajuda a leitura e compreensão da interface.
EXEMPLO: O esquema abaixo ilustra a estrutura sugerida:

definição de tipos	`typedef unsigned short shortU;`
definição de constantes	`const float PI = 3.1416;`
	`enum {A=12, B, C} TIPOS;`
declaração de objetos externos	`extern int codOper;`
declaração de funções	`extern void obtemMatricula();`

FONTE: [4] exem34.h

Recomendação 35. *As funções devem ser estruturadas de formas que as declarações e definições necessárias ocorram antes do código.*

RAZÃO: Colocar todas as declarações e definições no início ajuda a identificar os elementos necessários à compreensão do código.
EXEMPLO: O esquema abaixo ilustra a estrutura sugerida:

```
float funExem(const int p1) {
// declarações e definições

// código
}
```

FONTE: [4] exem35.c

2.2 Comentários

Recomendação 36. *Os comentários devem identificar o módulo, documentar a solução adotada e explicar passagens do código.*

Identificação. Tanto os arquivos de implementação quanto os de interface devem ser identificados. A identificação deve conter o código e nome do módulo, a versão, o nome do autor, a data da criação e, para cada modificação, a data e o nome do mantenedor.

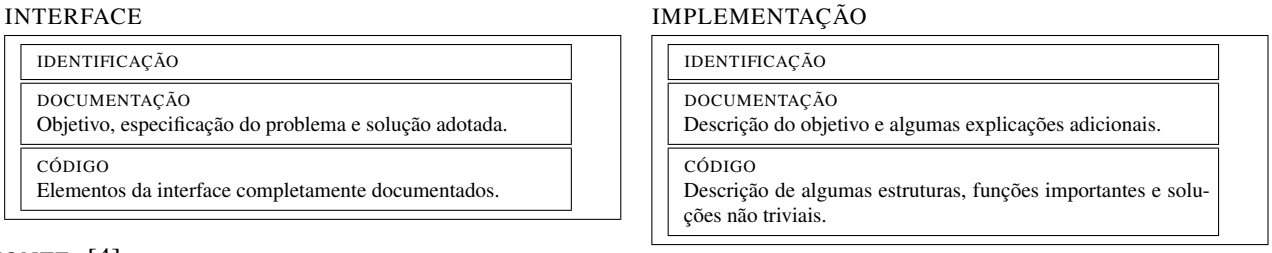
Documentação. A documentação deve conter a especificação do módulo, contemplando o seu objetivo e a solução adotada.

- *Interface. A descrição deve ser detalhada, de forma que todo usuário da interface entenda os seus serviços e mecanismos.*
- *Implementação. A descrição pode resumir-se ao objetivo do módulo.*

Compreensão do código.

- *Interface. Todos os elementos da interface devem ser documentados.*
- *Implementação. Os elementos da implementação devem ser documentados se necessários à compreensão. É usual a utilização de comentários para explicar estruturas de dados, variáveis globais, objetivos das funções mais importantes e partes do código contendo soluções não triviais.*

RAZÃO: O código já é uma descrição e será lido por pessoas que compreendem a linguagem na qual está escrito. Os comentários em linguagem natural são necessários para documentar o domínio do problema e explicar a solução adotada, incluindo o contexto e condições de uso. Com relação às estruturas da linguagem especificamente, algumas explicações podem ser necessárias, mas se um código necessita de muitos comentários para ser entendido deve-se verificar se ele não pode ser reescrito.
EXEMPLO: O esquema abaixo ilustra a documentação sugerida



FONTE: [4]

Recomendação 37. *Os comentários devem estar indentados no mesmo nível que o código ao qual são aplicados. Evite comentários na mesma linha dos códigos; se necessário separe-os à direita com alguns tabs.*

RAZÃO: A indentação subordina o comentário ao elemento que está sendo comentado; a separação evita que se misture visualmente comentários e linhas de código.
EXEMPLO:

```
if (valorTeste()) {
   /* A média será calculada apenas se o valor do
    * teste for verdadeiro. Neste caso, também será
    * determinado o desvio-padrão.
    */
    ...
    printf("exemplo\n");      // comentário separado
}
```

FONTE: [4]

2.3 Codificação de nomes

Recomendação 38. *Cada módulo deve estar associado a um código ou prefixo a ser usado na declaração dos seus identificadores globais.*

RAZÃO: Facilita a identificação dos locais de definição dos elementos e evita a colisão de nomes declarados em múltiplos módulos.

FONTE: [4]

Recomendação 39. *Use as seguintes recomendações para nomear elementos do programa:*

Identificadores ordinários.

Constantes. Todos os caracteres maiúsculos: `const int TX_BASE = 2;`

Constantes enumeradas. Inicial maiúscula. Se necessário considere o uso de um prefixo: `enum Val {Val_Um, Val_D`

Typedef. Inicial maiúscula: `typedef unsigned int TipoIU;`

Variáveis. Inicial minúscula: `int val;`

Funções. Inicial minúscula: `void funExem() {}`

Parâmetros de função. Mesma convenção de variáveis:

```
int funDois(int parUm, char parDois) {}
```

Membros de estruturas e uniões. Mesma convenção de variáveis. Não é necessário usar prefixo para distinguir esses membros: `union Grupo { int valorEx; }`

Tags de struct, union e enum. Inicial maiúscula: `struct Arq { };`

Símbolos do pré-processador. Todos os caracteres maiúsculos: `#define TBUF 100`

Rótulos. Inicial minúscula. De preferência uma única palavra curta.

Se o nome é visível externamente, use o prefixo do módulo. Para nomes compostos de mais de uma palavra diferencie cada palavra com uma inicial maiúscula ou use o sublinhado entre palavras (`InicialMaiuscula`, `Uso_de_sublinhado`.)

FONTE: [4]

Recomendação 40. *(i) Nomes de variáveis devem ser expressões substantivadas. (ii) Variáveis boolianas devem ser nomeadas para representar o significado do valor verdadeiro. (iii) Procedimentos devem ser nomeados pelo que eles fazem, não por como eles fazem. (iv) Nomes de função devem refletir o valor retornado. (v) Nomes de funções boolianas devem refletir o significado do valor verdadeiro.*

EXEMPLO:

	Conforme	Desconforme
(i)	`int valorTotal;`	`int totalizaValores;`
(ii)	`_Bool opcaoOK ;`	`_Bool opcaoInvalida;`
(iii)	`void calcularSaldo() {`	`void calculoComMedia() {`
(iv)	`int salarioFunc() {`	`int calcularSalario() {`
(v)	`if (tamanhoValido(s))`	`if (verificarTamanho(s))`

FONTE: [4]

2.4 Estilo de codificação

2.4.1 Chaves e indentação

Recomendação 41. *Para chaves use um dos dois estilos: o 1TBS (One True Brace Stile), como aparece no livro do Kernighan e Ritchie, ou o 2TBS (Two True Braces Style).*

No estilo 1TBS a chave inicial de um bloco aparece na mesma linha do elemento sintático que determina o bloco, e a chave final aparece em uma linha separada, alinhada com o elemento sintático que determina o bloco.

No estilo 2TBS ambas as chaves de um bloco aparecem em linhas separadas, alinhadas com o elemento sintático que determina o bloco.

RAZÃO: O uso consistente de um estilo evita erros por desatenção.

EXEMPLO:

1TBS

```
if (foo == 7) {
   funA();
} else {
   if (foo == 9) {
      funB();
      funC();
   } else {
      funD();
      funE();
   }
}
```

2TBS

```
if (foo == 7)
{
  funA();
}
else
{
  if (foo == 9)
  {
    funB();
    funC();
  }
  else
  {
    funD();
    funE();
  }
}
```

1TBS

```
do {

} while (teste);
```

2TBS

```
do
{

}
while (teste);
```

1TBS

```
switch (valor) {
  case 1:
    while (valor != 0) {
      funUm();
    }
    break;
  case 2:
  case 3:
    fun2e3();
    break;
  default:
    break;
}
```

2TBS

```
switch (valor)
{
  case 1:
    while (valor != 0)
    {
      funUm();
    }
    break;
  case 2:
  case 3:
    fun2e3();
    break;
  default:
    break;
}
```

FONTE: [4]

2.4.2 Operadores e operandos

Recomendação 42. *Deve existir espaços entre os operadores, exceto para os operadores unários e para aqueles usados como referência a elementos: ., ->, (). Os parênteses externos de uma expressão não precisam de espaços extras.*

EXEMPLO:

Conforme

```
if (value == 0)

soma = (*Aux).a + b / Aux->d;
soma = a + (b * c) / d;
if ((a > 10) || (b <= 5))
res = sqrt(a) + max(a, b);
x = (a > b)? c - d : c + d;
```

Desconforme

```
if (value==0)
if ( value == 0 )
soma = (*Aux) . a +b/ Aux -> d;
soma = a+(b*c)/d;
if ( ( a > 10 ) || ( b <= 5 ) )
res = sqrt ( a ) + max ( a , b );
x=(a>b)? c-d : c+d ;
```

FONTE: [3], [4]

2.4.3 Visualização de linhas

Recomendação 43. *As linhas devem ter um tamanho adequado aos meios nos quais serão impressas ou exibidas. Um tamanho de 80 caracteres, incluindo os espaços iniciais e finais, é usualmente aceito como razoável. Se não for possível manter a linha neste tamanho, ela deve ser quebrada após uma vírgula e antes de um operador. A nova linha deve ser alinhada com o elemento sintático que deu origem à quebra: lista de parâmetros, lado direito de expressões, corpo de bloco, etc.*

RAZÃO: A idéia é possibilitar a visualização sem quebra em terminais e formulários. As quebras, se existirem, devem respeitar a estrutura que está sendo quebrada, facilitando a leitura.

EXEMPLO:

```
funA(primeiroParametro, segundoParametro, terceiroParametro,
     quartoParametro);
funB(primeiroParametro,
     funC(priParametro, segParametro));
primeiraVar = segundaVar * (terceiraVar + quartaVar)
            + quintaVar;
```

FONTE: [3]

Capítulo 3

Pré-processador e compilação

3.1 Macros e definições

Regra 44. *Use funções em linha (inline) no lugar de macros que se comportam como funções.*

RAZÃO: Embora macros sejam geralmente mais rápidas que funções, estas são mais robustas e seguras. Isto é particularmente verdadeiro com relação à checagem do tipo dos parâmetros e à (potencial) múltipla avaliação dos parâmetros.
EXEMPLO:

```
#define FUNC_MACRO(X) ((X)+(X)) // Desconforme
```

FONTE: [1]

3.2 Compilação e dependência

Recomendação 45. *As unidades de compilação devem compilar sem mensagens de alerta.*

RAZÃO: Todo aviso do compilador deve ser tratado como um indicador de problema. É necessário que a indicação seja investigada e, preferencialmente, removida.

FONTE: [5], [2]

Recomendação 46. *Não se deve usar #ifdef e #ifndef para compilar condicionalmente código para várias plataformas.*

RAZÃO: As partes dependentes de plataforma devem ser organizadas em arquivos distintos. O projetista deve então planejar a manutenção destas dependências.

FONTE: [5]

Referências Bibliográficas

[1] Mira. *Misra-C++: Guidelines for the use of C++ language in critical systems*. Draft for comment. Mira Limited, 2007.

[2] Exida.com. *C/C++ Coding Standard Recommendations for IEC 61508*, Version V1, Revision R1.0, July, 2002.

[3] University of Nebraska-Lincoln. *Coding Standards for C, C++, and Java*, JD Edwards Honor Program in Computer Science and Management, version 3.0, September 2003.

[4] Jim Larson. *Standards and Style for Coding in ANSI C*. Página pessoal (www.jetcafe.org/jim/index.html), 1996 (última visita: out/2007).

[5] Shift-Right Technologies. *C/C++ Coding Style & Standards - Coding practices for embedded development*, 2004 (última visita out/2007).

[6] L.W. Cannon, R.A. Elliott, L.W. Kirchhoff et al. *Recommended C Style and Coding Standards*, versão adaptada de "AT&T Indian Hill Laboratory C Style and Coding Standards", Disponível em http://www.psgd.org/paul/docs/cstyle/cstyle.htm (última visita em out/2007).

[7] Arndt von Staa. Programação Modular. Editora Campus, 2000.