Guia de Programação

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos, SP
Brasil
email: jose@dc.ufscar.br

March 9, 2005

Contents

Sumário

Este relatório é um guia para construção de programas corretos, confiáveis, legíveis e reusáveis. Ele é apresentado através de regras de programação. Estas certamente possuem exceções, que não são aqui mencionadas pelo seu número e por sua aplicação apenas a situações muito particulares.

Um excelente guia para construção de software, complementar a este, é o clássico de Kernighan [1]. Sua leitura é altamente recomendada. O apêndice C é uma coleção de frases extraídas deste livro.

Um glossário dos termos utilizados é apresentado no apêndice A.

1 Identificadores

Utilize nomes significativos para identificadores, que permitam ao leitor saber qual a utilidade do identificador sem ler comentários. Nomes de variáveis temporárias ou de controle de laço, em geral, podem ser não significativos, isto é, formados com uma única letra. Procure usar, na ordem, as variáveis `i`, `j`, `k` para controle de laços `for` e `while` (e não usá-los em outras situações). Esta convenção facilita a leitura do programa.

Nunca use os caracteres l (L minúsculo), O ou o (letra 'o' maiúscula e minúscula) como identificadores, pois eles se confundem com 1 e 0 (zero).

Na criação de nomes de identificadores, é útil usar algumas abreviações, como `Max` para designar o máximo de alguma coisa. Estas abreviações são combinadas para formar nomes de variáveis, funções, etc, cujo significado é previsível. No apêndice B são sugeridas algumas abreviaturas.

2 Comentários

Comentários são usados para elucidar a função de um comando ou grupo de comandos que não podem ser compreendidos prontamente pelo leitor. Eles não devem ser usados para comentar variáveis locais usadas para controle de laços ou instruções que são auto explicativas, como no exemplo abaixo:

```
void main()
{
  int i;  // indice do for para percorrer o vetor

  ...
  for (i = 0; i < Max; i++) {
    ...
    NumElem = NumElem + 1; // incrementa o numero de elementos
    }
  ...
  }
```

Excesso ou falta de comentários dificultam a legibilidade dos programas.

Em geral, coloque um comentário para cada grupo de comandos que fazem uma tarefa específica dentro da função. Os comentários podem ser deduzidos a partir do algoritmo de alto nível da função. Por exemplo, se o algoritmo for aquele da Figura 1, o programa correspondente poderia ser o da Figura 2.

O programa desta Figura ilustra também a forma correta de colocar comentários descrevendo a finalidade da função (`/* Coloca em <pal> ...*/`). Vamos admitir que este texto é o que será consultado pelos programadores para saber o que esta função faz. Algumas regras devem ser seguidas para este tipo de comentário:

```
funcao PegaPalavra( s, pal )
    /* Coloca em <pal> a primeira palavra de <s>. Uma palavra
      e uma sequencia de caracteres diferentes de ' '.
      Admite que a palavra exista. */
inicio

- procura em <s> por inicio de uma palavra (caracter diferente de ' ')
- percorre a palavra a partir do seu inicio colocando os caracteres
  encontrados em <pal>.

fim
```

Figure 1: Algoritmo para encontrar palavras

```
void pegaPalavra( char *s, char *pal )
    /* Coloca em <pal> a primeira palavra de <s>. Uma palavra
       e uma sequencia de caracteres diferentes de ' '.
       Admite que a palavra exista. */
{
    // procura em <s> por inicio de uma palavra (carater
    // diferente de ' ')
  while ( *s != '\o'  &&  s == ' ' )
    s++;

    /* percorre a palavra a partir do seu inicio colocando os caracteres
       encontrados em <pal> */
  while ( *s != '\0' && *s != ' ' )
    *pal++ = *s++;
  *pal = '\0';
  }
```

Figure 2: Uso de comentário baseado no algoritmo

- Ele é colocado após o cabeçalho da função e afastado quatro colunas em relação a este:

```
void␣pegaPalavra(␣char␣*s,␣char␣*pal␣)
␣␣␣␣/*␣Coloca␣em␣<pal>␣a␣primeira␣palavra␣de␣<s>.␣Uma␣palavra
␣␣␣␣␣␣␣␣e␣uma␣sequencia␣de␣caracteres␣diferentes␣de␣'␣'.
␣␣␣␣␣␣␣␣Admite␣que␣a␣palavra␣exista.␣*/
```

- Ele deve informar o que a função **faz** e não **como ela faz**. Uma função é uma abstração porque não precisamos saber **como** ela funciona para utilizá-la (nos abstraímos deste detalhe). Portanto, não devemos introduzir esta informação interna no comentário da função. De qualquer forma, deve ser possível mudar o código da função sem alterar a sua descrição (dada pelo seu comentário).

- Ele deve descrever o papel de cada um dos parâmetros da função. Os nomes destes devem ser diferenciados por alguma notação especial. No exemplo, colocamos nomes de parâmetros envolvidos em `< >`. Isto elimina ambigüidades entre os nomes dos parâmetros e palavras do comentário como acontece no exemplo a seguir.

```
int buscaPessoa( Pessoa *v, char *nome )
    /* Retorna 1 se houver pessoa no vetor <v> com nome <nome>.
       Retorna 0 caso contrario */
```

- Ele deve descrever o significado do valor de retorno, se existir, como é feito no exemplo acima.

Exemplos incorretos de declaração de funções são dados abaixo.

1. ```
   void buscaEmpresaCompradora( Empresa &empresa[],
                                int v,
                                int n )
       /* Esta funcao busca o maior pedido e coloca este e a
          respectiva empresa no inicio do vetor */
   ...
   ```

   O comentário:

   - Não descreve o que são os parâmetros.
   - Onde está o "pedido" ?

2. ```
   void procuraNomeProduto( TpPedido &Produto, TpVetEmpresa Empr5 )
       /* Este procedimento usa a funcao ProcuraProduto para verificar se
          determinado produto existe na lista ou nao. Se a funcao
          ProcuraProduto retornar true o produto existe e a partir dai
          serao chamadas as procedures: CalculaPedido, ProcuraNomeEmpresa e
          ImprimePrecoPedido. Se a funcao ProcuraProduto retornar false sera
          emitida a seguinte mensagem: Produto que a X-Tal nao fabrica */
   ```

   Este comentário descreve **como** o procedimento foi implementado e não **o que** ele faz. Ele não descreve o significado dos parâmetros. Observe também que o parâmetro `Produto` é do tipo `TpPedido`, uma incoerência (um produto não é um pedido). O nome `Empr5` também é inadequado (Por que `5` no nome do parâmetro ?).

3.
```
void escrevePedido( TpPedido Pedido)
   /* Tem como objetivo ler o nome da empresa, nome do
      produto  e quantidade pedida do produto */
```

   O comentário não diz que nome da empresa, do produto e quantidade são campos de `TpPedido`. E a função não *lê*, ela *escreve*. Preferencialmente, o comentário não deve se referir a campos de uma estrutura, pois esta pode ser modificada, causando alterações neste comentário. Melhor seria um comentário mais genérico:

```
void escrevePedido( TpPedido Pedido )
    /* Escreve no video os dados de um pedido */
```

4.
```
int achaMaiorEmpresa( TpDadosEmpresa Vet,
                      int n,
                      float Maior )
    /* Este procedimento procura, dentro do vetor de dados
       da empresa, a empresa que mais comprou da X-Tal. */
```

   Não descreve os parâmetros e não diz o significado do valor de retorno da função.

Exemplos corretos seriam:

1.
```
int buscaMaiorEmpCompradora( TpVetEmpresa &VetEmp,
                             int NumElemVetEmp )
    /* Procura no vetor <VetEmp> de <NumElemVetEmp> posicoes
       pela empresa que possui o maior campo <ValorTotalCompras>.
       Retorna o indice desta empresa em <VetEmp> */
```

2.
```
int insereEmpresaVet( TpVetEmpresa &VetEmp,
                      int NumElemVetEmp,
                      TpStrNomeEmpresa NomeEmpresa
                      )
    /* Insere a empresa cujo nome eh <NomeEmpresa> no final
       do vetor <VetEmp> de <NumElemVetEmp> posicoes. O
       campo <ValorTotalCompras> da empresa inserida eh
       inicializado com zero. Retorna 1 se ha espaco
       disponivel em <VetEmp>, 0 caso contrario */
```

   Neste exemplo, são colocados alguns detalhes da função no comentário, como que o nome é inserido ao final do vetor e que o campo `ValorTotalCompras` é inicializado com zero. Freqüentemente, comentários deste tipo são necessários, mas eles podem ser evitados pela utilização de *tipos abstratos de dados* (TAD), que fornecem uma interface mais abstrata (menos detalhes, como "final do vetor" e o nome do campo) para seus usuários. A função `insereEmpresaVet` poderia ser uma função interna a um TAD. Isto é, só seria usada pelos outros procedimentos deste TAD.

Outras observações sobre comentários são dados abaixo:

1. Sempre comente variáveis e constantes globais. Como o escopo destes identificadores é grande, eles são mais sujeitos a usos incorretos. Constantes com objetivo similar podem ser comentadas em grupo:

```
const
      // Opcoes do menu principal
  OpInsere = 1,
  OpRetira = 2,
  OpLista  = 3,
  OpFim    = 4;
```

Observe que, geralmente, este tipo de declaração fica melhor usando um tipo enumerado:

```
enum TpOpcao {
  OpInsere, OpRetira, OpLista, OpFim
  };
```

2. Estruturas de dados, em geral, são auto-explicativas. Se os nomes das variáveis forem significativos, o comentário poderá estar ausente. Ex:

```
struct TpPessoa {
  TpNomePessoa nome;
  int idade;
  char sexo;
  };

struct TpEmpresa {
  TpNomeEmpresa nome;
  float TotalCompras; // total de compras deste ano
  float ImpostoPago;  // imposto pago no ano passado
  };
```

3. Funções muito pequenas em geral não precisam de comentários internos:

```
void zeraVet( int v[], int NumElem )
    // Coloca 0 em todas as <NumElem> primeiras posicoes do vetor <v>
{
  int i;

  for (i = 0; i < NumElem; i++)
    v[i] = 0;
  }
```

4. Segundo Weinberg [3], é melhor colocar comentários ao lado da instrução que se quer comentar do que antes dela. Melhor
`ListaLivre␣=␣0;␣␣//␣libera␣toda␣a␣memoria␣`
do que
`//␣libera␣toda␣a␣memoria␣`
`ListaLivre␣=␣0;`
O motivo é que esta última forma dificulta a leitura normal do programa, já que o leitor deve separar mentalmente o código dos comentários.

   Uma opção às duas formas descritas acima é colocar o comentário quatro (ou duas) colunas depois da coluna de início da instrução a ser comentada:

```
    // libera toda a memoria
ListaLivre = 0;
```

Esta forma é adotada por este guia para todos os tipos de comentários (funções, variáveis locais e globais, constantes, etc).

5. Coloque a especificação de um programa em seu início, precedido da data e autor. Ex:

```
/* Donald Knuth
   30/03/94

   Simulacao da maquina MIX
   ...
*/
...
```

3 Formatação

Use qualquer forma de tabulação que você queira, desde que ela atinja o objetivo de deixar o programa claro. Seja coerente, não usando diferentes formas de tabulação durante o programa. *Use apenas um comando em cada linha.*

Coloque sempre um caracter diferente de branco antes e espaço depois dos caracteres *vírgula* e *ponto e vírgula*. Exemplo:

```
f(1, 2, 3);   { Ok }
f( 1, 2, 3 ); { Ok}
g( 2 ,3);     { indesejavel }
h( 1, 3) ;    { indesejavel }
```

Coloque **um** espaço antes e um depois dos operadores aritméticos/lógicos, com exceção de `*`, `/` e `%`:

```
if␣(␣a␣>␣b␣&&␣x␣+␣y␣<␣z␣)␣a␣=␣b*c;
```

Use pelo menos 3 linhas em branco entre duas funções. Em geral, use linhas em branco para separar, dentro de uma função, grupos de comandos. Veja a Figura 2, onde os grupos de comandos são os correspondentes aos dois comentários internos à função.

Nunca coloque caracteres após a coluna 80, pois eles ficam escondidos (geralmente) no editor de texto e podem não ser impressos apropriadamente na impressora.

Em um comando `if-else`, procure colocar o menor número de comandos seguindo `if (...)` e o maior após o `else`. A Figura 3 é a maneira incorreta, corrigida na Figura 4.[1] O motivo desta regra é a clareza: se após o `if` houver muitos comandos, será mais trabalhoso encontrar os comandos seguintes ao `else`.

Não use `{` e `}` para um único comando:

```
while ( ! eof(arq) ) {
  read(arq, reg)
  }
```

É mais claro ter:

```
while ( ! eof(arq) )
  read(arq, reg);
```

[1] Note que o teste do `if` foi negado.

```
...
if (i > 0) {
  CalculaPreco(p, N);
  N++;
  EfetuaVendas(p, N);
  for (j = 0; j < N; j++)
    cout << p.total[j] << '\n';
  p.total[ N ] = 0;
  soma = soma  + p.total[N]
  }
else
  soma = 0;
...
```

Figure 3: Exemplo incorreto de `if`

```
...
if (i <= 0) {
  soma = 0;
else {
  CalculaPreco(p, N);
  N++;
  EfetuaVendas(p, N);
  for (j = 0; j < N; j++)
    cout << p.total[j] << '\n';
  p.total[ N ] = 0;
  soma = soma  + p.total[N]
  }
...
```

Figure 4: Exemplo correto de `if`

4 Reusabilidade

Nunca utilize entrada e saída de dados (E/S) em uma função que faz algum tipo de processamento (cálculos, operações — tudo o que não é E/S). O motivo é que, sempre que a função for chamada, é feito o processamento **e** a E/S. Não é possível fazer apenas o processamento, o que limita drasticamente o número de situações em que é adequado usar a função.

Como exemplo, a função

```
int fatorial( int n )
    // retorna o fatorial de <n>
{
  int i, p;

  p = 1;
  for (i = 1; i <= n; i++ )
    p = p*i;
  cout << "Fat(" << n << ") = " << p << '\n';
  return p;
  }
```

não é reutilizável. Se alguém quizer uma função para calcular o fatorial de um número, a função acima não seria usada, pois é provável que não se queira **também** escrever o resultado no vídeo.

Melhor é separar o processamento da E/S:

```
int fatorial( int n )
    // retorna o fatorial de <n>
{
  int i, p;

  p = 1;
  for (i = 1; i <= n; i++ )
    p = p*i;
  return p;
  }

void escreveFat( int n )
    // Escreve no video o fatorial de <n>
{
  cout << "Fat(" << n << ") = " << fatorial(n) << '\n';
  }
```

Observe que colocar usar `cout` em uma função e chamar esta função dentro de `fatorial` não resolve o problema — veja Figura 5.

Não use uma função para duas coisas muito diferentes. Em geral, não use nem para duas finalidades semelhantes. Há dois motivos para esta restrição:

1. Se a função for modificada para alterar um dos seus comportamentos ela também modificará o outro, que deveria continuar o mesmo.

2. Abstrações diferentes devem estar representadas por procedimentos diferentes — melhora a legibilidade e facilita futuras modificações no software.

```
void escreveFat( int n, int fat )
    // Escreve no video o fatorial de <n>, que eh <p>
{
  cout << "Fat(" << n << ") = " << fat << '\n';
  }

int fatorial( int n )
    // retorna o fatorial de <n>
{
  int i, p;

  p = 1;
  for (i = 1; i <= n; i++ )
    p = p*i;
  return p;
  escreveFat(n, p);
  }
```

Figure 5: Processamento com E/S mesmo usando funções separadas

Variável booleana[2] (`true` ou `false`) passada como parâmetro por valor é um mal sinal, pois freqüentemente ela é utilizada para o procedimento fazer processamentos diferentes dependendo do valor da variável. Nestes casos, a função deve ser dividida em duas outras, uma para cada tipo de tarefa.

Um exemplo incorreto é:

```
int somaElemVet( TpVetor v,
                 int NumElem,
                 int imp )
    /* Retorna a soma dos <NumElem> primeiros elementos do vetor
       <v>. Se <imp> for 1, o vetor tambem sera impresso no video */
{
  int i, soma;

  soma = 0;
  for (i = 0; i < NumElem; i++) {
    soma = soma + v[i];
    if ( imp )
      cout << v[i] << '\n';
    }

  return soma;
  }
```

Outros exempos de funções pouco reusáveis são dadas abaixo.

- A função

[2]Pode-se definir um tipo `boolean` em C++ utilizando enum: "`enum boolean { false, true }`". Usualmente, usa-se o tipo `int` para representar valores booleanos. PS: este guia foi feito antes de ser acrescentado a C++ o tipo `bool`.

```
const
  X1 = 20,
  Y1 = 10;
...
void leNumero( int n, int inf, int sup )
    /* ... */
{
  gotoxy( X1, Y1 );
  cin >> n;
  while ( n < inf || n > sup ) {
    gotoxy( X1, Y1 );
    cout << "      \n";
    gotoxy( X1, Y1 + 1 );
    cout << "Digite numero no intervalo (inf, sup)\n";
    gotoxy( X1, Y1 );
    cin >> n;
    }
  }
```

lê um numero sempre em uma posição fixa da tela. Isto é ruim, pois obriga todos os que a chamam a usar esta posição, diminuindo a sua reusabilidade. Melhor seria ler o número na posição corrente do cursor. Ou passar a posição `(X1,Y1)` de leitura como parâmetro.

- A função abaixo imprime um menu na tela (com os dados de seu parâmetro `<m>`) e espera que o usuário escolha uma opção, retornando-a em `<op>`. Ela deveria retornar `1` se a última opção foi escolhida e `0` caso contrário.

```
struct TpMenu {
  ...
    // numero de opcoes do menu
  int NumOpcoes;
  }

int menu( TpMenu &m, int &op )
    /* ... */
{
  ...
  cin >> op;
  if ( op == 3 )
    return 1;
  else
    return 0;
  }
```

Contudo, ele só retorna `1` se foi escolhida a opção `3`. Ela foi construída para ser usada com um menu (parâmetro `<m>`) com exatamente três opções. Ela não funciona com número diferente de opções por causa de

```
if ( op == 3 )
```

que deve ser substituído por

```
if ( op == m.NumOpcoes )
```

- A função `buscaPessoa`, definida a seguir,

```
const
  MaxNumPessoas = 50;

struct TpPessoa {
  TpNomePessoa nome;
  ...
  };

typedef
  TpPessoa TpVetPessoa[MaxNumPessoas];

int buscaPessoa( TpVetPessoa v,
                 TpNomePessoa nome )
    /* Retorna 1 se houver pessoa no vetor <v> com nome <nome> e
       0 caso contrario */
{
  int i, achou;

  achou = 0;
  i = 1;
  do {
    if ( strcmp( nome, v[i].nome ) == 0 )
      achou = 1;
    else
      i++;
    } while ( ! achou && i < MaxNumPessoas );
  return achou;
  }
```

admite que o vetor possua pelo menos um elemento, o que pode não ser verdade. O correto seria usar um `while` em lugar de `do-while`. Poderia acontecer de encontrar `nome` na posição `1` de um vetor de zero elementos !

Esta função só poderá ser usada quando o vetor de pessoas está totalmente preenchido (com `MaxNumPessoas` elementos). Melhor seria passar o número de elementos como parâmetro:

```
...
int buscaPessoa( TpVetPessoa v,
                 int NumElem,
                 TpNomePessoa nome )
    /* Retorna 1 se ha pessoa nas primeiras <NumElem> posicoes
       do vetor <v> com nome <nome> e 0 caso contrario */
...
```

De um modo geral, uma função pode se tornar mais reusável aumentando-se o número de seus parâmetros.

5 Comandos de Repetição/Decisão/Desvio

Utilize o comando de repetição adequado em cada caso. Os comandos do lado esquerdo devem ser codificados como mostrado à direita. `I1` (`I2`) é um conjunto qualquer de comandos.

```
i = 0;
while (i < n) {
  ...
  i = i + 1
  }
```

```
for (i = 0; i < n; i++) {
    ...
    }
```

```
I1
while ( expr )
  I1
```

```
do {
  I1
  } while ( expr );
```

```
for (i = 0; i < NumElem; i++ )
  if ( Vet[i] == Valor )
    achou = 1;
```

```
i = 0;
while ( i < NumElem && !achou )
  if ( Vet[i] == Valor )
    achou = 1;
  else
    i++;
```

No último caso (lado esquerdo), o laço continua mesmo após o elemento `Valor` ser encontrado, inutilmente. Usando `while`, o laço termina imediatamente.

O exemplo correto poderia ser melhorado para:

```
i = 0;
while ( i < NumElem && Vet[i] != Valor )
  i = i + 1;
achou = (i <= NumElem); // parenteses sao necessarios
```

Não use `if` em ocasiões em que é possível usar `switch`, como a abaixo:

```
...
if (op == 1 )
  C1
else
  if ( op == 2 )
    C2
  else
    if ( op == 3 )
      C3
...
```

Mais claro é

```
switch (op) {
  case 1 :
    C1;
    break;
  case 2 :
    C2;
    break;
  case 3 :
    C3
  }
```

Nunca modifique a variável de um laço `for` ou seu limite superior, como em

```
...
    // Caso 1
for (i = 0; i < N; i++)
  if ( expr )
    i = 0;
...
    // Caso 2
for (i = 0; i < N; i++)
  if ( expr ) {
    N++;
    v[N + 1] = 0;
    }
```

No caso 1, temos o equivalente a um comando `goto` para antes do laço:

```
antes:
for (i = 0; i < N; i++)
  if ( expr )
    goto antes;
...
```

Dificilmente é justificável uma grande quantidade de comandos `if` aninhados ou muitas opções para comando `switch`, como em

```
const
  MaxChStrSigno = 11;

typedef
  char TpStrSigno[ MaxChStrSigno + 1 ];

char *retornaSignoHoroscopo( TpData &D )
    /* Retorna uma string com o nome do signo correspondente
       à data <D> */
{
  TpStrSigno strSigno;

  switch (D.mes) {
```

```
    case 1 :
      if ( D.dia > 21 )
        strSigno = "Aquario";
      else
        strSigno = "Capricornio";
      break;
    case 2 :
      if ( D.dia > 21 )
        strSigno = "Peixes";
      else
        strSigno = "Aquario";
      break;
    ...
    case 12:
      if ( D.dia > 21  )
        strSigno = "Capricornio";
      else
        strSigno = "Sagitario";
    }
  return strSigno;
  }
```

Melhor seria[3]

```
const
  MaxChStrSigno = 11,
  MaxNumSignos = 12;

typedef
  char TpNomeSigno[ MaxChStrSigno + 1 ];

typedef
  TpNomeSigno TpVetStrNomesSignos[ MaxNumSignos + 1 ];
typedef
  int TpVetDiasInicioSigno[ MaxNumSignos ];


TpVetStrNomesSignos VetStrNomesSignos = {
  "Capricornio", "Aquario", "Peixes", "Aries",
  "Touro", "Gemeos", "Cancer", "Leao",
  "Virgem", "Libra", "Escorpiao", "Sagitario",
  "Capricornio"
  };

TpVetDiasInicioSigno VetDiasInicioSigno= {
  21, 20, 19, 20, 20, 20, 20, 21, 22, 22, 22, 21
  };
```

[3]Solução de Thelma Filipovitch, BCC-93.

```
...
achou = 0;
    // procura por Val na matriz M
while  ( i < NumLinhas ) {
  j = 0;
  while ( j < NumColunas )
    if ( M[i][j] == Val ) {
      achou = 1;
      goto fim;
      }
    else
      j++;
  i++;
  }
fim:
```

Figure 6: Um uso correto do comando `goto`

```
TpNomeSigno RetornaStrSignoHoroscopo( TpData &D )
    /* Retorna uma string com o nome do signo correspondente
       à data <D> */
{
  if ( D.dia <= VetDiasInicioSigno[ D.mes ] )
    return VetStrNomesSignos[ D.mes ];
  else
    return VetStrNomesSignos[ D.mes + 1];
  }
```

O comando `goto` deve ser usado apenas em casos especiais, como sair de laços profundamente aninhados[4] — veja Figura 6.

A função `exit(int)` (termina o programa) é adequada apenas quando é impossível ou inútil (caso raro) continuar a execução do programa.

O comando `return` termina a execução de qualquer função (mesmo as que não retornam valores) e deve ser usado com cuidado, pois facilmente torna o código difícil de ler.

6 Variáveis

Não é necessário, em comandos de decisão (`if`) e repetição (`while`, `do-while`, `for`), testar variável booleana contra `1` (`true`). Testes contra `0` (`false`) podem ser substituídos pela negação `!`.

Ao invés de

```
if ( achou == 1 )
...
if ( abriu == 0 )
...
```

[4] Um laço dentro de outro, dentro de outro, etc.

use

```
if ( achou )
...
if ( ! abriu )
...
```

É mais claro. Não use variável do tipo `char` em lugar de booleana. Exemplo incorreto:

```
...
do {
  fim = 's';   //     's' = sim
      // arquivo inexistente
  existe = 'n';   //  'n' = nao
  ...
      // arquivo ja existe
  existe = 's';
  } while ( fim != 's' );
```

O correto é

```
...
do {
  fim = 0;
  ...
  existe = 0;
  ...
  existe = 1;
  } while ( ! fim );
```

Ou melhor:

```
enum boolean { false, true };
...
do {
  fim = false;
  ...
  existe = false;
  ...
  existe = true;
  } while ( ! fim );
```

Evite colocar a palavra `Nao` no início do nome de variáveis booleanas, como em `NaoAchou`, `NaoExiste`. Estes nomes tornam menos legível o programa:

```
if ( ! NaoExiste && NaoAchou ) ...
```

melhor seria:

```
if ( Existe && ! Achou ) ...
```

7 Parâmetros

Se um parâmetro for passado por valor, você obrigatoriamente deverá usá-lo para alguma coisa ! É incorreto, na primeira utilização de um parâmetro por valor, inicializá-lo. Ex:

```
void DesenhaMoldura( int x1, int y1, int x2, int y2, char chBorda )
    /* ... */
{
  chBorda = ChBordaSimples;  // erro !
  ...
  }
```

Vetores globais de constantes, quase sempre, devem ser passados como parâmetros. Use outras constantes globais com cautela (veja item 4 na página 12).

8 Constantes

Os únicos locais onde uma constante numérica diferente de 0 ou 1 deve aparecer é na declaração de uma constante (com `const` ou `#define`). Ex:

```
const
  MaxChNomePessoa = 60,
  MaxPessoas = 40;
...
gotoxy(10, 15); // Errado !
```

Use uma constante para cada finalidade. Não use, por exemplo, `MaxChStr` para máximo de caracteres de um nome e de um endereço (ou nome de arquivo). Um exemplo errado é:

```
const
  MaxChStr = 20;

struct TpPessoa {
  char nome[MaxChStr + 1];
  char ender[MaxChStr + 1];
  char sexo;
  };
```

O motivo é que, se houver necessidade de aumentar o número de caracteres do endereço (aumentando o valor de `MaxChStr`), será aumentado o tamanho do nome também, sem necessidade. Estas variáveis representam coisas diferentes e assim devem ser tratadas.

Deve-se definir uma constante para cada campo. O ideal é um tipo para cada *string* diferente:

```
const
  MaxChNomePessoa  = 50,
  MaxChEnderPessoa = 60;

typedef
  char TpNomePessoa[MaxChNomePessoa + 1];
typedef
  char TpEnderPessoa[MaxChEnderPessoa + 1];
```

```
struct TpPessoa {
  TpNomePessoa nome;
  TpEnderPessoa ender;
  char sexo;
  };
```

É provável que variáveis do mesmo tipo que o campo `nome` sejam declaradas no programa. É mais seguro declarar

```
TpNomePessoa NomePes;
```

do que

```
char NomePes[MaxChNomePessoa + 1];
```

pois no último corre-se o risco de se enganar ao colocar o tamanho do vetor.

O nome de uma constante deve refletir o seu significado e não o seu próprio valor. Por exemplo, as constantes

```
const
  trinta = 30,
  seis   = 6;
```

não possuem nomes significativos. O correto seria

```
const
  X_janela = 30,
  Y_janela = 6;
```

Sempre que possível, utilize cálculos com constantes em declarações de outras constantes e tipos. Ao invés de

```
const
  MaxColunasVideo = 80,
  MaxLinhasVideo  = 25,
  MaxCaracteresVideo = 2000;
```

use

```
const
  MaxColunasVideo = 80,
  MaxLinhasVideo  = 25,
  MaxCaracteresVideo = MaxColunasVideo*MaxLinhasVideo;
```

Nesta última declaração a definição da semântica de `MaxCaracteresVideo` é definida precisamente, ao contrário da anterior. E se for necessário alterar o número máximo de colunas, apenas uma declaração deverá ser modificada.

9 Tipos

Coloque dados relacionados em uma estrutura. Por exemplo, considere os dados dos produtos de uma fábrica (nome, preço, etc). Eles poderiam ser arranjados em dois vetores:

```
typedef
  TpStrNomeProduto TpVetNomeProduto[MaxNumProdutos];
typedef
  float TpVetPrecoProduto[MaxNumProdutos];
```

ou em uma estrutura com dois vetores

```
struct TpDadosDosProdutos {
  TpStrNomeProduto VetNomeProduto[MaxNumProdutos];
  float VetPrecoProduto[MaxNumProdutos];
  }
```

Nestas duas formas há uma duplicação desnecessária da constante `MaxNumProdutos`. E elas não são uma boa forma de abstração, já que os dados de um produto estão espalhados em estruturas diferentes, o que dificultará a compreensão do programa que as usa. Melhor é agrupar todos os dados de um produto em um registro e fazer um vetor deste registro:

```
struct TpProduto {
  TpStrNomeProduto NomeProduto;
  float PrecoProduto;
  };

typedef
  TpProduto TpVetProdutos[MaxNumProdutos];
```

Use um registro para cada entidade real. Por exemplo, não use o registro abaixo para armazenar dados de uma empresa (nome, valor total de pedidos) e também de um produto (nome produto, preço).

```
struct Registro {
  TpChNome nome;
  float valor;
  };
```

10 Correção de Programas

Os programas devem funcionar em todos os casos possíveis e não apenas nos casos mais comuns. O mesmo se aplica individualmente a cada rotina do programa. Em uma rotina

```
const
  MaxElem = 30;

typedef
  int TpVetInt[MaxElem + 1];

int insVet( TpVetInst v, int NumElem, int x )
    /* Insere <x> na ultima posicao do vetor <v> de <NumElem>
       elementos. Retorna 0 em erro. */
...
```

devemos verificar se ela funcionará quando

- `NumElem` for `0`;
- não houver espaço em `v`, isto é, `NumElem = MaxElem`.

```
const
  ChBorda = ...

void DesenhaMoldura( ... char *ChBorda ) { ... } ... Const
  MaxOpcoesMenu = 10,
  MaxChStrOpcoesMenu = 40;
typedef
  TpStrMenu  ...
typedef
  TpStrVetMenu  ...
typedef
  TpMenu ...

int Menu( ... ) {
  ...
  }
... typedef
  struct TpData ...
void LeiaData( TpData &D ) ...
```

Figure 7: Declarações de tipos espalhados entre as funções

11 Estruturação do Programa

Variáveis globais devem ser evitadas. Se forem necessárias, prefira declará-las como `static` para restringir o uso delas ao arquivo onde estão definidas.

Coloque todas as declarações de tipos e constantes globais **antes** de qualquer função. Os tipos e constantes serão usados pelas funções.

Colocar estas declarações espalhadas entre as funções (veja Figura 7) prejudica a legibilidade do programa, pois para encontrar a declaração de um dado tipo (ou constante) temos que percorrer o código fonte do ponto em que o tipo está sendo usado até o início do programa (no pior caso).

Não abuse das funções de manipulação de tela (`gotoxy`, `TextColor`, `window`, `TextBackGround`, etc). Elas deixam o programa terrivelmente difícil de ler. Coloque-as, de alguma forma, em procedimentos separados do restante do código.

Coloque qualquer parte do programa que faz uma tarefa específica como uma função, mesmo que seja minúscula. Ex:

```
void fazLinhasEmBranco( int NumLinBranco ) {
  for (int i = 0; i < NumLinBranco; i++)
    cout << '\n';
  }
```

12 Orientação a Objetos

Coloque a parte pública antes da parte privada da classe. Isto facilita a leitura da classe, já que o que interessa aos seus usuários é a parte pública.

Não coloque variáveis de instância na parte pública ou protegida de uma classe. Se isto for necessário, use métodos para recuperar e obter o valor de variáveis. Usualmente, estes métodos

```
... class Pessoa {
  public:
    char *getNome();
    void  setNome( char *p_nome );
    int   getIdade();
    void  setIdade( int p_idade );
    ...
  private:
    TpNomePessoa nome;
    TpEnderPessoa ender;
    int idade;
  }
```

Figure 8: Declaração de uma classe com indentação correta

possuem nomes iniciando com `get` e `set` (ou `put`) — veja Figura 8.

Classes amigas (`friends`) em C++ não prejudicam a proteção de informação a menos que sejam extensivamente usadas. É melhor ter classes amigas do que variáveis de instância públicas ou protegidas. No primeiro caso, sabemos que partes do programa devem ser reescritas se a representação de uma classe for alterada: serão os métodos da classe mais as classes e funções amigas. No último caso (variáveis públicas/protegidas), não sabemos o que deve ser modificado.

Variáveis de instância são informações caracteristicas das entidades que as classes representam. Como exemplo, na classe

```
class VetorInt {
  public:
    int soma();
    ...
  private:
    int vet[MaxElemVet];
    int n;
    int s, i;
  };

int VetorInt::soma() {
  s = 0;
  for (i = 0; i < MaxElemVet; i++)
    s += vet[i];
  return s;
  }
...
```

as variáveis de instância `s` e `i` estão incorretas. Elas devem ser variáveis locais do método `soma`. Não é necessária a existância de `s` e `i` para se definir um vetor.

Ao criar uma subclasse, devemos sempre fazer a pergunta : "um objeto da subclasse é também um objeto da superclasse ?". Se a resposta for não, a subclasse será semanticamente incorreta e poderá causar erros de execução no programa porque objeto da subclasse pode ser usado onde se espera objeto da classe. Como exemplo, considere um procedimento que aceita um parâmetro que deve ser

um `Poligono`. É correto passarmos a ele um `Retangulo` mas não um `Circulo`. Mas o compilador não acusaria o erro se o programador tivesse declarado que `Circulo` herda de `Poligono`.

Alguns exemplos deste tipo de erro são:

- Uma classe `VetorPessoas` herda de `Pessoa`. Um vetor de pessoas não é uma pessoa. Uma das alternativas de construção correta desta classe é `VetorPessoas` possuir a seguinte variável de instância:

  ```
  Pessoa vet[MaxPessoas];
  ```

- Uma classe `Empresa` herda de `VetorFuncionarios`. Novamente, uma empresa não é um conjunto de funcionários. A classe `Empresa` deve ter como uma variável de instância uma variável do tipo `VetorFuncionarios`. Isto significa que a empresa **possui** funcionários.

Um dos objetivos de proteção de informação é impedir a manipulação incorreta dos dados (variáveis de instância) dos objetos. Os dados são alterados apenas através de métodos, e estes preservam a consistência dos dados. Este objetivo é corrompido se os métodos em si danificarem os dados, como no exemplo abaixo.

```
    // classe vetor de empresas
class VetEmpresa {
  public:
      // incrementa o numero de empresas
    void inc() { NumEmp++; }
    ...
  private:
    int NumEmp;
    TpVetEmpresas VetEmp;
  ...
  };
```

O método `inc` permite incrementar o número de empresas no vetor sem a inserção dos dados de uma nova empresa.

Comentários de métodos (aqueles utilizados como documentação para que outros programadores utilizem a classe) devem seguir as mesmas recomendações que comentários para funções — veja seção 2 na página 2. Eles devem informar **o que** (e não **como**) o método faz. Não devem citar as variáveis de instância da classe, que são dados protegidos e podem ser modificados sem invalidar os usuários[5] desta classe.

Métodos são operações sobre os dados dos objetos. Logo, métodos devem usar as variáveis de instância da classe. Se não usam, provavelmente[6] estão errados. Um exemplo deste erro é dado abaixo.

```
class Produto {
  public:
    void init( TpFichaProduto &DadosProduto );
    void leia( TpFichaProduto &DadosProduto );
    ...
  private:
```

[5] Funções, classes e outros métodos que declaram variáveis ou parâmetros da classe.

[6] Há exceções !

```
    TpFichaProduto produto;
  };

void main() {
  Produto prod;
  TpFichaProduto dp;

  ...
  prod.leia( dp );
  prod.init( dp );
  ...
  }
```

A função do método `leia` é ler os dados de um objeto produto (`prod.leia( dp )`). O problema é que ele coloca os dados lidos no parâmetro formal `DadosProduto` (parâmetro real `dp`) e não na variável de instância `produto`. Este método não usa a variável de instância. A variável lida `dp` é usada para inicializar o objeto com método `init`. O correto é `leia` não possuir parâmetros e os dados lidos serem colocados na variável de instância `produto`.

Métodos são operações características da entidade que a classe está representando. Logo, não podemos ter uma classe `Pessoa` com método `acelere` ou classe `ConjuntoPessoas` com método `getEndereco` (de uma pessoa). O último método é característico de uma só pessoa, não de um conjunto delas.

Alguns exemplos de classes com métodos incorretos são enumerados abaixo.

- ```
  class VetEmpresas {
    public:
      void localizaEmpresa( TpNomeEmpresa NomeEmpresa );
      void acumulaValorDaCompra( float ValorCompra );
      ...
    private:
      TpVetEmpresas vetEmp;
      int controlador;
      ...
    };
  ```

  Esta classe armazena dados de um conjunto (vetor) de empresas. Para se fazer uma compra para determinada empresa, primeiro ela deve ser especificada com `localizaEmpresa` (1) e depois a compra é feita com `acumuloValorDaCompra` (2). A variável controlador é inicializada por uma chamada a (1) e usada por (2) para encontrar a empresa que fez a compra, que é

  ```
  vetEmp.v[controlador]
  ```

  O método (2) não é uma operação sobre um vetor de empresas, mas sobre uma **única** empresa. A variável `controlador` está sendo usada como uma variável de comunicação entre duas abstrações diferentes (uma empresa e um vetor de empresas) e não é um dado característico de um vetor de empresas.

- ```
  class Pedido {
    public:
          /* Verifica se o produto com nome <Nome> eh fabricado
             pela X-Tal. Se sim, retorna em <posicao> sua posicao
             no vetor Produtos. Retorna 0 em erro. */
  ```

```
    int  confereNomeProduto( int &posicao, TpNomeProd nome );
        /* Pesquisa a maior encomenda, retornado a posicao dela
           no arquivo */
    int pesquisaMaiorPreco();
    ...
  }
```

Esta classe representa um pedido de compras para uma empresa chamada X-Tal, que vende vários tipos de produtos.

O método `confereNomeProduto` é uma operação sobre o conjunto de produtos da X-Tal, não sobre um pedido.

O método `pesquisaMaiorPreco` é uma operação sobre o arquivo de encomendas, que contém dados de vários pedidos, não sobre um pedido em particular.

Observe que estes métodos provavelmente não usam as variáveis de instância da classe.

- ```
  struct TpMenu {
      x, y : integer;
      ...
    };

    class Menu {
      public:
        int  selecOpcao( int &op );
        void desenhaMoldura( int x1, int y1, int x2, int y2, TpBorda chBorda );
        ...
      private:
        m : TpMenu;
        ...
    };
  ```

  Um menu é formado por uma moldura e diversas opções para escolha, que são colocadas dentro da moldura. O método `selecOpcao` lê uma opção escolhida pelo usuário e a coloca em `op`. Na variável de instância `m` estão os dados dos menu (*strings* das opções, posição na tela, etc).

  `desenhaMoldura` é uma operação sobre uma parte de um menu, que é a moldura, não exatamente sobre um menu. Se `desenhaMoldura` fosse método, não deveria ter parâmetros, já que estes podem ser deduzidos da variável de instância `m`. De fato, este método não utiliza a variável de instância da classe.

- ```
  class Empresa {
    public:
      ...
          // retorna o valor total do pedido
      float calculaTotal( int quant, float preco );
    };
  ```

  O último método é uma operação sobre um pedido, e não sobre a empresa que fez o pedido. Este método não utiliza variáveis de instância da classe.

13 Técnicas de Programação

1. A mais importante de todas as regras de codificação é "mantenha os algoritmos o mais simples possíveis". Não sacrifique legibilidade por eficiência. Usualmente, 5% do código é responsável por 70% a 80% do tempo de execução do programa. Qualquer tipo de otimização que prejudique a legibilidade deve ser feita apenas nestes 5%.

A codificação de um programa pode ser feita de diversas formas. Uma delas é dada pelos itens a seguir.

- Divida o programa em TAD's[7] ou classes.[8] Codifique-os e complete o programa.
- Compile o programa apenas para retirar os erros de sintaxe.
- Rastreie o programa através de uma listagem em papel procurando erros. Esta fase pode ser dividida em duas partes:
  - **(a)** Confira a correção de cada TAD ou classe e verifique se os comentários deles estão corretos.
  - **(b)** Confira se as interfaces entre os TAD ou classes estão corretas. Isto é, se cada TAD utiliza os outros de acordo com suas respectivas especificações. 40% dos erros de *software* são erros de interface entre subsistemas [4].
- Execute o programa com um *debugger* para retirar mais erros.
- Quando o programa estiver funcionando corretamente, reestruture-o para torná-lo mais fácil de ser modificado. Nesta fase, funções específicas podem ser tornadas mais genéricas, procedimentos podem ser divididos em duas ou mais funções, comentários podem ser melhorados, etc.

Esta última parte é fundamental para futuras modificações no *software*. Ela torna alterações mais fáceis, diminui a ocorrência de erros e aumenta a legibilidade do programa. O programador obtém um conhecimento muito melhor do *software*, o que lhe permite descobrir mais erros de codificação.

2. Não misture as funções de entrada e saída de C com aquelas de C++. Elas podem não funcionar corretamente.

Nunca utilize a função `scanf` para ler dados da entrada. Se for necessário utilizar as funções da linguagem C, leia uma linha inteira com `gets` e depois extraia os campos com `sscanf`.

3. Feche todos os arquivos após usá-los. Se você não fechá-los, o sistema operacional os fechará automaticamente. Contudo, deixar arquivos abertos é uma prática que dificulta a modificação (manutenção) dos programas.

4. Desaloque no fim do programa (com `delete`) toda a memória alocada dinamicamente com `new`. Se isto não for feito pelo programa, o será pelo sistema operacional. Porém, deixar memória alocada dificultará futuras modificações no programa. Isto não se aplica a Java, que suporta coleta de lixo.

5. Para ler dados do teclado, utilize o método de fim de arquivo para terminar a leitura de dados. Isto é, o usuário do programa termina a entrada digitando uma tecla especial (como Enter, Esc ou `^Z`) na leitura normal de dados. A alternativa é ler o número de entradas antes de começar a leitura. As duas formas são apresentadas a seguir.

[7] Tipos abstratos de dados.

[8] Há alguns tipos de problemas em que isto é difícil. Esta técnica não deve ser aplicada a eles.

```
const
  MaxChNome = 50,
  MaxNomes  = 10;

typedef
  char TpNome[MaxChNome + 1];

typedef
  TpNome TpArrayNomes[MaxNomes ];

struct TpVetNomes {
  TpArrayNomes v;
  int NumElem;
  };


  void leVetNomes( TpVetNomes vetNomes );
      /* ...
         le primeiro o numero de elementos a serem lidos */‘
  {
    int i;

    do {
      cout << "Digite o numero de elementos a serem lidos\n";
      cin >> vetNomes.NumElem;
      } while ( vetNomes.NumElem < 0 ||  vetNomes.NumElem >= MaxElem );

    for (i = 0; i < vetNomes.NumElem; i++) {
      cout << i << ": ";
      cin >> vetNomes.v[i];
      }
    }



  void eofLeVetNomes( TpVetNomes vetNomes );
      /* ...
         Leitura por fim de arquivo */
  {
    int i;

    cout << "Digite os elementos a serem lidos. Digite ^Z ao fim dos dados\n";
    vetNomes.NumElem = 0;
    do {
      if ( cin >> vetNomes.v[vetNomes.NumElem] )
        vetNomes.NumElem++;
      else
        break;
```

```
  } while ( vetNomes.NumElem != MaxNomes );
}
```

A leitura por fim de arquivo é mais flexível para o usuário, que pode termina-la a qualquer momento. De um modo geral, ela também causa muito menos erros de programação, sendo recomendada por Kernighan [1].

Em leituras de números, não utilize um número especial (como 0 ou 9999) para fim de leitura. Mesmo que atualmente este número não possa ser entrada válida, poderá vir a ser futuramente.

References

[1] Kernighan, B., Plauger, P. J., *The Elements of Programming Style*, MacGraw-Hill, 1974.
Livro clássico sobre programação — usa exemplos nas linguagens FORTRAN e PL/1.

[2] Cannon, L. W. et all, Recommended C Style and Coding Standards, Relatório técnico

[3] Weinberg, Gerald M., *The Psychology of Computer Programming*, van Nostrand Reinhold Company, 1971.

[4] Pressman, Roger S., *Software Engineering – A Practitioner's Approach*, McGraw-Hill, 1982.

A Glossário

- TAD: tipo abstrato de dados.
- Legibilidade. Um programa é legível se é fácil de ler e compreender.
- Reusabilidade. Uma função é reusável se ela pode facilmente ser utilizada no programa em que ela foi definida ou em outros programas.
- Subrotina, rotina, procedimento, função. Utilizamos estas palavras como sinônimas.

B Abreviaturas Para Formação de Identificadores

Abaixo são apresentadas algumas abreviaturas recomendados para a formação de identificadores.[9] Algumas delas possuem significados obvios e não contém nenhuma descrição.

`Abre`	—	Abre alguma coisa, como arquivo, janela.
`Apaga`	—	Elimina ou apaga.
`Arq`	—	Arquivo.
`Atr`	—	Atributo = cor.
`Atrib`	—	Atributo = cor do video.
`Aux`	—	auxiliar.
`Buf`	—	*Buffer*.
`Busca`	—	
`Bytes`	—	
`Calc`	—	Calcula.
`Car`	—	Caracter.
`Ch`	—	Caracter.
`Chama`	—	
`Cmp`	—	Comparação.
`Col`	—	Coluna.
`Coloca`	—	
`Comp`	—	Comprimento.
`Cria`	—	
`Cur`	—	Cursor.
`Cursor`	—	
`Dec`	—	Decrementa.
`Dir`	—	Direita, Diretorio.
`Down`	—	
`Enche`	—	Preenche.
`Entra`	—	
`Erro`	—	
`Esq`	—	Esquerda.
`Exec`	—	Execução.
`Ext`	—	Extensão de arquivo (doc em tese.doc), geralmente.
`Fila`	—	
`File`	—	Arquivo.
`Fun`	—	Função.
`Get`	—	Obtem alguma coisa.
`Ha`	—	Ha, existe.
`Help`	—	Ajuda atraves de janela, em geral.
`Imp`	—	Imprime, impressora.
`Imprime`	—	
`In`	—	Entrada.
`Inc`	—	Incrementa.
`Inic`	—	Inicio.
`Ins`	—	Insere.
`Int`	—	Inteiro.
`Jan`	—	Janela.
`Key`	—	Tecla, teclado.
`Le`	—	Lê.
`Leia`	—	
`Len`	—	*Length*, tamanho.
`Letra`	—	
`Lin`	—	Linha.
`Linha`	—	
`Lst`	—	Lista.
`Max`	—	Maximo.
`Mem`	—	Memoria.
`Menu`	—	
`Move`	—	
`Msg`	—	Mensagem.
`Nao`	—	
`Nome`	—	
`Num`	—	Numero.
`Off`	—	Desligado.
`On`	—	Ligado.
`Opcao`	—	Opção.
`Out`	—	Saida.
`Pal`	—	Palavra.
`Path`	—	
`Pilha`	—	
`Pop`	—	Função de uma pilha para desempilhar elementos.
`Prg`	—	Programa.
`Print`	—	Imprime.
`Proc`	—	Procedimento.
`Push`	—	Função de uma pilha para empilhar elementos.
`Read`	—	
`S`	—	Simples, como em `MsgS` (mensagem simples).
`Saida`	—	
`Sai`	—	Sai (verbo sair).
`Sda`	—	Saida.
`Set`	—	Inicializar variaveis.
`Sim`	—	
`Smp`	—	Simples.
`Sort`	—	

[9] Por questões técnicas, as palavras não estão acentuadas.

`Str` — *String*.
`Tam` — Tamanho.
`Tecla` —
`Tela` —
`Texto` —
`To` — Indica conversão de uma coisa em outra. Ex: `DoubleToStr` — conversão de `double` para *string*.
`Tp` — Tipo. E aconselhavel começar todos os tipos com `Tp`: `TpMatriz`.
`Ult` — Ultima.
`Un` — Indica o sentido oposto da palavra que se segue. Ex: `unAbraArq` — Fecha o arquivo.
`Up` —
`Val` — Valor.
`Vert` — Vertical.
`Vet` — Vetor.
`Write` —
`XY` — Ponto (x,y) do video ou algo semelhante.

Alguns exemplos de identificadores com o respectivo significado são mostrados abaixo.

`MsgErro` — (Emite) Mensagem de erro.
`printStr` — Imprime *string*.
`printVetStr` — Imprime vetor de *strings*.
`SetPilha` — Inicializa variaveis de uma pilha.
`DoubleToStr` — Converte numero `double` para *string*.
`LstInOut` — Lista de entrada e saida.
`LeInt` — Lê numero inteiro.
`TpVetStrMsgHelp` — Tipo vetor de *strings* de mensagens para *help*.
`CursorOff` — Desliga cursor.
`CriaFila` — Cria fila (inicializa seus dados).
`CorMsgAux` — Cor da mensagem auxiliar.
`CmpFun` — Função de comparação.
`MaxChStrDouble` — Maximo de caracteres para um *string* que ira armazenar um numero `double`.

C Regras para Programação

As regras abaixo foram retiradas do livro [1], páginas 159-161. Muitas regras não foram incluídas porque não são auto-explicativas.

1. Escreva claramente — não seja muito inteligente.

2. Diga o que você quer dizer simples e diretamente.

3. Use funções de biblioteca.

4. Evite variáveis temporárias.

5. Escreva claramente — não sacrifique claridade por eficiência.

6. Deixe a máquina fazer o trabalho sujo.

7. Troque expressões repetidas por chamadas a uma função comum.

8. Use parênteses para evitar ambigüidade.

9. Escolha nomes de variáveis que não trarão confusão.

10. Use as boas características da linguagem e evite as ruins.

11. Faça com que cada procedimento execute na mesma ordem em que ele foi escrito — do início para o fim.

12. Utilize as construções de controle de fluxo fundamentais.

13. Escreva primeiro em uma pseudo-linguagem que é fácil de entender, então traduza o algoritmo para outra linguagem.

14. Siga cada decisão com sua ação associada.

15. Escolha as estruturas de dados que fazem o programa simples.

16. Não pare com o primeiro rascunho de algoritmo. Melhore-o.

17. Modularize. Use subrotinas.

18. Cada módulo (subrotina) deve fazer uma coisa bem.

19. Garanta que cada rotina esconda alguma coisa.

20. Permita que os dados estruturem o programa.[10]

[10] Em Orientação a Objetos, são as classes e não os dados que estruturam o programa. Cuidado com esta regra !

21. Não conserte código ruim. Reescreva-o.

22. Escreva e teste um programa grande em peças pequenas.

23. Use procedimentos recursivos para estruturas de dados recursivamente definidas.

24. Teste entrada de dados por validade.

25. Garanta que as entradas não podem violar os limites do programa.

26. Termine entrada por fim-de-arquivo ou algum marcador, não por contador.[11]

27. Identifique entrada ruim e mantenha o processamento normal se possível.

28. Faça entrada fácil de preparar e saída auto-explicativa.

29. Use formatos de entrada uniformes.

30. Localize entrada e saída em subrotinas.

31. Garanta que todas as variáveis são inicializadas antes do uso.

32. Não pare no primeiro erro.

33. Use compiladores com *debugger*.

34. Evite multiplas saídas de um laço (com comando `break` de C, por exemplo).

35. Garanta que seu código funcione quando a entrada para alguns procedimentos é o conjunto vazio.

36. Teste programas em valores limites (máximos e mínimos).

37. Programe defensivamente, testando dados mesmo onde eles não deveriam estar errados.

38. Não compare números em ponto flutuante por igualdade.

39. Faça o programa correto antes de torná-lo rápido.

40. Mantenha-o correto quando você torná-lo rápido.

41. Faça-o claro antes de torná-lo rápido.

42. Não sacrifique claridade por pequenos ganhos em eficiência.

43. Deixe o compilador fazer as otimizações simples.

[11] Veja página 13

44. Não force demasiadamente para reusar código.

45. Tenha certeza que casos especiais são realmente especiais.

46. Mantenha o algoritmo simples para fazer o programa rápido.

47. Meça o programa para descobrir quem está gastando mais tempo de execução antes de fazer otimizações.

48. Garanta que comentários e código concordam.

49. Faça com que cada comentário seja significativo.

50. Não comente código ruim. Reescreva-o.

51. Use nomes de variáveis que significam alguma coisa.

52. Formate o programa para ajudar o leitor a entendê-lo.

53. Indente para mostrar a estrutura lógica do programa.

54. Documente a organização dos dados.

55. Não comente em excesso.