

Assertivas

Baseado em Arndt von Staa

Especificação

- Objetivo dessa aula
 - Discutir como especificar funções
 - Apresentar assertivas de entrada, de saída e estruturais como um instrumento de especificação
 - Introduzir o conceito de desenvolvimento dirigido por contratos

Sumário

- Assertivas, definição
- Exemplos de assertivas
- Assertivas como parte de especificações
- Contratos
- Conteúdo das assertivas
- Exemplos de assertivas estruturais

O que são assertivas?

- Assertivas são **relações** (expressões lógicas) envolvendo dados e estados manipulados
- Assertivas são restrições sobre os dados manipulados pelo programa
- São definidas em vários níveis de abstração
 - funções
 - devem estar satisfeitas em **determinados pontos** do corpo da função
 - usualmente assertivas de **entrada** e assertivas de **saída**
 - **pré** e **pós condições**
 - classes e módulos
 - devem estar satisfeitas ao entrar e ao retornar de funções
 - assertivas **invariantes**, ou assertivas **estruturais**
 - programas
 - devem estar satisfeitas para os **dados persistentes** (arquivos)

Exemplos de assertivas

- $y : x - \varepsilon < y^2 < x + \varepsilon$
 - neste caso o resultado do cálculo da raiz quadrada deve estar limitado a um **erro absoluto** de $\pm\varepsilon$
 - problema: ε depende da **magnitude** de x
 - assumindo um sistema com 6 algarismos significativos
 - se $x = 10^{14}$ então ε deveria ser $> 10^8$
 - se $x = 10^{-14}$ então ε deveria ser $> 10^{-20}$
 - e agora, como resolver isso?
- $y : 1 - \varepsilon < y^2 / x < 1 + \varepsilon$
 - neste caso o resultado do cálculo da raiz quadrada deve estar limitado a um **erro relativo** de $\pm\varepsilon$
 - ε independe da magnitude de x , pode agora determinar a **precisão** (número de algarismos significativos) desejada, por exemplo 10^{-6}
 - cuidado para não solicitar uma **precisão não alcançável**

Exemplos de assertivas

- Em uma lista duplamente encadeada
 1. $\forall pElem \in lista : pElem \rightarrow pAnt \neq NULL \Rightarrow pElem \rightarrow pAnt \rightarrow pProx == pElem$
 - note o uso da **linguagem de programação** com algumas extensões de notação
 - $x \Rightarrow y$ se x então y em que x e y são expressões lógicas
 - é lido: x **implica** y (a verdade de x **implica a** verdade de y)
 - se x for **verdadeiro** e y também for **verdadeiro**, a expressão será **verdadeira**
 - se x for **verdadeiro** e y for **falso**, a expressão será **falsa**
 - se x for **falso**, a expressão será **verdadeira** independentemente de y
 - » na realidade se x for **falso** a expressão passa a ser irrelevante
 2. Outra redação: para todos os elementos *elem* pertencentes a uma *lista duplamente encadeada*, se *elem* possui um antecessor, então o sucessor deste é o próprio *elem*

Exemplos de assertivas

- É dado um *arquivo-A* contendo $n \geq 0$ registros
 - cada registro contém um campo *chave*
 - \forall registros r_i e $r_k \mid r_i, r_k \in \text{arquivo-A} :$
se r_i antecede r_k então $r_i.chave < r_k.chave$
- Isso poderia ser dito de uma forma mais compreensível?
- É dado um *arquivo-A* contendo $n \geq 0$ registros
 - cada registro contém um campo *chave*
 - o arquivo é ordenado em ordem **estritamente** crescente segundo *chave*
 - **estritamente**: sem conter chave repetida
 - por que não pode conter chave repetida?
 - cuidado com sutilezas notacionais!

Exemplos de assertivas

$$M \subseteq A :: \forall \alpha \in M : \exists \delta \in D \mid \mu(\alpha, \delta)$$

- o que quer dizer isso?
- O conjunto *AlunosMatriculados* \subseteq *AlunosRegistrados* é definido:
 $\forall a \in \text{AlunosMatriculados} :$
 $\exists d \in \text{DisciplinasOferecidasSemestre} \mid \text{matriculado}(a, d)$
 - o predicado *matriculado*(*a* , *d*) terá o valor verdadeiro se e somente se *a* estiver cursando a disciplina *d*
- Cada *AlunoMatriculado* estará matriculado em pelo menos uma disciplina oferecida no semestre.
 - qual delas é melhor para uma pessoa com pouca formação?

Um critério fundamental

- Assertivas podem ser utilizadas como especificação
 - projeto baseado em contratos (*design by contract*)
- Neste caso é fundamental
 - comunicação com o cliente ou usuário
- Clientes e usuários não precisam ter formação em computação
 - portanto, terão dificuldade em ler notações formais elaboradas
 - ou seja, a notação matemática talvez não seja a melhor forma de comunicação entre os interessados
- São importantes
 - clareza
 - não ambigüidade
 - precisão de linguagem
 - concisão
 - sintaxe, ortografia e apresentação
 - . . .

Assertiva como parte da especificação

```
/* **** */
* Função: Converter long para ASCII
* Descrição
*   Converte um inteiro long para um string ASCII.
*   O string resultado estará alinhado à esquerda no buffer de dimASCII
*   caracteres fornecido
* Parâmetros
*   dimASCII - número máximo de caracteres do string inclusive
*             o caractere zero terminal.
*   pNumASCII - ponteiro para o espaço que receberá o string.
*              Será truncado à direita caso o string convertido
*              exceda a dimensão limite. O primeiro caractere
*              será '-' se e somente se número < 0
*   Numero - inteiro a ser convertido para string
* Valor retornado
*   veja as declarações das condições de retorno
* Assertiva de entrada
*   pNumASCII != NULL
*   dimensao( *pNumASCII ) >= dimASCII
*   dimASCII >= max( 3 , 2 + log10( abs( Numero ) )
**** */
char * BCD_ConverterLongASCII( int    dimASCII ,
                               char * pNumASCII ,
                               long   Numero   ) ;
```

que tal 12?

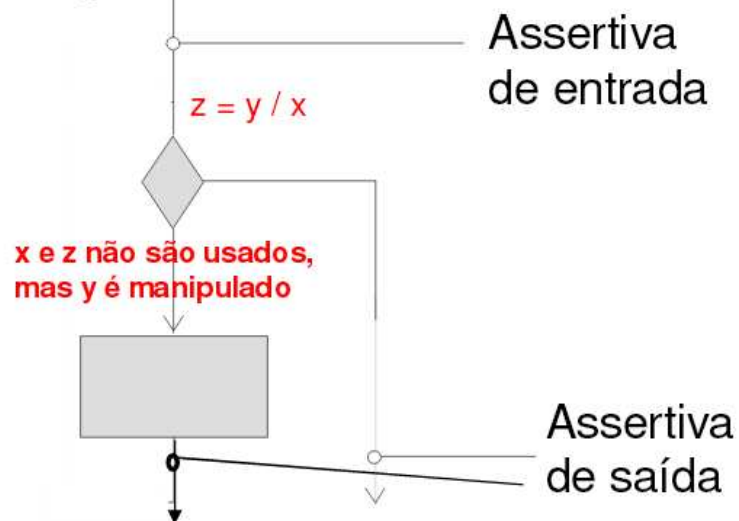
Assertivas de entrada e saída

- delimitam fragmentos de código (blocos de código)
 - desde elementares
 - até o corpo inteiro de uma função tão complexa quanto se queira
- uma função tem vários pontos de saída/retorno dependendo do uso de if's, while's, etc...
 - garantir que todas as possíveis pós-condições estão especificadas corretamente em todos estes pontos
- definem o que se entende por corretude desses fragmentos de código

Assertivas de entrada e saída

- são relações (expressões lógicas) envolvendo dados e estados manipulados pelo algoritmo
- devem estar satisfeitas em determinados pontos do algoritmo

x (divisor) é um dos parâmetros da função



Exemplo

AE: $x > 0$

DIVISÃO: $z = y / x$

AS: $y == x * z$

garantir que y é lida, mas não modificada

Exemplos de assertivas de entrada e saída

- Exemplos de assertivas de entrada

`pTabela` - referencia uma tabela existente

`Simbolo` - é uma seqüência de um ou mais *bytes*

`quaisquer`

`idSimbolo` - é um identificador em relação um para

um com um símbolo existente

Exemplos de assertivas de entrada e saída

- Exemplo de assertiva de saída:

```
Se CondRet == OK
Então
    a tabela pTabela conterá 1 ou mais símbolos
    Se Simbolo já figurava na tabela
    Então
        idSimbolo será o valor associado ao Simbolo já existente
        na tabela
    Senão
        terá sido criado um idSimbolo diferente de todos os
        demais identificadores registrados na tabela
        o par < idSimbolo , Simbolo > terá sido acrescentado à
        tabela pTabela
    FimSe
FimSe
Se CondRet != OK
Então
    Tabela não terá sido alterada
    idSimbolo será igual a NIL_SIMBOLO
FimSe
```

Exemplos de assertivas de função

- Duas formas de implementação :

```
LIS_tpCondRet LIS_ExcluirElementoCorrente( LIS_tppLista pLista ) {  
    tpElemLista * pElem ;  
    // forma 1  
    #ifdef _DEBUG  
        assert( pLista != NULL ) ; // retirado se _DEBUG não definido  
    #endif  
    // forma 2  
    if ( pLista->pElemCorr == NULL ) // sempre verifica  
    {  
        return LIS_CondRetListaVazia ;  
    } /* if */  
    pElem = pLista->pElemCorr ;  
}
```

Assertiva executável

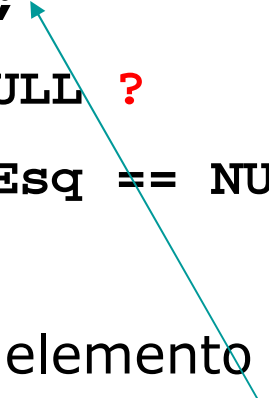
```
/* Verificar encadeamento de elemento de lista com anterior */
if ( pElemento != pOrigemLista )
{
    if ( pElemento->pAnt != NULL )
    {
        if ( pElemento->pAnt->pProx != pElemento )
        {
            ExibirErro( "Encadeamento antes está errado." ) ;
        } /* if */
    } else {
        ExibirErro( "pAnt == NULL, mas não é a origem." ) ;
    } /* if */
} else
{
    if ( pElemento->pAnt != NULL )
    {
        ExibirErro( "pAnt != NULL em origem" ) ;
    } /* if */
} /* if */
```

- Note que `ExibirErro` não pode retornar

Assertivas em C

- Considere uma lista com cabeça

```
assert( pLista != NULL ) ;  
assert( pLista->pOrg != NULL ?  
        pLista->pOrg->pEsq == NULL : TRUE ) ;
```



- Considere um determinado elemento da lista

```
assert( pElem != NULL ) ;  
assert( pElem->pDir != NULL ? pElem->pDir->pEsq ==  
        pElem : TRUE ) ;  
assert( pElem->pEsq != NULL ? pElem->pEsq->pDir ==  
        pElem : TRUE ) ;
```

Deve vir antes dos demais. Por que?

Assertivas como comentários de argumentação

```
/* Assertiva de entrada
* vtElem    - é o vetor a ser ordenado
* Inferior  - é o índice limite inferior da região a ser ordenada
* Superior  - é o índice limite superior da região a ser ordenada
*/

void MeuQuicksort( int * vtElem , int Inferior , int Superior )
{
    int Pivot ;
    if ( Inferior < Superior )
    {
        Pivot = Partition( vtElem , Inferior , Superior ) ;
        /* vtElem[ Pivot ] contém o valor ordenado final
        * para todos i < Pivot : vtElem[ i ] <= vtElemPivot
        * para todos i > Pivot : vtElem[ i ] > vtElemPivot
        */
        MeuQuicksort( vtElem , Inferior , Pivot - 1 ) ;
        /* Sub-região até Pivot inclusive está ordenada */
        MeuQuicksort( vtElem , Pivot + 1 , Superior ) ;
        /* Toda a região de Inferior a Superior está ordenada */
    } /* if */
} /* Quicksort */
```

Quando utilizar assertivas?

- Podem ser utilizadas
 - ao **especificar** funções
 - desenvolvimento **dirigido por contratos**
 - *contract driven development*
 - visam desenvolver funções corretas por construção
 - ao **argumentar** a corretude de programas
 - estabelecem os **predicados** utilizados na argumentação
 - ao **instrumentar** programas
 - assertivas executáveis **monitoram** o funcionamento do programa
 - ao **testar** programas
 - apóiam a **diagnose** da falha visando encontrar o defeito causador (teste-diagnóstico)
 - ao **depurar** (debugging) programas
 - **facilitam** a completa e correta remoção do defeito

Exemplo de contrato na especificação

```
/*    AssertivaEntrada
*        !Tabela_Cheia( ) ;
*        !ExisteChave( Chave ) ;
*
*    AssertivaSaida
*        numSimbolos() == Entrada.numSimbolos() + 1 ;
*        ExisteChave( Chave ) ;
*        Igual( Valor , ObterValor( Chave )) ;
*/
```

```
InserirElemento( tpChave Chave , tpValor Valor )
```

Regras para contratos: confie

- Cabe ao **cliente** assegurar que o **contrato de entrada vale antes** de chamar a função (assertiva de entrada)
 - por isso precisa figurar na especificação
- Cabe ao **servidor** assegurar que o **contrato de saída vale ao retornar** da função (assertiva de saída)
 - tem que valer **para todos os return**

Regras para contratos: desconfie

- Sempre que a fonte de dados for não confiável, os contratos nem sempre são assegurados
- Exemplos de causas para não confiar
 - interfaces com humanos
 - uso de software (bibliotecas) de procedência duvidosa
 - dados vindos de sistemas de qualidade não confiável
 - . . .
- Cabe ao servidor **verificar o contrato de entrada**
- Cabe ao cliente **verificar o contrato de saída**

Regras para contratos

- Ao verificar o contrato de entrada
 - o contrato **especificado** continua o mesmo
 - o cliente estará ciente do que deve ser assegurado
 - o contrato **implementado** passa a ser: “**vale qualquer coisa**”
 - o servidor não acredita no cliente
 - em compensação, o contrato de saída especificado precisa conter a especificação **do que acontecerá** se o contrato de entrada não vale
- Exemplo: RaizQuadrada: $RQ(x)$
 - Entrada
 - vale qualquer x
 - Saída
 - se $x \geq 0 \Rightarrow RQ = y : 1 - \epsilon < y^2/x < 1 + \epsilon$
 - se $x < 0 \Rightarrow RQ = -y : 1 - \epsilon < y^2/(-x) < 1 + \epsilon$

Regras para contratos

- **Assertivas estruturais** (assertivas invariantes) definem as condições a serem satisfeitas pelos dados e estados que constituem o módulo (estrutura de dados)
 - valem somente quando a estrutura não estiver sendo alterada
 - cuidado com **multi-threading**: assegure sincronização ao utilizar uma função que possa modificar uma estrutura compartilhada
- Todas as funções do módulo
 - **assumem a validade** das assertivas estruturais **ao entrar**
 - exceto, caso exista, a função `ZZZ_Reset()`
 - **devem assegurar** a validade delas **ao sair**

Regras para contratos

- Cabe à **função de inicialização** do módulo (construtor da classe) assegurar que as **assertivas estruturais valem ao iniciar** a execução do módulo
 - controles inicializados por declaração
 - função `ZZZ_Reset()`
 - deve ser chamada **antes** de utilizar o módulo pela **primeira vez**
 - pode ser chamada para reinicializar
 - função `ZZZ_Criar()` construtor
 - cria uma instância nova
- Cabe ao **conjunto de funções** do módulo assegurar que as **assertivas estruturais sempre valem ao retornar**
 - se inicializadas corretamente e sempre valem ao retornar, sempre valerão ao chamar

Conteúdo de uma assertiva de entrada

- Assume-se a validade da assertiva estrutural, logo não precisa estar presente na especificação de entrada de cada função
- Devem aparecer nas expressões lógicas das assertivas de entrada
 - todos os parâmetros de entrada
 - são dados de entrada os dados do escopo externo à função que podem ser acessados antes de serem alterados
 - todas as variáveis globais de entrada
 - evite variáveis globais que servem a somente uma função
 - ao invés delas use variáveis locais `static`
 - todos os arquivos de entrada
 - sugestão: defina assertivas invariantes para os arquivos

Conteúdo de uma assertiva de saída

- Assume-se a validade da assertiva estrutural, logo não precisa estar presente na especificação de saída de cada função
- Devem aparecer nas expressões lógicas das assertivas de saída
 - o valor retornado pela função
 - todos os parâmetros de saída (parâmetros por referência)
 - são dados de saída os dados do escopo externo à função que podem ser alterados ao executar a função
 - são dados atualizados dados que são ao mesmo tempo de entrada e de saída
 - todas as variáveis globais de saída
 - evite variáveis globais que servem a somente uma função
 - ao invés delas use variáveis locais `static`
 - todos os arquivos de saída
 - sugestão defina assertivas invariantes para os arquivos

Exemplo assertiva estrutural

- Lista duplamente encadeada

$\forall elem \in lista : elem \rightarrow pAnt \neq NULL \Rightarrow$

$elem \rightarrow pAnt \rightarrow pProx == elem$

$\forall elem \in lista : elem \rightarrow pProx \neq NULL \Rightarrow$

$elem \rightarrow pProx \rightarrow pAnt == elem$

- De forma **fatorada**:

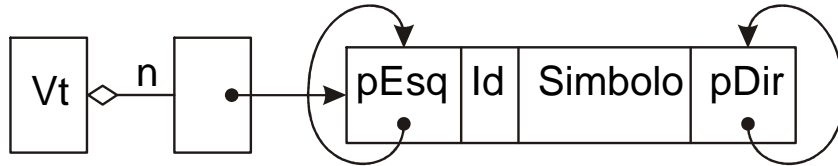
$\forall elem \in lista :$

$elem \rightarrow pAnt \neq NULL \Rightarrow elem \rightarrow pAnt \rightarrow pProx == elem$

$elem \rightarrow pProx \neq NULL \Rightarrow elem \rightarrow pProx \rightarrow pAnt == elem$

Exemplo assertiva estrutural

Modelo



Exemplo

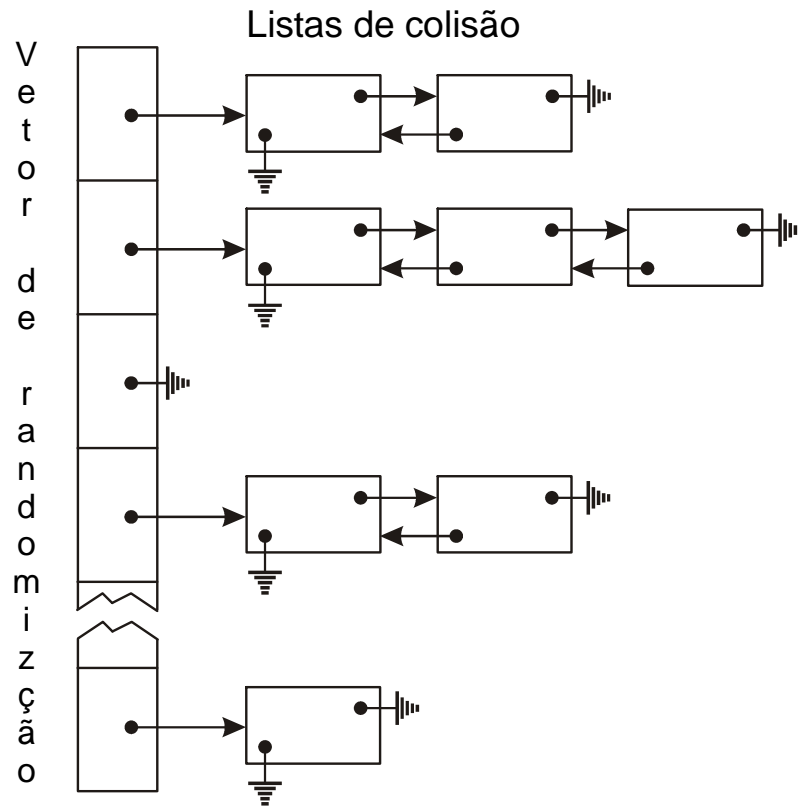
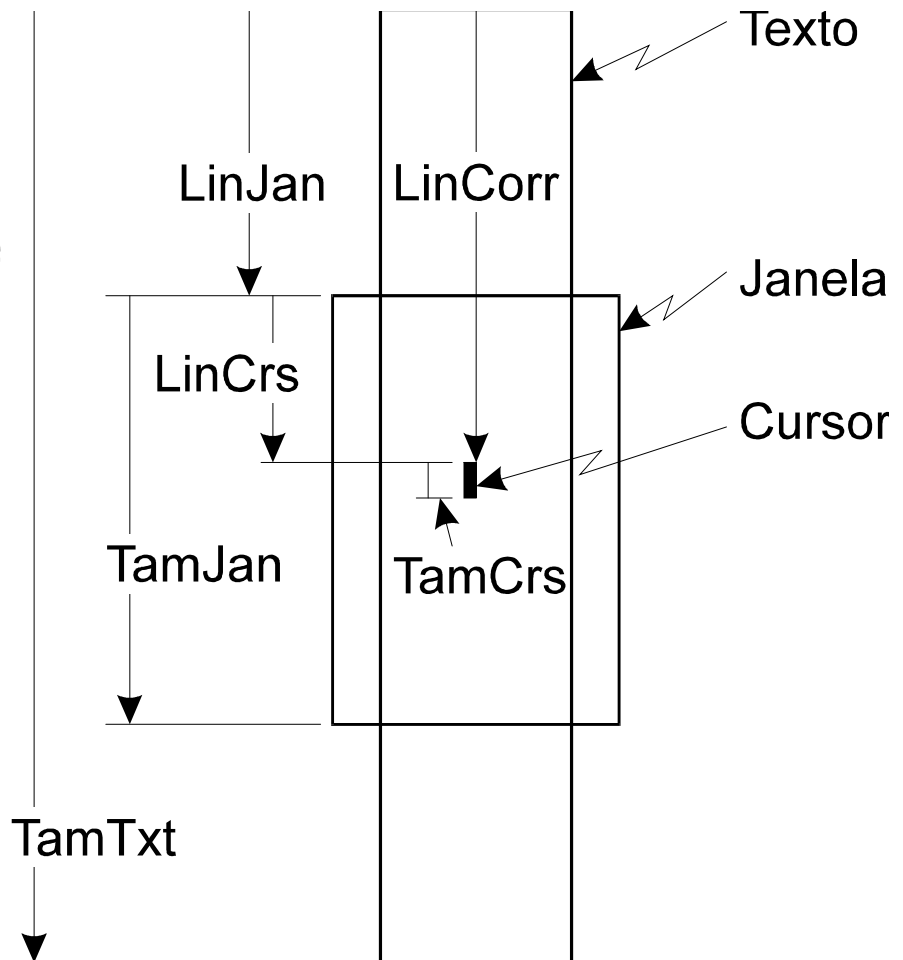


Tabela hash (randomização)

- $n \geq 1$
- Para todos os Simbolos
 - {
 - $0 \leq \text{ObterInxHash}(\text{Simbolo}) < n$
 - }
- Para todos os símbolos da lista $\text{inx} : 0 \leq \text{inx} < n$
 - {
 - $\text{inx} = \text{ObterInxHash}(\text{Simbolo})$
 - }
- cada lista é duplamente encadeada.

Exemplo assertiva estrutural 1/3

- Controle da janela de editor de textos simples
- $TamTxt$, $LinJan$ e $LinCorr$ são medidos em número de linhas
- $0 \leq TamTxt$ é o número total de linhas do texto
- $0 \leq LinJan \leq TamTxt$ é o índice da primeira linha visível na janela
- $0 \leq LinCorr \leq TamTxt + 1$ é o índice da linha que está sendo manipulada
(\leq ou $< e + 1$)
- $TamJan$, $TamCrs$ e $LinCrs$ são medidos em pixel de vídeo
- $1 < TamCrs$ é o tamanho da linha em pixel (constante)



Exemplo assertiva estrutural 2/3

- $TamCrs \leq TamJan$ é o tamanho da janela
- $0 \leq LinCrs \leq TamJan - TamCrs$ é o pixel inicial da linha corrente na janela
 - o cursor de edição sempre estará contido dentro da janela
- $LinJan \leq LinCorr < LinJan + (TamJan / TamCrs)$
 - a linha corrente encontra-se em uma linha contida na janela
- $0 \leq LinJan \leq \max(0 , TamTxt - LinJan / TamCrs + 1)$
 - a origem da janela é posicionada de modo a maximizar a porção de texto que é exibida.

Esquema do código do editor 3/3

- Observação: caso o editor assegure a validade destas relações
 - o **texto focal** no entorno de *LinCorr* estará sempre visível
 - pode-se dissociar as funções que controlam a exibição das funções que realizam a alteração do texto e/ou movimentam o cursor

```
enquanto DeveEditar( )
{
    realiza uma ação de edição ou movimentação,
           possivelmente alterando os valores de LinCorr,
           TamTxt e TamJan.
    se uma ou mais das assertivas não forem válidas
    {
        recalcula LinJan para assegurar a validade
        redesenha a janela
    }
    recalcula o valor de LinCrs    // sempre na janela
    reposiciona o cursor           // será sempre visível
}
```


FIM