12. Leiaute e Estilo

Este capítulo trata de um aspecto estético da programação de computadores: o leiaute do código-fonte. O desfrute visual e intelectual de um código bem formatado é um prazer que poucos não-programadores podem apreciar. Mas os programadores que se orgulham de seu trabalho sentem uma grande satisfação artística com o aprimoramento da estrutura visual de seus códigos.

As técnicas deste capítulo não afetam a velocidade de execução nem o uso de memória ou outros aspectos de um programa que sejam visíveis fora dele. Eles afetam a facilidade de entender o código, examiná-lo e revisá-lo meses depois de você tê-lo escrito. Eles também afetam a facilidade com que outras pessoas o leem, entendem e modificam, quando você for carta fora do baralho.

Este capítulo está repleto de detalhes minuciosos aos quais as pessoas se referem quando falam sobre "atenção aos detalhes". Durante a vida de um projeto, a atenção a esses detalhes faz diferença na qualidade inicial e na manutenibilidade final do código que você escreve. Tais detalhes são de importância do processo de codificação que torna-se difícil alterá-los eficientemente mais tarde. Já que tratar desses detalhes é mesmo inevitável, que isso seja feito durante a construção inicial. Se você estiver trabalhando em um projeto em equipe, faça todos lerem este capítulo e concordarem com um estilo único, antes de começar a codificar.

Talvez você não esteja de acordo com tudo o que ler aqui, mas meu objetivo não é tanto obter sua concordância, mas convencê-lo de que os problemas envolvidos no estilo de formatação devem ser levados a sério.

12.1 Fundamentos do leiaute

Esta seção explica a teoria do bom leiaute. O restante do capítulo explica a prática.

Extremos do leiaute

Considere a rotina mostrada na Listagem 1:

Listagem 1: Primeiro exemplo de leiaute em Java.

```
/* Usa a técnica de ordenação por inserção para classificar o array "data" em ordem ascendente. Esta rotina presume que data[ firstElement ] não é o primeiro elemento em data e que data [ firstElement - 1 ] pode ser acessado. */ public void InsertionSort( int[ ]data, int firstElement, int lastElement ) { /* Substitui o elemento no limite inferior por um elemento que com certeza é o primeiro em uma lista ordenada. */ int lowerBoundary <= data[ firstElement-1 ]; data[ firstElement-1 ] - SORT_MIN; /* Os elementos nas posições firstElement a sortBoundary-1 são sempre ordenados. Em cada passagem pelo loop, sortBoundary é aumentado e o elemento que está na posição do novo sortBoundary provavelmente não está em seu lugar ordenado no array; assim, ele é inserido no lugar correto, entre firstElement e sortBoundary. */ for ( int sortBoundary = firstElement+1; sortBoundary <= lastElement; sortBoundary++ ) { int insertVal = data [ sortBoundary ]; int insertPos = sortBoundary; while (insertVal < data[ insertPos-1 ] ) { data[ insertPos ] = data [ insertPos-1 ]; insertPos [ ] =  insertPos-1; } data[ insertPos ] = insertVal; } /* Substitui o elemento do limite inferior original */ data[ firstElement-1 ] = lowerBoundary; }
```

A rotina está sintaticamente correta. Ela está totalmente comentada, tem bons

nomes de variável e a lógica é clara. Se você não acredita nisso, leia-a e tente encontrar um erro! O que a rotina não tem é um bom leiaute. Esse é um exemplo extremo, tendendo para o "infinito negativo" na escala "ruim-até-bom" de leiaute. A Listagem 2 é um exemplo menos extremo:

Listagem 2: Segundo exemplo de leiaute em Java.

```
/* Usa a técnica de ordenação por inserção para classificar o array "data"
em ordem ascendente. Esta rotina presume que data[ firstElement ] não
é o primeiro elemento em data e que data[ firstElement-1 ] pode ser
acessado. */ public void InsertionSort( int [ ] data, int firstElement,
int lastElement ) {
/* Substitui o elemento no limite inferior por um elemento que com certeza
é o primeiro em uma lista ordenada. */ int lowerBoundary = data
[firstElement-1]; data[ firstElement-1 ] = SORT_MIN;
/* Os elementos nas posições firstElement a sortBoundary-1 são sempre
ordenados. Em cada passagem pelo loop, sortBoundary é aumentado e o
elemento que está na posição do novo sortBoundary provavelmente não está
em seu lugar ordenado no array, assim, ele é inserido no lugar correto
entre firstElement e sortBoundary. */
for (
int sortBoundary = firstElement+1;
sortBoundary <= lastElement;
sortBoundary++
) {
int insertVal = data [ sortBoundary ] ;
int insertPos = sortBoundary;
while ( insertVal < data [ insertPos-1 ] ) {
data[ insertPos ] = data [ insertPos-1 ];
insertPos = insertPos-1;
}
data[ insertPos ] = insertVal;
}
/* Substitui o elemento do limite inferior original */
data [ firstElement-1 ] = lowerBoundary;
}
```

Esse código é o mesmo da Listagem 1. Embora a maioria das pessoas concorde que o leiaute do código é muito melhor do que o do primeiro exemplo, o código ainda não está muito legível. O leiaute ainda está amontoado e não oferece nenhuma pista da organização lógica da rotina. Ele está próximo de 0 na escala "ruim-até-bom" de leiaute. O primeiro exemplo foi inventado, mas o segundo não é totalmente incomum. Tenho visto programas de milhares de linhas com um leiaute pelo menos tão ruim quanto esse. Com ausência de documentação e nomes de variável ruins, a legibilidade global acabava sendo pior do que nesse exemplo. Este código é formatado para o computador; não há nenhuma evidência de que o autor esperasse que ele fosse lido por seres humanos. A Listagem 3 é um aprimoramento.

Esse leiaute da rotina é um forte positivo na escala "ruim-até-bom" de leiaute. Agora, a rotina está disposta de acordo com os princípios que serão explicados ao longo de todo este capítulo. A rotina se tornou muito mais legível e o esforço feito na documentação e nos bons nomes de variável agora é evidente. Os nomes de variável eram tão bons quanto nos exemplos anteriores, mas o leiaute era tão ruim que eles em nada ajudavam.

A única diferença entre este exemplo e os dois primeiros é o uso do espaço em branco - o código e os comentários são exatamente os mesmos. O espaço em branco serve apenas para os leitores humanos - seu computador poderia interpretar qualquer um dos

três trechos com igual facilidade. Mas você não deve sentir-se desconfortável, caso não possa se igualar ao seu computador!

Listagem 3 Terceiro exemplo de leiaute em Java.

```
/* Usa a técnica de ordenação por inserção para classificar o array "data"
em ordem ascendente. Esta rotina presume que data[ firstElement ] não é
o primeiro elemento em data e que data[ firstElement-1 ] pode ser acessado.
*/
public void InsertionSort(int[ ]data,int firstElement,int lastElement ){
   // Substitui o elemento no limite inferior por um elemento que com
   // certeza é o primeiro em uma lista ordenada.
   int lowerBoundary = data [firstElement-1];
   data[ firstElement-1 ] = SORT_MIN;

   /* Os elementos nas posições firstElement a sortBoundary-1 são sempre
   ordenados. Em cada passagem pelo loop, sortBoundary é aumentado e o
   elemento que está na posição do novo sortBoundary provavelmente não
   está em seu lugar ordenado no array, assim, ele é inserido no lugar
   correto entre firstElement e sortBoundary. */
   for(int sortBoundary = firstElement+1;sortBoundary <= lastElement;
     sortBoundary++) {
     int insertVal = data [ sortBoundary ] ;
     int insertPos = sortBoundary;
     while ( insertVal < data [ insertPos-1 ] ) {
       data[ insertPos ] = data [ insertPos-1 ];
       insertPos = insertPos-1;
     }
     data[ insertPos ] = insertVal;
   }
   // Substitui o elemento do limite inferior original */
   data [ firstElement-1 ] = lowerBoundary;
}
```

O Teorema Fundamental da Formatação

O Teorema Fundamental da Formatação estabelece que um bom leiaute visual mostra a estrutura lógica de um programa.

Dar boa aparência para o código é importante, porém mais importante ainda é mostrar a estrutura do código. Se uma técnica mostra melhor a estrutura e outra realça a aparência, use aquela que privilegia a estrutura. Este capítulo apresenta numerosos exemplos de estilos de formatação que têm boa aparência, mas que representam mal a organização lógica do código. Na prática, priorizar a representação lógica normalmente não gera um código feio - a menos que a lógica do código seja feia. As técnicas que fazem um bom código ter boa aparência e um código ruim ter má aparência são mais úteis do que as técnicas que fazem todo código ter boa aparência.

Interpretações de um programa por seres humanos e por computadores

O leiaute é um indício forte da estrutura de um programa. Enquanto o computador pode se preocupar exclusivamente com as chaves ou com as instruções *begin* e *end*, um leitor humano está apto a extrair indícios a partir da apresentação visual do código. Considere o trecho de código da Listagem 4, no qual o esquema de indentação faz parecer, para um ser programadores humano, como se três instruções fossem executadas cada vez que o loop é executado.

Listagem 4: Exemplo em Java de leiaute que conta histórias diferentes para seres humanos e computadores.

```
// troca os elementos da esquerda e da direita para o array inteiro
for ( i = 0; i < MAX_ELEMENTS; i++ )
   leftElement = left[ i ];
   left [ i ] = right[ i ] ;
   right [ i ] = leftElement;
```

Se o código não estiver entre chaves, o compilador executará a primeira instrução MAX_ELEMENTS vezes e, a segunda e terceira instruções, uma vez cada. A indentação torna claro para você e para mim que o autor do código queria que as três instruções fossem executadas juntas e pretendia colocar chaves em torno delas. Isso não ficará claro para o computador. A Listagem 5 é outro exemplo:

Listagem 5: Outro exemplo em Java de leiaute que conta histórias diferentes para seres humanos e computadores.

```
x = 3+4 * 2+7;
```

Um leitor humano desse código ficaria inclinado a interpretar a instrução como x recebendo o valor (3+4) * (2+7), ou 63. O computador ignorará o espaço em branco e obedecerá as regras da precedência, interpretando a expressão como 3 + (4*2) + 7, ou 18. A questão é que um bom esquema de leiaute faria a estrutura visual de um programa corresponder à estrutura lógica ou contar para o ser humano a mesma história que conta para o computador.

Quanto vale um bom leiaute?

> *Nossos estudos corroboram a afirmação de que o conhecimento de planos e regras do discurso da programação podem ter um impacto significativo sobre a compreensão do programa. Alguns autores também identificam o que chamaríamos de regras do discurso. Nossos resultados empíricos reforçam essas regras: não apenas uma questão de estética o fato de que os programas devem ser escritos em um estilo específico. Em vez disso, há uma base psicológica para a escrita de programas de uma maneira convencional: os programadores têm fortes expectativas de que outros programadores sigam essas regras do discurso. Quando as regras são violadas, a utilidade produzida pelas expectativas que os programadores construíram com o passar do tempo é efetivamente anulada. Os resultados das experiências com alunos de programação iniciantes e avançados, e com programadores profissionais, descritas neste artigo fornecem uma prova clara dessas afirmativas. - Elliot Soloway e Kate Ehrlich*

No leiaute, talvez mais do que em qualquer outro aspecto da programação, aparece a diferença entre comunicar-se com o computador e comunicar-se com leitores humanos. A parte menos importante do trabalho de programação é escrever um programa de modo que o computador possa lê-lo; a parte mais importante é escrevê-lo de modo que outros seres humanos possam lê-lo.

Em seu artigo clássico “Perception in Chess”, Chase e Simon relatam um estudo que comparou a capacidade de especialistas e iniciantes de lembrarem das posições das peças no jogo de xadrez (1973). Quando as peças eram organizadas no tabuleiro conforme poderiam ficar durante um jogo, a memória dos especialistas era bem superior à dos iniciantes. Quando as peças eram organizadas aleatoriamente, havia pouca diferença entre a memória dos especialistas e dos iniciantes. A interpretação tradicional desse resultado é que a memória de um especialista não é inerentemente melhor do que a de um iniciante, mas o especialista tem uma estrutura de conhecimento que o ajuda a se lembrar de tipos de informação específicos. Quando uma nova informação

corresponde à estrutura do conhecimento - neste caso, o posicionamento coerente das peças de xadrez –, o especialista pode lembrar-se dela facilmente. Quando a nova informação não corresponde a uma estrutura de conhecimento - as peças de xadrez são posicionadas aleatoriamente –, o especialista não consegue lembrar-se dela melhor do que o iniciante.

Leiaute como religião

A importância, para a compreensão e para a memorização, de estruturar o ambiente de alguém de uma maneira familiar, tem levado alguns pesquisadores a formular a hipótese de que o leiaute pode prejudicar a capacidade de um especialista de ler um programa, caso seja diferente do esquema que ele utiliza (Sheil 1981, Soloway e Ehrlich 1984). Essa possibilidade, combinada com o fato de que o leiaute é um exercício tanto de estética como de lógica, significa que os debates sobre a formatação de um programa frequentemente parecem mais guerras religiosas do que discussões filosóficas.

Grosso modo, é claro que algumas formas de leiaute são melhores do que outras. Os leiautes sucessivamente melhores do mesmo código, que aparecem no início deste capítulo, tornaram isso evidente. Os bons programadores devem ter a mente aberta a respeito de suas práticas de leiaute e aceitar as práticas que se mostrarem melhores do que aquelas com as quais estão acostumados a usar, mesmo que o ajuste a um novo método resulte em certo desconforto inicial.

Objetivos do bom leiaute

Muitas decisões sobre os detalhes do leiaute são uma questão subjetiva de estética; frequentemente, você pode atingir o mesmo objetivo de muitas maneiras. Você pode tornar os debates sobre questões subjetivas menos subjetivos se especificar explicitamente os critérios de suas preferências. Explicitamente, então, um bom esquema de leiaute deve:

Representar precisamente a estrutura lógica do código Aí está o Teorema Fundamental da Formatação novamente: o principal objetivo do bom leiaute é mostrar a estrutura lógica do código. Normalmente, os programadores usam indentação e outro espaço em branco para mostrar a estrutura lógica.

Representar consistentemente a estrutura lógica do código Alguns estilos de leiaute apresentam regras com tantas exceções que é difícil segui-las rigorosamente. Um bom estilo deve se aplicar à maioria dos casos.

Melhorar a legibilidade Uma estratégia de indentação que seja lógica, mas torne o código mais difícil de ler, é inútil. Um esquema de leiaute que exija espaços somente onde são obrigados pelo compilador é lógico, mas não é legível. Um bom esquema de leiaute torna o código mais fácil de ler.

Resistir às modificações Os melhores esquemas de leiaute se sustentam bem em caso de modificação do código. A modificação de uma linha de código não deve levar à modificação de várias outras linhas.

Além desses critérios, muitas vezes também é importante minimizar o número de linhas de código necessárias para implementar uma instrução simples ou um bloco de instruções.

Como usar os objetivos do leiaute

Você pode usar os critérios de um bom esquema de leiaute para fundamentar uma discussão sobre o leiaute, de forma que sejam esclarecidas as subjetividades da preferência de um estilo em detrimento de outro.

Ponderar os critérios de diferentes maneiras pode levar a diferentes conclusões. Por exemplo, se você está bastante seguro de que minimizar o número de linhas usadas na tela é importante - talvez porque você tenha uma tela pequena no computador –, poderá criticar um estilo simplesmente por ele utilizar duas linhas a mais para uma lista de parâmetros de uma rotina do que um outro.

12.2 Técnicas de leiaute

Você pode obter um bom leiaute usando algumas técnicas de leiaute de várias maneiras diferentes. Esta seção descreve cada uma delas.

Espaço em branco

Useespaçosembrancoparamelhoraralegibilidade. O espaço em branco, incluindo os espaços propriamente ditos, tabulações, quebras de linha e linhas em branco, é a principal ferramenta disponível para mostrar a estrutura de um programa.

Você não imaginaria escrever um livro sem nenhum espaço entre as palavras, nenhuma quebra de parágrafo e nenhuma divisão nos capítulos. Tal livro poderia ser legível de capa a capa, mas seria praticamente impossível lê-lo superficialmente para entender uma linha de pensamento ou encontrar uma passagem importante. Talvez o mais importante, o leiaute do livro não mostraria ao leitor como o autor pretendia organizar as informações. A organização do autor é um indício importante sobre a organização lógica do assunto.

A divisão de um livro em capítulos, parágrafos e sentenças mostra ao leitor como organizar um assunto mentalmente. Se a organização não é evidente, o leitor precisa estabelecê-la, o que impõe para ele uma carga muito maior e cria a possibilidade de que ele nunca possa descobrir como o assunto está organizado.

As informações contidas em um programa são mais densas do que as informações contidas na maioria dos livros. Enquanto você pode ler e entender uma página de um livro em um ou dois minutos, a maioria dos programadores não consegue ler e entender uma listagem de programa simples com essa rapidez toda. Um programa deve oferecer mais indícios organizacionais do que um livro, e não menos.

Agrupamento Do outro lado do espelho, o espaço em branco é um agrupamento, garantindo que instruções relacionadas sejam agrupadas.

Na escrita, os pensamentos são agrupados em parágrafos. Um parágrafo bem escrito contém apenas sentenças relacionadas a um determinado pensamento. Ele não deve conter sentenças díspares. Analogamente, um parágrafo de código deve conter instruções que executam uma única tarefa e que estejam relacionadas entre si.

Linhas em branco Assim como é importante agrupar instruções relacionadas, também é importante separar entre si as instruções não-relacionadas. O início de um novo parágrafo em português é identificado com indentação ou com uma linha em branco. O início de um novo parágrafo de código deve ser identificado com uma linha em branco.

Usar linhas em branco é uma maneira de indicar como um programa está organizado. Você pode usá-las para dividir grupos de instruções relacionadas em parágrafos, para separar rotinas umas das outras e para destacar comentários.

Embora esta estatística em particular possa ser difícil de por em prática, um estudo descobriu que o número ideal de linhas em branco em um programa é de cerca de 8 a 16%. Acima de 16%, o tempo de depuração aumenta substancialmente (1990).

Indentação Use indentação para mostrar a estrutura lógica de um programa. Como regra, você deve indentar as instruções sob aquela a que as demais estão logicamente subordinadas.

A indentação está correlacionada com o aumento da compreensão do programador. O artigo "Program Indentation and Comprehensibility" (Indentação de programas e a capacidade de compreensão) relatou que vários estudos descobriram correlações entre indentação e uma maior compreensão (Miaria et al. 1983). Em um teste de compreensão, quando os programas tinham um esquema de indentação de dois a quatro espaços, as pessoas alcançaram uma pontuação 20 a 30% mais alta do que quando os programas não tinham nenhuma indentação.

O mesmo estudo revelou que era importante não colocar ênfase em excesso nem escassamente na estrutura lógica de um programa. Os piores resultados em termos de compreensão foram obtidos em programas que não tinham nenhuma indentação. Os segundos piores resultados foram obtidos em programas que usavam indentação de seis espaços. O estudo concluiu que uma indentação de dois a quatro espaços era a ideal. É interessante notar que muitas pessoas que participaram da experiência concluíram que uma indentação de seis espaços era mais fácil de usar do que as indentações menores, mesmo sendo piores os seus resultados, isso provavelmente aconteceu porque a indentação de seis espaços parece agradável. Mas, independentemente de quanto pareça boa, a indentação de seis espaços se mostra menos legível. Esse é um exemplo de choque entre o apelo estético e a legibilidade.

Parênteses

Use mais parênteses do que você crê que precisa. Use parênteses para tornar mais claras as expressões que envolvam mais do que dois termos. Eles podem até não ser necessários, mas aumentam a clareza e não custam nada a você. Por exemplo, como as expressões a seguir são avaliadas?

```
Versão em C++: 12 + 4 % 3 * 7 / 8
Versão em Microsoft Visual Basic: 12 + 4 mod 3 * 7 / 8
```

A questão principal é: você teve que pensar sobre como as expressões foram avaliadas? Você pode ter confiança em sua resposta sem verificar algumas referências? Mesmo os programadores experientes não respondem com segurança e é por isso que você deve usar parênteses quando houver dúvida sobre como uma expressão é avaliada.

12.3 Estilos de leiaute

A maioria dos problemas de leiaute está relacionada com a disposição de blocos, os grupos de instruções que ficam abaixo de instruções de controle. Um bloco é colocado entre chaves ou palavras-chave: { e } em C++ e Java, if-then-endif em Visual Basic e outras estruturas semelhantes, em outras linguagens. Para simplificar, grande parte desta abordagem usa *begin* e *end* genericamente, supondo que você saiba como ela se aplica às chaves em C++ e Java ou a outros mecanismos de bloco de outras linguagens. As seções a seguir descrevem quatro estilos gerais de leiaute:

- Blocos puros
- Simulação de blocos puros
- Uso de pares begin-end (chaves) para designar os limites do bloco
- Leiaute de fim de linha

Blocos puros

Grande parte da controvérsia sobre o leiaute deriva da falta de jeito inerente às linguagens de programação mais populares. Uma linguagem bem projetada tem estruturas de bloco claras, que servem como um estilo de indentação natural. Em Visual Basic, por exemplo, cada construção de controle tem sua própria terminação e você não pode usar uma construção de controle sem usar a terminação. O código é colocado em blocos naturalmente. Alguns exemplos em Visual Basic aparecem nas listagens 6, 7 e 8:

Listagem 6: Exemplo em Visual Basic de um bloco *if* puro.

```
If pixelColor = Color_Red Then
   instrução1
   instrução2
   ...
End If
```

Listagem 7: Exemplo em Visual Basic de um bloco *while* puro.

```
While pixelColor = Color_Red
   instrução1
   instrução2
   ...
Wend
```

Listagem 8: Exemplo em Visual Basic de um bloco *case* puro.

```
Select Case pixelColor
   Case Color_Red
     instrução1
     instrução2
     ...
   Case Color_Green
     instrução3
     instrução4
     ...
   Case Else
     instrução5
     instrução6
     ...
End Select
```

Uma construção de controle em Visual Basic tem sempre uma instrução inicial - nos exemplos, *If-Then, While e Select-Case* - e tem sempre uma instrução End correspondente. Indentar o interior da estrutura não é uma prática controversa e as opções de alinhamento das outras palavras-chave são um tanto limitadas. A Listagem 9 é uma representação abstrata de como esse tipo de formatação funciona:

Listagem 9: Exemplo abstrato do estilo de leiaute de bloco puro.

```
A XXXXXXXXXXX
B  XXXXXXX
C  XXXXXXXXXXX
D XXX
```

Nesse exemplo, a instrução A inicia a construção de controle, e a instrução D a finaliza. O alinhamento entre as duas fornece um fechamento visual evidente.

A controvérsia sobre a formatação de estruturas de controle surge em parte pelo fato de que algumas linguagens não exigem estruturas de bloco. Você pode ter uma

instrução *if-then* seguida de uma única instrução e não ter um bloco formal. Você tem de adicionar um par *begin-end* ou chaves de abertura e fechamento para criar um bloco, em vez de obtê-lo automaticamente, com cada construção de controle. O desacoplamento de *begin* e *end* da estrutura de controle - como ocorre com { e } nas linguagens C++ e Java - leva a dúvidas sobre onde colocar as instruções *begin* e *end*. Consequentemente, muitos problemas de indentação só se tornam problemas porque você tem de compensar as estruturas mal projetadas de algumas linguagens. Várias maneiras de compensar estão descritas nas seções a seguir.

Simulação de blocos puros

No caso das linguagens que não têm blocos puros, uma boa estratégia é encarar as palavras-chave begin e end (ou os sinais { e }) como extensões da construção de controle com a qual são usadas. Então, é sensato tentar simular a formatação do Visual Basic em sua linguagem. A Listagem 10 oferece uma representação abstrata da estrutura visual que você está tentando simular:

Listagem 10: Exemplo abstrato do estilo de leiaute de bloco puro.

```
A XXXXXXXXXXX
B  XXXXXXX
C  XXXXXXXXXXX
D XXX
```

Nesse estilo, a estrutura de controle abre o bloco na instrução A e o termina na instrução D. Isso significa que a palavra-chave begin deve estar no final da instrução A e que a palavra-chave end deve ser a instrução D. Na representação abstrata, para simular blocos puros, você teria que fazer algo como a Listagem 11:

Listagem 11: Exemplo abstrato da simulação do estilo de bloco puro.

```
A XXXXXXXXXXX {
B  XXXXXXX
C  XXXXXXXXXXX
D }
```

Alguns exemplos de como fica o estilo em C++ aparecem nas listagens 12, 13 e 14:

Listagem 12: Exemplo em C++ de um bloco *if* puro.

```
if ( pixelColor == Color_Red ) {
   instrução1;
   instrução2;
   ...
}
```

Listagem 13: Exemplo em C++ de um bloco *while* puro.

```
while ( pixelColor == Color_Red) {
   instrução1;
   instrução2;
   ...
}
```

Listagem 14: Exemplo em C++ de um bloco *case* puro.

```
switch ( pixelColor ) {
   case Color_Red:
```

```
      instrução1;
      instrução2;
      ...
   break;
   case Color_Green:
      instrução3;
      instrução4
      ...
   break;
   default:
      instrução5
      instrução6
      ...
   break;
}
```

Esse estilo de alinhamento funciona muito bem. Sua aparência é boa, você pode aplicá-lo consistentemente e é fácil de manter. Ele sustenta o Teorema Fundamental da Formatação, no sentido de que ajuda a mostrar a estrutura lógica do código. Trata-se de uma opção de estilo razoável. Esse estilo é padrão em Java e comum em C++.

Uso de pares begin-end (chaves) para designar os limites do bloco

Uma alternativa para uma estrutura de bloco puro é considerar que os pares begin-end (chaves) são limites de bloco. (A discussão a seguir usa begin-end para se referir genericamente aos pares begin-end, às chaves e a outras estruturas de linguagem equivalentes.) Se adotar essa estratégia, você considerará que as palavras-chave begin e end são instruções que seguem a construção de controle, em vez de trechos que fazem parte dela. Graficamente, isso é o ideal, assim como acontecia com a simulação de bloco puro, mostrada novamente na Listagem 15:

Listagem 15: Exemplo abstrato da simulação do estilo de bloco puro.

```
A XXXXXXXXXXX
B  XXXXXXX
C  XXXXXXXXXXX
D XXX
```

Mas, nesse estilo, para tratar as palavras-chave begin e end como partes da estrutura de bloco, em vez de partes da instrução de controle, você tem de colocar a palavra- chave begin no início do bloco (em vez de colocar no fim da instrução de controle) e a palavra-chave end no fim do bloco (em vez de terminar a instrução de controle). Na representação abstrata, você deverá fazer algo como o que foi feito na Listagem 16:

Listagem 16: Exemplo abstrato do uso de *begin* e *end* como limites do bloco.

```
A XXXXXXXXXXX
    {
B  XXXXXXX
C  XXXXXXXXXXX
D  }
```

Alguns exemplos de como usar begin e end como limites de bloco em C++ você pode observar nas listagens 17, 18 e 19:

Listagem 17: Exemplo em C++ do uso de begin e end como limites de bloco em um *if*.

```
if ( pixelColor == Color_Red )
    {
    instrução1;
    instrução2;
    ...
    }
```

Listagem 18: Exemplo em C++ do uso de begin e end como limites de bloco em um *while*.

```
while ( pixelColor == Color_Red)
    {
    instrução1;
    instrução2;
    ...
    }
```

Listagem 19: Exemplo em C++ do uso de begin e end como limites de bloco em *switch/case* .

```
switch ( pixelColor )
    {
    case Color_Red:
      instrução1;
      instrução2;
      ...
      break;
    case Color_Green:
      instrução3;
      instrução4
      ...
      break;
    default:
      instrução5
      instrução6
      ...
      break;
    }
```

Esse estilo de alinhamento funciona bem; ele sustenta o Teorema Fundamental da Formatação (mais uma vez, expondo a estrutura lógica subjacente do código). Sua única limitação é que ele não pode ser aplicado literalmente em instruções switch/case nas linguagens C++ e Java, como mostrado pela Listagem 19. (A palavra-chave break é uma substituta da chave de fechamento, mas não há nenhuma equivalente para a chave de abertura.)

Leiaute de final de linha

Outra estratégia de leiaute é o "leiaute de final de linha", que se refere a um grande grupo de estratégias de leiaute no qual o código é indentado no meio ou no final da linha. A indentação de final de linha é usada para alinhar um bloco com a palavra-chave que o iniciou, para fazer os parâmetros subsequentes de uma rotina ficarem alinhados sob seu primeiro parâmetro, para alinhar casos em uma instrução case, e para outros propósitos semelhantes. A Listagem 20 é um exemplo abstrato.

Listagem 20: Exemplo abstrato do estilo de leiaute de final de linha.

```
A XXXX  XXXXXXX
B       XXXXXXX
C       XXXXXXXXXXX
D       }
```

Nesse exemplo, a instrução A inicia a construção de controle, e a instrução D a finaliza. As instruções B, C e D estão alinhadas sob a palavra-chave que iniciou o bloco na instrução A.

A indentação uniforme de B, C e D mostra que elas estão agrupadas. A listagem 21 é um exemplo menos abstrato de código formatado usando essa estratégia:

Listagem 21: Exemplo em Visual Basic de leiaute de fim de linha de um bloco *while*.

```
While   pixelColor = Color_Red
        instrução1
        instrução2
        ...
        Wend
```

No exemplo, a representação de begin foi colocada no final da linha, em vez de ser colocada sob a palavra-chave correspondente. Algumas pessoas preferem colocar begin sob a palavra-chave, mas a escolha entre esses dois pontos delicados é o menor dos problemas desse estilo.

O estilo de leiaute de final de linha funciona de forma aceitável em alguns casos. A Listagem 22 é um exemplo em que ele funciona.

Listagem 22: Exemplo raro, em Visual Basic, em que o leiaute de fim de linha parece bom.

```
If ( soldCount > 1000 )  Then
                           markdown = 0.10
                           profit = 0.05
                         Else
                           markdown = 0.05
                         End If
```

Neste caso, as palavras-chave Then, Else e End If estão alinhadas e o código após elas também está. O efeito visual é uma estrutura lógica clara.

Se você olhar de forma crítica o exemplo de instrução case anterior, poderá prever o desmoronamento desse estilo. Quando a expressão condicional se tornar mais complicada, o estilo dará pistas inúteis ou erradas sobre a estrutura lógica. A Listagem 23 é um exemplo de como o estilo falha quando é usado com uma condicional mais complicada:

Listagem 23: Exemplo mais típico, em Visual Basic, em que o leiaute de fim de linha falha.

```
If ( soldCount > 10 And prevMonthSales > 10 )  Then
   If ( soldCount > 100 And prevMonthSales > 10 )  Then
     If ( soldCount > 1000 ) Then
                               markdown = 0.10
                               profit = 0.05
                             Else
                               markdown = 0.05
                             End If
                                                  Else
                                                    markdown = 0.025
                                                  End If
                                              Else
                                                markdown = 0.0
                                              End If
```

Qual é o motivo da formatação bizarra das cláusulas Else no final do exemplo?

Elas estão indentadas consistentemente sob as palavras-chave correspondentes, mas é difícil dizer que suas indentações tornam mais clara a estrutura lógica. E se o código fosse modificado de modo que o comprimento da primeira linha mudasse, o estilo de final de linha exigiria que a indentação das instruções correspondentes fosse alterada. Isso acarreta um problema de manutenção que o bloco puro, a simulação de bloco puro e o uso de begin-end para designar os limites de bloco não acarretam.

Você pode estar pensando que esses exemplos foram inventados apenas para atingir um objetivo, mas esse estilo tem sido persistente, a despeito de seus inconvenientes. Numerosos livros-texto e referências de programação o têm recomendado.

De um modo geral, o leiaute de final de linha é impreciso, difícil de aplicar consistentemente e difícil de manter. Você verá outros problemas do leiaute de final de linha ao longo de todo este capítulo.

Qual estilo é melhor?

Se você estiver trabalhando com Visual Basic, use indentação de bloco puro. (De qualquer forma, o IDE do Visual Basic torna difícil não usar esse estilo.)

Em Java, a prática-padrão é usar indentação de bloco puro.

Em C++, você poderia simplesmente escolher o estilo de que gosta ou o que é preferido pela maioria das pessoas de sua equipe. Tanto a simulação de bloco puro como os limites de bloco begin-end ({ e }) funcionam bem. O único estudo que comparou os dois estilos não encontrou nenhuma diferença estatisticamente significativa entre eles, no que diz respeito à inteligibilidade (Hansen e Yim 1987).

Nenhum dos dois estilos é infalível e cada um deles exige um compromisso "razoável e óbvio" ocasional. Você pode preferir um ou outro por motivos estéticos. Este livro usa o estilo de bloco puro nos exemplos de código,- portanto, examinando tais exemplos, você poderá saber mais sobre como esse estilo funciona. Uma vez escolhido um estilo, você usufrui da maior vantagem do bom leiaute aplicando-o consistentemente.

12.4 Organizando estruturas de controle

Referência o leiaute de alguns elementos do programa é principalmente uma questão de estética. No entanto, o leiaute das estruturas de controle afeta a legibilidade e a capacidade de compreensão; portanto, tem prioridade prática.

Pontos delicados da formatação de blocos de estrutura de controle

Trabalhar com blocos de estrutura de controle exige atenção a alguns detalhes especiais. Aqui estão algumas diretrizes:

Evite pares de palavras-chave begin-end não indentados No estilo mostrado na Listagem 24, o par begin-end está alinhado com a estrutura de controle, e as instruções que begin e end englobam estão indentadas sob begin.

Listagem 24: Exemplo em Java de pares begin-end não indentados.

```
for ( int i = 0; i < MAX_LINES; i++ )
{
   ReadLine( i );
   ProcessLine( i );
}
```

Embora essa estratégia pareça boa, ela viola o Teorema Fundamental da Formatação, pois não mostra a estrutura lógica do código. Usadas dessa maneira, as instruções

begin e end não fazem parte da construção de controle, mas também não fazem parte da(s) instrução(ões) que aparece(m) depois. A Listagem 25 é uma visão abstrata dessa estratégia:

Listagem 25: Exemplo abstrato de indentação errada.

```
A   XXXXXXXXXXXXX
B   XXX
C      XXXXXXXXX
D      XXXXXXXXXXXXXXX
E   XXX
```

Nesse exemplo, a instrução B está subordinada à instrução A? Ela não parece fazer parte da instrução A e também não parece que está subordinada a ela. Se você tiver usado essa estratégia, mude para um dos dois estilos de leiaute descritos anteriormente e sua formatação será mais consistente.

Evite indentação dupla com begin e end Um corolário da regra contra os pares begin-end não-indentados é a regra contra os pares begin-end duplamente indentados. Nesse estilo, mostrado na Listagem 26, begin e end são indentados e as instruções que eles englobam são novamente indentadas:

Listagem 26: Exemplo em Java de indentação dupla inadequada de um bloco begin-end.

```
for ( int i = 0; i < MAX_LINES; i++ )
    {
      ReadLine( i );
      ProcessLine( i );
    }
```

Esse é outro exemplo de estilo que parece correto, mas viola o Teorema Fundamental da Formatação. Um estudo realizado não mostrou nenhuma diferença na compreensão entre programas que têm indentação simples e programas que têm indentação dupla (Miaria et al. 1983), mas este estilo não mostra precisamente a estrutura lógica do programa. ReadLine( ) e ProcessLine ( ) são mostrados como se estivessem logicamente subordinados ao par begin-end, e não estão.

A estratégia também aumenta a complexidade da estrutura lógica de um programa. Qual entre as estruturas mostradas nas Listagens 27 e 28 parece a mais complicada?

Listagem 27: Estrutura abstrata 1.

```
    XXXXXXXXXXXXX
      XXX
        XXXXXXXXX
        XXXXXXXXXXXXXXX
      XXX
```

Listagem 28: Estrutura abstrata 2.

```
    XXXXXXXXXXXXX
      XXX
      XXXXXXXXX
      XXXXXXXXXXXXXXX
      XXX
```

Ambas são representações abstratas da estrutura do loop “for”. A estrutura abstrata 1 parece mais complicada, apesar de representar o mesmo código da estrutura

abstrata 2. Se você fosse aninhar instruções em dois ou três níveis, a indentação dupla forneceria quatro ou seis níveis de indentação. O leiaute resultante pareceria mais complicado do que o código realmente é. Evite o problema, usando simulação de bloco puro ou usando begin e end como limites de bloco e alinhando begin e end com as instruções que elas englobam.

Outras considerações

Embora a indentação de blocos seja o maior obstáculo na formatação de estruturas de controle, você encontrará alguns outros tipos de problemas; portanto, aqui estão mais algumas diretrizes:

Use linhas em branco entre parágrafos Alguns blocos de código não são demarcados com pares begin-end. Um bloco lógico - um grupo de instruções relacionadas - deve ser tratado da mesma maneira que os parágrafos em português. Separe-as umas das outras usando linhas em branco. A Listagem 29 mostra um exemplo de parágrafos que devem ser separados:

Listagem 29: Exemplo em C++ de código que deve ser agrupado e separado.

```
cursor.star   = startingScanLine;
cursor.end    = endingScanLine;
window.title  = editWindow.title;
window.dimension         = editWindow.dimensions;
window.foregroundColor   = userPreferences.foregroundColor;
cursor.blinkRate         = editMode.blinkRate;
window.backgroundColor   = userPreferences.backgroundColor;
SaveCursor( cursor );
SetCursor( cursor );
```

Esse código parece estar correto, mas linhas em branco o melhorariam de duas maneiras. Primeiramente, quando você tem um grupo de instruções que não precisam ser executadas em nenhuma ordem específica, é tentador juntá-las dessa maneira. Para o computador, você não precisa refinar mais a ordem das instruções, mas os leitores humanos gostam de mais indícios sobre quais instruções precisam ser executadas em uma ordem específica e quais ficam juntas apenas por conveniência. A disciplina de colocar linhas em branco por todo um programa faz você pensar quais instruções realmente estão relacionadas. O trecho revisado que aparece na Listagem 30 mostra como esse conjunto deve ser organizado.

Listagem 30: Exemplo em C++ de código que está adequadamente agrupado e separado.

```
window.title             = editWindow.title;
window.dimension         = editWindow.dimensions;
window.foregroundColor   = userPreferences.foregroundColor;
window.backgroundColor   = userPreferences.backgroundColor;

cursor.star              = startingScanLine;
cursor.end               = endingScanLine;
cursor.blinkRate         = editMode.blinkRate;
SaveCursor( cursor );
SetCursor( cursor );
```

O código reorganizado mostra que duas coisas estão acontecendo. No primeiro exemplo, a falta de organização das instruções e de linhas em branco, e o antigo truque dos sinais de igualdade alinhados, fazem as instruções parecem mais relacionadas do que realmente são.

A segunda maneira pela qual o uso de linhas em branco tende a melhorar o código é que elas abrem espaços naturais para comentários. Na Listagem 30, um comentário acima de cada bloco complementaria muito bem o leiaute aprimorado.

Formate consistentemente os blocos de uma única instrução Um bloco de uma única instrução é aquele que contém apenas uma instrução após a estrutura de controle, como no caso de um teste if. Nesse caso, begin e end não são necessários para uma compilação correta e você tem as três opções de estilo mostradas na Listagem 31:

Listagem 31: Exemplo em Java de opções de estilo para blocos de uma única instrução.

```
//estilo 1
if ( expressão )
   uma instrução;

//estilo 2a
if ( expressão ) {
   uma instrução;
}

//estilo 2b
if ( expressão )
   {
   uma instrução;
   }

//estilo 3
if ( expressão ) uma instrução;
```

Existem argumentos a favor de cada uma dessas estratégias. O estilo 1 segue o esquema de indentação usado com blocos; portanto, ele é compatível com outras estratégias. O estilo 2 (tanto 2a como 2b) também é compatível, e o par begin-end reduz a chance de que você venha a acrescentar instruções após o teste if e se esqueça de adicionar begin e end. Isso seria um erro particularmente sutil, pois a indentação diria que tudo está correto, mas não seria interpretada da mesma maneira pelo compilador. A principal vantagem do estilo 3 em relação ao estilo 2 é que ele é mais fácil de digitar. A vantagem do estilo 3 sobre o estilo 1 é que, se for copiado em outro lugar do programa, cresce a possibilidade de que seja copiado corretamente. Sua desvantagem é que, em um depurador orientado a linhas, esse depurador trata a linha como uma só e não mostra a você se executa a instrução após o teste if.

Para expressões complicadas, coloque condições separadas em linhas separadas Coloque cada parte de uma expressão complicada em sua própria linha. A Listagem 32 mostra uma expressão formatada sem qualquer atenção à legibilidade:

Listagem 32 Exemplo em Java de expressão complicada basicamente não-formatada (e ilegível).

```
if ((('0' <= inChar) && (inChar <= '9')) || (('a' <= inChar) && (inchar
<= 'z')) || (('A' <= inChar) && (inchar <= 'Z')))
```

Esse é um exemplo de formatação para o computador e não para leitores humanos. Dividindo a expressão em várias linhas, como na Listagem 33, você pode melhorar a legibilidade.

Listagem 33 Exemplo em Java de expressão complicada legível.

```
if ( ( ( '0' <= inChar ) && ( inChar <= '9' ) ) ||
   ( ( 'a' <= inChar ) && ( inchar <= 'z' ) ) ||
   ( ( 'A' <= inChar ) && ( inchar <= 'Z' ) ) )
```

O segundo trecho usa várias técnicas de formatação - indentação, espaçamento, ordenação pelo número da linha e tornar óbvia cada linha incompleta - e o resultado é uma expressão legível. Além disso, a intenção do teste é clara. Se a expressão contivesse um erro menor, como o uso de z em vez de Z, ele ficaria evidente no código formatado dessa maneira, enquanto que o erro não seria claro com uma formatação menos cuidadosa.

Evite as instruções goto O motivo original para se evitar instruções goto era que elas tomavam difícil provar que um programa estava correto. Esse é um argumento excelente para todos que queiram provar que seus programas estão corretos, o que significa ninguém,praticamente. O problema que mais pesa para a maioria dos programadores é que as instruções goto tornam o código difícil de formatar. Você indenta todo o código entre a instrução goto e o rótulo para onde ele vai? E se você tiver várias instruções goto para o mesmo rótulo? Você indenta cada nova instrução sob a anterior? Aqui estão algumas orientações para a formatação de instruções goto:

- Evite as instruções goto. Isso acaba completamente com o problema da formatação.
- Use um nome com todas as letras maiúsculas para o rótulo para onde o código vai. Isso torna o rótulo óbvio.
- Coloque sozinha em uma linha a instrução que contém a palavra-chave goto. Isso torna a instrução goto óbvia.
- Coloque sozinho em uma linha o rótulo para onde a instrução goto vai. Circunde-a com linhas em branco. Isso torna o rótulo óbvio. Retire a indentação da linha que contém o rótulo na margem esquerda, para torná-lo o mais evidente possível.

A Listagem 34 mostra essas convenções de leiaute da instrução goto em ação.

Listagem 34 Exemplo em C++ de tirar o melhor proveito de uma situação ruim usando goto.

```
void PurgeFiles( ErrorCode & errorCode ) {
   FileList fileList;
   int numFilesToPurge = 0;
   MakePurgeFileList( fileList, numFilesToPurge );

   errorCode = FileError_Success;
   int fileIndex = 0;
   while ( fileIndex < numFilesToPUrge ) {
     DataFile fileToPurge;
     if ( !FindFile( fileList[ fileInde ], fileToPurge ) ) {
       errorCode = FileError_NotFound;
       goto END_PROC;
     }

     if ( !OpenFile( fileToPurge ) ) {
       errorCode = FileError_NotOpen;
       goto END_PROC;
     }

     if ( !OverwriteFile( fileToPurge ) ) {
```

```
        errorCode = FileError_CantOverwrite;
        goto END_PROC;
      }

      if ( !Erase( fileToPurge ) ) {
        errorCode = FileError_CantErase;
        goto END_PROC;
      }
      fileIndex++;
    }

    END_PROC:
      DeletePurgeFileList( fileList, numFilesToPurge );
}
```

o exemplo mostrado é relativamente longo, para que você possa ver uma situação em que um programador especialista poderia decidir conscientemente que uma instrução goto é a melhor opção de projeto. Nesse caso, a formatação mostrada é quase tudo que você pode fazer.

Nenhuma exceção de final de linha para instruções case Um dos perigos do leiaute de final de linha surge na formatação de instruções case. Um estilo popular de formatação de instruções case é a indentação à direita da descrição de cada caso, como mostrado na Listagem 35. O maior problema desse estilo é a dor de cabeça no momento da manutenção.

Listagem 35 Exemplo em C++ de leiaute de final de linha difícil para manter.

```
switch ( ballColor ) {
  case BallColor_Blue:              Rollout();
                                    break;
  case BallColor_Orange:            SpinOnFinger();
                                    break;
  case BallColor_FluorescentGreen:  Spike();
                                    break;
  case BallColor_White:             KnockCoverOff();
                                    break;
  case BallColor_WhiteAndBlue:      if ( mainColor == BallColor_White ) {
                                      KnockCoverOff();
                                    }
                                    else if (mainColor == BallColor_Blue){
                                      RollOut();
                                    }
                                    break;
  default:                          FatalError( "Unrecognized type" );
                                    break;
}
```

Se você adicionar um caso com um nome mais longo do que qualquer um dos nomes existentes, terá que deslocar todos os casos e o código que os acompanha. A grande indentação inicial torna complicado acomodar mais lógica, como se vê no caso WhiteAndBlue. A solução é trocar para o incremento de indentação-padrão. Se você indentar as instruções de um loop por três espaços, indente os casos de uma instrução case pelo mesmo número de espaços, como na Listagem 36:

Listagem 36 Exemplo em C++ de boa indentação-padrão de uma instrução case.

```
switch ( ballColor ) {
  case BallColor_Blue:
    Rollout();
    break;
  case BallColor_Orange:
    SpinOnFinger();
    break;
  case BallColor_FluorescentGreen:
    Spike();
    break;
  case BallColor_White:
    KnockCoverOff();
    break;
  case BallColor_WhiteAndBlue:
    if ( mainColor == BallColor_White ) {
      KnockCoverOff();
    }
    else if (mainColor == BallColor_Blue){
      RollOut();
    }
    break;
  default:
    FatalError( "Unrecognized type" );
    break;
}
```

Esse é um caso em que muitas pessoas poderão preferir a aparência do primeiro exemplo. Entretanto, para a capacidade de acomodar linhas mais longas, consistência e manutenibilidade, a segunda estratégia vence facilmente.

Se você tivesse uma instrução case na qual todos os casos fossem exatamente equiparados e todas as ações fossem curtas, poderia considerar a colocação do caso e da ação na mesma linha. Na maioria das situações, entretanto, você lamentará por isso. A formatação é inicialmente complicada, se desfaz no caso de modificação, e é difícil manter a estrutura de todos os casos em paralelo, quando algumas das ações curtas se tomarem mais longas.

12.5 Organizando instruções individuais

Esta seção explica muitas maneiras de melhorar a organização das instruções individuais em um programa.

Comprimento da instrução

Uma regra comum, é limitar o comprimento da linha da instrução a 80 caracteres. Aqui estão os motivos:

- Linhas mais longas do que 80 caracteres são difíceis de ler.
- A limitação a 80 caracteres desencoraja o aninhamento profundo.
- Linhas mais longas do que 80 caracteres frequentemente não se encaixarão em papel de 8,5" x 11", especialmente quando o código é impresso no modo "2 up" (duas páginas de código por página física impressa).

Com telas maiores, tipos mais estreitos e modo horizontal, o limite de 80 caracteres parece cada vez mais arbitrário. Uma única linha de 90 caracteres normalmente é mais legível do que outra dividida em duas, apenas para evitar o

transbordamento na 80ª coluna. Com uma tecnologia moderna, provavelmente não haverá problemas em ultrapassar as 80 colunas ocasionalmente.

Usando espaços para proporcionar maior clareza

Adicione espaço em branco dentro de uma instrução para favorecer a legibilidade:

Use espaços para tomar expressões lógicas legíveis A expressão

```
while(pathName[startPath+position]<>';') and
   ((startPath+position)<length(pathName)) do
```

é quase tão legível quanto desafiovocêaleristo.

Como regra, você deve separar os identificadores usando espaços. Se você aplicar essa regra, a expressão while ficará como segue:

```
while ( pathName[ startPath+position ] <> ';' ) and
   (( startPath+position ) < length( pathName ) ) do
```

Alguns artistas do software poderão recomendar uma melhoria nessa expressão em particular, com espaços adicionais para dar ênfase à sua estrutura lógica, como segue:

```
while ( pathName[ startPath + position ] <> ';' ) and
   (( startPath + position ) < length( pathName ) ) do
```

Isso está certo, embora o primeiro uso de espaços seja suficiente para garantir a legibilidade. Entretanto, espaços extras dificilmente prejudicam; por isso, seja generoso com eles.

Use espaços para tomar legíveis as referências de array A expressão

```
grossRate[census[groupId].gender, census[groupId].ageGroup]
```

não é mais legível do que a expressão while densa vista anteriormente. Use espaços em torno de cada índice no array, para tomar os índices legíveis. Se você aplicar essa regra, a expressão ficará como segue:

```
grossRate[ census[ groupId ].gender, census[ groupId ].ageGroup ]
```

Use espaços para tomar legíveis os argumentos de rotina Qual é o quarto argumento da rotina a seguir?

```
ReadEmployeeData(maxEmps,empData,inputFile,empCount,inputError);
```

E agora, qual é o quarto argumento da rotina a seguir?

```
GetCensus( inputFile, empCount, empData, maxEmps, inputError );
```

Qual deles foi mais fácil de encontrar? Essa é uma questão interessante e realista, pois as posições dos argumentos são significativas em todas as linguagens procedurais importantes. É comum ter uma especificação de rotina na metade de sua tela e a chamada da rotina na outra metade, e comparar cada parâmetro formal com cada parâmetro real.

Formatação de linhas de continuação

Um dos problemas mais irritantes do leiaute de programa é decidir o que fazer com a parte de uma instrução que transborda para a próxima linha. Você a indenta com

a quantidade de indentação normal? Você a alinha sob a palavra-chave? E quanto às atribuições?

Eis, a seguir, uma estratégia sensata e consistente, particularmente útil em Java, C, C++, Visual Basic e em outras linguagens que estimulam o uso de nomes de variável longos:

Torne evidente o fato de uma instrução estar incompl. Às vezes, uma instrução precisa ser dividida em duas linhas, ou porque ela é mais longa do que os padrões de programação permitem ou porque é absurdamente longa para ser colocada em uma única linha. Tome evidente que o trecho da instrução na primeira linha é apenas parte de uma instrução. A maneira mais fácil de fazer isso é dividir a instrução, de modo que a parte que está na primeira linha seja evidentemente incorreta sintaticamente, caso fique sozinha. Alguns exemplos aparecem na Listagem 37:

Listagem 37 Exemplos em Java de instruções obviamente incompletas.

```
//os caracteres && sinalizam que a instrução não está completa
while ( pathName[ startPath + position ] != ';' ) &&
   ( ( startPath + position ) <= pathName.length() )
...

//o sinal de adição (+) indica que a instrução não está completa
totalBill = totalBill + customerPurchases[ customerID ] +
   SalesTax( customerPurchases[ customerID ] );
...

//a vírgula (,) sinaliza que a instrução não está completa
DrawLine( window.north, window.south, window.east, window.west,
   currentWidth, currentAttribute );
...
```

Além de informar ao leitor que a instrução não está completa na primeira linha, a quebra ajuda a evitar modificações incorretas. Se a continuação da instrução fosse excluída, não pareceria que você tinha simplesmente se esquecido de um parênteses ou de um ponto e vírgula na primeira linha - claramente, ela precisaria de algo mais.

Uma estratégia alternativa que também funciona adequadamente é colocar o caractere de continuação no início da linha de continuação, como mostrado na Listagem 38.

Listagem 38 Exemplos em Java de instruções obviamente incompletas – estilo alternativo.

```
while ( pathName[ startPath + position ] != ';' )
   && ( ( startPath + position ) <= pathName.length() )
...

totalBill = totalBill + customerPurchases[ customerID ]
   + SalesTax( customerPurchases[ customerID ] );
...
```

Embora esse estilo não induza a um erro de sintaxe com um sinal && ou + pendente, ele torna mais fácil procurar operadores na margem esquerda da coluna, onde o texto está alinhado, do que na margem direita, onde é irregular. Ele tem a vantagem adicional de tornar mais clara a estrutura das operações, como ilustrado na Listagem 39.

Listagem 39 Exemplo em Java de um estilo que esclarece operações complexas.

```
totalBill = totalBill
```

```
  + customerPurchases[ customerID ]
  + CitySalesTax( customerPurchases[ customerID ] )
  + StateSalesTax( customerPurchases[ customerID ] )
  + FootballStadiumTax()
  - SalesTaxExemption( customerPurchases[ customerID );
```

Mantenha juntos os elementos intimamente relacionados Quando você quebrar uma linha, mantenha juntos os elementos que estão relacionados: referências de array, argumentos de uma rotina, etc. O exemplo mostrado na Listagem 40 é uma forma incorreta:

Listagem 40 Exemplo em Java de quebra de linha malfeita.

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) + LateCharge(
  paymentHistory[ customerID ] );
```

Com certeza, essa quebra de linha segue a diretriz de evidenciar o fato de que a instrução está incompleta, mas ela faz isso de uma maneira que torna a instrução desnecessariamente difícil de ler. Você poderia encontrar um caso em que a quebra é necessária, mas neste caso ela não é. É melhor manter todas as referências de array em uma única linha. A Listagem 41 mostra uma formatação melhor:

Listagem 41 Exemplo em Java de quebra de linha bem feita.

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) +
  LateCharge( paymentHistory[ customerID ] );
```

Indente as linhas de continuação de chamada de rotina com a quantidade-padrão Se você normalmente indenta em três espaços as instruções de um loop ou de uma condicional, indente as linhas de continuação de uma rotina em três espaços. Alguns exemplos aparecem na Listagem 42:

Listagem 42: Exemplos em Java de indentação de linhas de continuação de chamada de rotina usando o incremento de indentação indentação-padrão.

```
DrawLine( window.north, window.south, window.east, window.west,
   currentWidth, current Attribute );
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],
   italic[ fontId ], syntheticAttribute[ fontId ].underline,
   syntheticAttribute [ fontId ].strikeout );
```

Uma alternativa a essa estratégia é alinhar as linhas de continuação sob o primeiro argumento da rotina, como mostrado na Listagem 43:

Listagem 43: Exemplos em Java de indentação de uma linha de continuação de chamada de rotina para dar ênfase aos nomes de rotina.

```
DrawLine( window.north, window.south, window.east, window.west,
          currentWidth, currentAttribute ) ;
SetFontAttributes ( faceName[ fontId ], size[ fontId ], bold[ fontId ],
                    italic[fontId], syntheticAttribute[fontId].underline
                    , syntheticAttribute[ fontId ].strikeout );
```

Do ponto de vista estético, isso parece um pouco irregular, comparado com a primeira estratégia. Também é difícil de manter, quando mudam os nomes de rotina, os nomes de argumento, etc. Com o passar do tempo, a maioria dos programadores tende a usar o primeiro estilo.

Facilite a maneira de encontrar o fim de uma linha de continuação Um problema da estratégia mostrada anteriormente é que você não consegue encontrar facilmente o final de cada linha. Outra alternativa é colocar cada argumento em sua própria linha e indicar o fim do grupo com um parênteses de fechamento. A Listagem 44 mostra como isso funciona.

Listagem 44 Exemplos em Java de formatação de linhas de continuação de chamada de rotina com um argumento em cada linha.

```
DrawLine(
  window.north,
  window.south,
  window.east,
  window.west,
  currentWidth,
  currentAttribute
);

SetFontAttributes (
  faceName[ fontId ],
  size[ fontId ],
  bold[ fontId ],
  italic[ fontId ],
  syntheticAttribute[ fontId ].underline,
  syntheticAttribute[ fontId ].strikeout
);
```

Obviamente, essa estratégia ocupa muito espaço. Entretanto, se os argumentos de uma rotina são longas referencias para campo de objeto ou nomes de ponteiro, como acontece com as duas últimas, usar um argumento por linha melhora substancialmente a legibilidade. Os caracteres); no final do bloco tornam claro o final da chamada. Você também não precisa reformatar quando adicionar um parâmetro; basta adicionar uma nova linha.

Na prática, normalmente apenas algumas rotinas precisam ser divididas em várias linhas. Você pode deixar as outras em uma única linha. Qualquer uma das três opções de formatação de chamadas de rotina em várias linhas funciona bem, basta você usá-las consistentemente.

Indente as linhas de continuação de instrução de controle com a quantidade-padrão Se você ficar sem espaço para um *loop for,* para um *loop while* ou para uma instrução *if*, indente a linha de continuação com a mesma quantidade de espaço com a qual indenta as instruções de um loop ou após uma instrução if. Dois exemplos são mostrados na Listagem 45:

Listagem 45 Exemplos em Java de indentação de linhas de continuação de instrução de controle.

```
while ( pathName[ startPath + position ] != ';' ) &&
     ( ( startPath + position ) <= pathName.length() ) {
     ...
}

for ( int employeeNum = employee.primeiro + employee.offset;
     employeeNum < employee.primeiro + employee.offset + employee.total;
     employeeNum++) {
     ...
}
```

Isso satisfaz os critérios definidos anteriormente neste capítulo. A parte da continuação da instrução é feita logicamente - ela sempre fica indentada sob a instrução a que dá continuidade. A indentação pode ser feita adequadamente - ela usa apenas alguns espaços a mais do que a linha original. Ela é legível e fácil de manter como todo o resto. Em alguns casos, você poderá melhorar a legibilidade otimizando a indentação ou o espaçamento, mas tenha sempre em mente o compromisso com a manutenibilidade.

Não alinhe o lado direito das instruções de atribuição Na primeira edição deste livro, eu recomendei alinhar o lado direito das instruções contendo atribuições, como mostrado na Listagem 46:

Listagem 46: Exemplo em Java de leiaute de final de linha usado para continuação de instrução de atribuição - prática não-recomendável.

```
customerPurchases = customerPurchases + CustomerSales( CustomerID );
customerBill      = customerBill + customerPurchases;
totalCustomerBill = customerBill + PreviousBalance( customerID ) +
                    LateCharge( customerID );
customerRating    = Rating( customerID, totalCustomerBill );
```

Com a vantagem da percepção de 10 anos depois, descobri que, embora esse estilo de indentação possa parecer atraente, ele torna problemático manter o alinhamento de sinais de igualdade quando os nomes de variável mudam e o código é executado por meio de ferramentas que substituem tabulações por espaços e espaços por tabulações. Ele também é difícil de manter quando as linhas são movidas entre diferentes partes do programa que tenham diferentes níveis de indentação.

Por coerência em relação às outras diretrizes de indentação, assim como em relação à manutenibilidade, trate os grupos de instruções contendo operações de atribuição exatamente como você trataria outras as instruções, como mostra a Listagem 47:

Listagem 47: Exemplo em Java de indentação indentação-padrão para continuação de instrução de atribuição - prática recomendável.

```
customerPurchases = customerPurchases + CustomerSales ( CustomerID );
customerBill = customerBill + customerPurchases;
totalCustomerBill = customerBill + PreviousBalance ( customerlD ) +
    LateCharge ( customerlD );
customerRating = Rating( customerID, totalCustomerBill );
```

Indente as linhas de continuação de instrução de atribuição com a quantidade- padrão Na Listagem 45, a linha de continuação da terceira instrução de atribuição está indentada com a quantidade-padrão. Isso foi feito pelos mesmos motivos pelos quais as instruções de atribuição em geral não são formatadas de nenhuma maneira especial: legibilidade geral e manutenibilidade.

Usando apenas uma instrução por linha

As linguagens modernas, como C++ e Java, permitem ter várias instruções por linha. Entretanto, o poder da formatação livre é um misto de bênção e maldição, quando se trata de colocar várias instruções em uma linha. A linha a seguir contém várias instruções que poderiam estar logicamente separadas em suas próprias linhas:

```
i = 0; j = 0; k = 0; DestroyBadLoopNames( i, j, k );
```

Um argumento a favor da colocação de várias instruções em uma única linha é que isso exige menos linhas de espaço na tela ou no papel da impressora, o que permite que mais código seja visto simultaneamente. Essa também é uma maneira de agrupar instruções relacionadas; alguns programadores acreditam que isso forneça indícios de otimização para o compilador.

Esses são bons motivos, mas as razões para limitar-se a uma instrução por linha são mais fortes:

- Colocar cada instrução em sua própria linha proporciona uma visão precisa da complexidade de um programa. Esse procedimento não oculta a complexidade, fazendo com que instruções complexas pareçam simples. As instruções complexas parecem complexas. As instruções fáceis parecem fáceis.
- Colocar várias instruções em uma única linha não fornece indícios de otimização para os compiladores modernos. A otimização dos compiladores atuais não depende de indícios de formatação para ser realizada.
- Com as instruções em suas próprias linhas, o código é lido de cima para baixo apenas, em vez de ser lido de cima para baixo e da esquerda para a direita. Quando você está procurando uma linha de código específica, seus olhos devem acompanhar a margem esquerda do código. Eles não devem ter de percorrer cada linha, apenas porque uma linha pode conter duas ou mais instruções.
- Com as instruções em suas próprias linhas, é fácil encontrar erros de sintaxe, quando seu compilador fornece apenas os números de linha dos erros. Se você tem várias instruções em uma linha, o número da linha não indica qual instrução está errada.
- Com apenas uma instrução em uma linha, é fácil percorrer o código com depuradores orientados a linhas. Quando há várias instruções em uma linha, o depurador executa todas elas de uma só vez e você precisa trocar a visualização para assembler para percorrer as instruções individuais.
- Com uma instrução em uma linha, é fácil editar instruções individuais - excluir uma linha ou converter uma linha em comentário, temporariamente. Se você tem várias instruções em uma linha, precisa fazer sua edição entre outras instruções.

Em C++, evite o uso de várias operações por linha (efeitos colaterais)

Efeitos colaterais são uma consequência de uma instrução, mas não a consequência principal. Em C++, o operador ++ em uma linha contendo outras operações é um efeito colateral. Do mesmo modo, atribuir um valor a uma variável e usar o lado esquerdo da atribuição em uma condicional é um efeito colateral.

Os efeitos colaterais tendem a tornar o código difícil de ler. Por exemplo, se n é igual a 4, qual é a saída impressa da instrução mostrada na Listagem 48?

Listagem 48: Exemplo em C++ de um efeito colateral imprevisível.

```
PrintMessage( ++n, n + 2 ) ;
```

A saída é 4 e 6? É 5 e 7? É 5 e 6? A resposta é “Nenhuma das anteriores”. O primeiro argumento, ++n, é 5. Mas a linguagem C++ não define a ordem em que os termos de uma expressão ou os argumentos de uma rotina são avaliados. Assim, o compilador pode avaliar o segundo argumento, n + 2, antes ou depois do primeiro argumento; o resultado poderia ser 6 ou 7, dependendo do compilador. A Listagem 49 mostra como você deve reescrever a instrução de modo que o objetivo fique claro;

Listagem 49: Exemplo em C++ que evita um efeito colateral imprevisível.

```
++n;
PrintMessage ( n, n + 2 );
```

Se você ainda não estiver convencido de que deve colocar os efeitos colaterais em suas próprias linhas, tente descobrir o que a rotina mostrada na Listagem 50 faz:

Listagem 50: Exemplo em C de operações demais em uma linha.

```
strcpy( char * t, char * s ) {
     while ( *++t = *++s )
        ;
}
```

Alguns programadores experientes em C não vêem a complexidade desse exemplo, pois essa é uma função familiar. Eles olham para ela e dizem "Isso é strcpy()". Neste caso, entretanto, não é realmente strcpy(). Ela contém um erro. Se você disse "Isso é strcpy()", quando viu o código, estava na verdade reconhecendo-o e, não, lendo-o. É exatamente essa a situação em que você está quando depura um programa: o código que você ignora porque o "reconhece", em vez de lê-lo, pode conter o erro que é mais difícil de encontrar do que precisa ser.

O trecho mostrado na Listagem 51 é funcionalmente idêntico ao primeiro, e é mais legível:

Listagem 51: Exemplo em C de várias operações legíveis em cada linha.

```
strcpy( char * t, char * s ) {
   do {
       ++t;
       ++s;
       *t = *s;
   }
       while ( *t != '\0' );
}
```

No código reformatado, o erro fica aparente. Claramente, t e s são incrementados antes que *s seja copiado em *t. O primeiro caractere é perdido.

O segundo exemplo parece mais elaborado do que o primeiro, mesmo sendo idênticas as operações efetuadas no segundo. O motivo de parecer mais elaborado é que ele não oculta a complexidade das operações.

Um melhor desempenho também não justifica colocar várias operações na mesma linha. Como as duas rotinas strcpy() são logicamente equivalentes, você esperaria que o compilador gerasse código idêntico para elas. No entanto, quando foi traçado o perfil das duas versões da rotina, a primeira versão levou 4,81 segundos para copiar 5.000.000 strings e a segunda levou 4,35 segundos.

Neste caso, a versão "engenhosa" acarreta uma desvantagem de 11% na velocidade, o que a faz parecer muito menos engenhosa. Os resultados variam de um compilador para outro, mas, em geral, eles sugerem que, até que você tenha medido os ganhos de desempenho, é melhor **lutar primeiro pela clareza e pela correção, e depois pelo desempenho**.

Mesmo que você leia facilmente instruções com efeitos colaterais, tenha piedade das outras pessoas que lerão seu código. A maioria dos bons programadores precisa pensar duas vezes para entender expressões com efeitos colaterais. Deixe que eles usem seus neurônios para entender questões mais importantes de como seu código funciona, em vez dos detalhes sintáticos de uma expressão específica.

Organizando declarações de dados

Use apenas uma declaração de dados por linha Como foi mostrado nos exemplos anteriores, você deve colocar cada declaração de dados em sua própria linha. É mais fácil colocar um comentário próximo a cada declaração, caso cada uma esteja em sua própria linha. É mais fácil modificar as declarações, pois cada declaração é autocontida. É mais fácil encontrar variáveis específicas, pois você pode percorrer uma única coluna, em vez de ler cada linha. É mais fácil encontrar e corrigir erros de sintaxe, pois o número de linha que o compilador fornece contém apenas uma declaração.

Rapidamente - na declaração de dados da Listagem 52, que tipo de variável é currentBottom?

Listagem 52: Exemplo em C++ que amontoa mais de uma declaração de variável em uma linha.

```
int rowIndex, columnIdx; Color previousColor, currentColor, nextColor; Point previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom; Font previousTypeface, currentTypeface, nextTypeface; Color choices[ NUM_COLORS ];
```

Esse é um exemplo extremo, mas não está muito longe de um estilo bem mais comum, mostrado na Listagem 53:

Listagem 53: Exemplo em C++ que amontoa mais de uma declaração de variável em uma linha.

```
int rowIndex, columnIdx;
Color previousColor, currentColor, nextColor;
Point previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom;
Font previousTypef ace, currentTypeface, nextTypeface;
Color choices [ NUM_COLORS ] ;
```

Esse não é um estilo incomum de declaração de variáveis; a variável ainda é difícil de encontrar, pois todas as declarações estão amontoadas. O tipo da variável também é difícil de descobrir. Agora, qual é o tipo de nextColor na Listagem 54?

Listagem 54: Exemplo em C++ de legibilidade obtida colocando-se apenas uma declaração de variável em cada linha.

```
int rowlndex; int columnldx;
Color previousColor;
Color currentColor;
Color nextColor;
Point previousTop;
Point previousBottom;
Point currentTop;
Point currentBottom;
Point nextTop;
Point nextBottom;
Font previousTypef ace;
Font currentTypeface;
Font nextTypeface;
Color choices [ NUM_COLORS ] ;
```

A variável nextColor provavelmente foi mais fácil de encontrar do que nextTypeface, na Listagem 53. Esse estilo é caracterizado por uma declaração por linha, e uma declaração completa, incluindo o tipo da variável.

Com certeza, este estilo consome multo espaço na tela – 20 linhas, em vez das

três do primeiro exemplo, embora aquelas três linhas fossem horríveis. Não posso mencionar nenhum estudo que mostre que este estilo leve a menos erros ou a uma maior compreensão. Entretanto, se alguém me pedisse para revisar seu código e suas declarações de dados fossem parecidas com as do primeiro exemplo, eu diria: "De jeito nenhum - são difíceis demais para ler". Se elas fossem parecidas com as do segundo exemplo, eu diria; "Ahn...talvez mais tarde". Se elas fossem parecidas com o último exemplo, eu diria; "Com certeza - com todo prazer!".

Declare as variáveis próximas de onde elas são usadas pela primeira vez Um estilo que é preferível à declaração de todas as variáveis em um bloco enorme é declarar cada variável perto de onde ela é usada pela primeira vez. Isso reduz a "abrangência" e o "tempo de vida" e facilita a refatoração do código em rotinas menores, quando necessário. Para ver mais detalhes, consulte o Item "Mantenha as variáveis "vivas" pelo tempo mais curto possível", na seção 12.4.

Ordene as declarações sensatamente Na Listagem 54, as declarações são agrupadas por tipos. Agrupar por tipos normalmente é adequado, pois variáveis do mesmo tipo tendem a ser usadas em operações relacionadas. Em outros casos, você poderia optar por colocá-las em ordem alfabética, pelo nome da variável. Embora a colocação em ordem alfabética tenha muitos defensores, na minha opinião isso dá mais trabalho e não vale a pena. Caso sua lista de variáveis seja tão longa a ponto de a ordem alfabética servir de ajuda, provavelmente sua rotina é grande demais. Divida-a, para que você tenha rotinas menores, com menos variáveis.

Em C++, coloque o asterisco ao lado do nome da variável em declarações de ponteiro ou declare os tipos de ponteiro É comum ver declarações de ponteiro que colocam o asterisco ao lado do tipo, como na Listagem 55:

Listagem 55: Exemplo em C++ de asteriscos em declarações de ponteiro.

```
EmployeeList* employees;
Fila* inputFile;
```

O problema da colocação do asterisco ao lado do nome do tipo, em vez de colocá-lo ao lado do nome da variável, é que, se você colocar mais de uma declaração em uma linha, o asterisco será aplicado apenas à primeira variável, mesmo que a formatação visual sugira que ele se aplica a todas as variáveis da linha. Você pode evitar esse problema colocando o asterisco ao lado do nome da variável, em vez de colocá-lo ao lado do nome do tipo, como na Listagem 56:

Listagem 56: Exemplo em C++ de uso de asteriscos em declarações de ponteiro.

```
EmployeeList *employees;
File *inputFile;
```

Essa estratégia apresenta o problema de sugerir que o asterisco faz parte do nome da variável, o que não é verdade. A variável pode ser usada com ou sem o asterisco.

A melhor estratégia, em vez disso, é declarar um tipo para o ponteiro e usá-lo. Um exemplo aparece na Listagem 57;

Listagem 57: Exemplo em C++ de bons usos de um tipo de ponteiro em declarações.

```
EmployeeListPointer employees;
FilePointer inputFile;
```

O problema em particular tratado por essa estratégia pode ser resolvido exigindo-se que todos os ponteiros sejam declarados usando os tipos de ponteiro, como mostrado na Listagem 57, ou exigindo-se que não haja mais do que uma declaração de variável por linha. Escolha pelo menos uma dessas soluções!

12.6 Organizando comentários

Comentários bem elaborados podem melhorar muito a legibilidade de um programa; comentários mal elaborados podem prejudicá-la. O layout dos comentários desempenha um papel importante no fato de ajudarem ou atrapalharem a legibilidade.

Indente um comentário com seu código correspondente A indentação visual é uma ajuda valiosa para se entender a estrutura lógica de um programa e bons comentários não interferem na indentação visual. Por exemplo, qual é a estrutura lógica da rotina mostrada na listagem 58?

Listagem 58: Exemplo em Visual Basic de comentários mal indentados.

```
For transactionId = 1 To totalTransactions
' obtém dados da transação
    GetTransactionType ( transactionType )
    GetTransactionAmount ( transactionAmount )

' processa a transação com base no tipo
    If transactionType = Transaction_Sale Then
      AcceptCustomerSale( transactionAmount )
    Else
      If transactionType = Transaction_CustomerReturn Then
' processa o retorno automaticamente ou obtém aprovação da gerência
        If transactionAmount >= MANAGER_APPROVAL__LEVEL Then
' tenta obter a aprovação da gerência e, então, aceita ou rejeita o retorno
' com base no fato de a aprovação ser concedida
          GetMgrApproval( isTransactionApproved )
          If ( isTransactionApproved ) Then
            AcceptCustomerReturn( transactionAmount )
          Else
            RejectCustomerReturn( transactionAmount )
          End If
        Else
' aprovação da gerência não exigida; portanto, aceita o retorno
          AcceptCustomerReturn( transactionAmount )
        End If
      End If
    End If
    Next
```

Nesse exemplo, você não tem muitas pistas sobre a estrutura lógica, pois os comentários ocultam completamente a indentação visual do código. Talvez você sinta dificuldade em acreditar que alguém tome a decisão consciente de usar tal estilo de indentação, mas eu o tenho visto em programas profissionais e sei de pelo menos um livro-texto que o recomenda.

O código mostrado na Listagem 59 é exatamente igual ao da Listagem 58, a não ser pela indentação dos comentários.

Listagem 59: Exemplo em Visual Basic de comentários indentados de forma correta.

```
For transactionId = 1 To totalTransactions
    ' obtém dados da transação
    GetTransactionType ( transactionType )
    GetTransactionAmount ( transactionAmount )

    ' processa a transação com base no tipo
    If transactionType = Transaction_Sale Then
      AcceptCustomerSale( transactionAmount )

    Else
      If transactionType = Transaction_CustomerReturn Then

      ' processa o retorno automaticamente ou obtém aprovação da
      ' gerência, se necessário
        If transactionAmount >= MANAGER_APPROVAL__LEVEL Then

          ' tenta obter a aprovação da gerência e, então, aceita ou rejeita
          ' o retorno com base no fato de a aprovação ser concedida
          GetMgrApproval( isTransactionApproved )
          If ( isTransactionApproved ) Then
            AcceptCustomerReturn( transactionAmount )
          Else
            RejectCustomerReturn( transactionAmount )
          End If
        Else
        ' aprovação da gerência não exigida; portanto, aceita o retorno
          AcceptCustomerReturn( transactionAmount )
        End If
      End If
    End If
Next
```

Na Listagem 59, a estrutura lógica é mais visível Um estudo da eficácia dos comentários descobriu que a vantagem de se ter comentários não era conclusiva e o autor especulou que isso se dava porque eles “destroem a varredura visual do programa” (Shneiderman 1980). A partir desses exemplos, é evidente que o estilo do comentário influencia fortemente o fato de ele ser destruidor.

Inicie cada comentário com pelo menos uma linha em branco Se alguém estiver tentando ter uma visão geral de seu programa, a maneira mais eficiente de fazer isso é ler os comentários sem ler o código. Separar os comentários usando linhas em branco ajuda o leitor a percorrer o código. Observe o exemplo da Listagem 60:

Listagem 60: Exemplo em Java de separação de um comentário usando uma linha em branco.

```
// comentário zero
CodeStatementZero;
CodeStatementOne;

// comentário um
CodeStatementTwo;
CodeStatementThree;
```

Há quem prefira usar uma linha em branco antes e outra depois do comentário. Duas linhas em branco ocupam mais espaço na tela, mas muitos entendem que o código fica melhor do que com apenas uma. Observe o exemplo da Listagem 61:

Listagem 61: Exemplo em Java de separação de um comentário usando duas linhas em branco.

```
// comentário zero

CodeStatementZero;
CodeStatementOne;

// comentário um

CodeStatementTwo;
CodeStatementThree;
```

A não ser que seu espaço de exibição seja exíguo, esse é um julgamento puramente estético e você pode tomar a decisão conforme for adequado. Nesta área, assim como em muitas outras, o fato de existir uma convenção é mais importante do que os detalhes específicos da convenção.

12.7 Organizando rotinas

As rotinas são compostas de instruções individuais, dados, estruturas de controle, comentários - itens já discutidos neste capítulo. Esta seção fornece diretrizes de leiaute exclusivas para rotinas.

Use linhas em branco para separar as partes de uma rotina Use linhas em branco entre o cabeçalho da rotina, seus dados e declarações de constantes nomeadas (se houver), e seu corpo.

Use indentação indentação-padrão para os argumentos da rotina As opções no caso do leiaute do cabeçalho de rotina são praticamente as mesmas de muitas outras áreas do leiaute: nenhum leiaute consciente, leiaute de final de linha ou indentação-padrão. Como acontece na maioria dos outros casos, a indentação-padrão é a melhor solução em termos de precisão, consistência, legibilidade e facilidade de modificação. A Listagem 62 mostra dois exemplos de cabeçalhos de rotina sem nenhum leiaute consciente:

Listagem 62: Exemplos em C++ de cabeçalhos de rotina sem nenhum leiaute consciente:

```
bool ReadEmployeeData(int maxEmployees,EmployeeList *employees,
   EmployeeFile *inputFile,int *employeeCount,bool *isInputError)
...

void InsertionSort(SortArray data,int firstElement,int lastElement)
```

Esses cabeçalhos de rotina são puramente utilitários. O computador pode lê-los, assim como pode ler cabeçalhos em qualquer outro formato, mas eles causam problemas para seres humanos. Sem um esforço consciente para tornar os cabeçalhos difíceis de ler, como eles poderiam ser piores?

A segunda estratégia de leiaute de cabeçalho de rotina é o leiaute de fim de linha, que normalmente funciona bem. A Listagem 63 mostra os mesmos cabeçalhos de rotina reformatados:

Listagem 63: Exemplo em C++ de cabeçalhos de rotina com leiaute de final de linha medíocre.

```
bool ReadEmployeeData( int          maxEmployees,
                       EmployeeList *employees,
```

```
                        EmployeeFile *inputFile,
                        int          *employeeCount,
                        bool         *isInputError )
...
void InsertionSort ( SortArray       data,
                     int             firstElement,
                     int             lastElement )
```

A estratégia referente a final de linha é organizada e esteticamente atraente. O problema é que ela exige muito trabalho de manutenção e os estilos difíceis de manter não permanecem. Suponha que o nome da função mude de ReadEmployeeData() para ReadNewEmployeeData(). Isso tiraria o alinhamento da primeira linha em relação às outras quatro. Você teria que reformatar as outras quatro linhas da lista de parâmetros para alinhar com a nova posição de maxEmployees, causada pelo nome de função mais longo. E provavelmente ocorreria falta de espaço no lado direito, pois os elementos já estão muito à direita.

Os exemplos mostrados na Listagem 64, formatados usando a indentação-padrão, têm o mesmo apelo estético, mas exigem menos trabalho de manutenção.

Listagem 64: Exemplo em C+ + de cabeçalhos de rotina com indentação-padrão legível e fácil de manter.

```
public bool ReadEmployeeData(
    int maxEmployees,
    EmployeeList *employees,
    EmployeeFile *inputFile,
    int *employeeCount,
    bool *isInputError
)
...

public void InsertionSort(
    SortArray data,
    int firstElement,
    int lastElement
)
```

Esse estilo se mantém melhor em caso de modificação. Se o nome da rotina for alterado, a modificação não terá nenhum efeito sobre quaisquer parâmetros. Se algum parâmetro for adicionado ou excluído, somente uma linha precisará ser modificada - mais ou menos uma vírgula. Os Indícios visuais são semelhantes àqueles do esquema de indentação para um loop ou para uma instrução if. Seus olhos não precisam percorrer diferentes partes da página para cada rotina específica a fim de encontrar informações significativas: eles sabem onde as informações estão o tempo todo.

Esse estilo é traduzido para o Visual Basic de maneira simples e direta, embora exija o uso de caracteres de continuação de linha, como mostrado na Listagem 65:

Listagem 65: Exemplo em Visual Basic de cabeçalhos de rotina com Indentação-padrão legível e fácil de manter

```
Public Sub ReAdEmployeeData ( _
    ByVal maxEmployees as Integer, _
    ByRef employees As EmployeeList, _
    ByRef inputFile As EmployeeFile, _
    ByRef employeeCount As Integer, _
    ByRef isInputError As Boolean _
)
```

31.8 Organizando classes

Esta seção apresenta diretrizes para organizar código em classes. A primeira subseção descreve como organizar a interface da classe. A segunda descreve como organizar as implementações da classe. A última subseção discute a organização de arquivos e programas.

Organizando Interfaces de classe

Na organização de Interfaces de classe, convencionou se apresentar os membros da classe na seguinte ordem:

1. Comentário do cabeçalho descrevendo a classe e fornecendo todas as notas sobre a utilização global da classe
2. Construtores e destrutores
3. Rotinas públicas
4. Rotinas protegidas
5. Rotinas privadas e dados membro

Organizando Implementações de classe

As Implementações de classe geralmente são organizadas nesta ordem:

1. Comentário do cabeçalho descrevendo o conteúdo do arquivo em que a classe está
2. Dados da classe
3. Rotinas públicos
4. Rotinas protegidas
5. Rotinas privadas

Se você tiver mais de uma classe em um arquivo, identifique claramente cada uma delas As rotinas relacionadas devem ser agrupadas em classes. Um leitor que esteja percorrendo seu código deverá ser capaz de identificar facilmente cada classe. Identifique cada classe claramente, usando várias linhas em branco entre ela e as classes próximas. Uma classe é como um capitulo de um livro. Em um livro, você inicia cada capitulo em uma nova página e usa letras grandes para o titulo. Enfatize o começo de cada classe de modo semelhante. Um exemplo de separação de classes aparece na Listagem 66:

Listagem 66: Exemplo em C++ de formatação de separação entre classes

```
// cria uma string idêntica a sourceString, exceto que os
// espaços em branco são substituídos por sublinhados.
void EditString::ConvertBlanks(
    char *sourceString,
    char *targetString
    ) {
    Assert( strlen( sourceString ) <= MAX_STRING_LENGTH );
    Assert( sourceSTring != NULL );
    Assert( targetSTring != NULL );
    int charIndex = 0;
    do {
      if ( sourceString[ charIndex ] == " " ) {
        targetString[ charIndex ] = '_;'
      }
      else {
        targetString[ charIndex ] = sourceString[ charIndex ];
```

```
      }
      charIndex++;
    } while sourceString[ charIndex ] != '\0';
}
//---------------------------------------------------------------------
// FUNÇÕES MATEMÁTICAS
//
// Esta classe contém as funções matemáticas do programa.
//---------------------------------------------------------------------

// encontra o máximo aritmético de arg1 e arg2
int Math::Max( int arg1, int arg2 ) {
    if ( arg1 > arg2 ) {
      return arg1;
    }
    else {
      return arg2;
    }
}

// encontra o mínimo aritmético de arg1 e arg2
int Math::Min( int arg1, int arg2 ) {
    if ( arg1 < arg2 ) {
      return arg1;
    }
    else {
      return arg2;
    }
}
```

Evite dar ênfase demais aos comentários dentro de classes. Se você marcar cada rotina e cada comentário com uma fileira de asteriscos, em vez de usar linhas em branco terá dificuldade em sugerir um esquema que enfatize efetivamente o início de uma nova classe. A Listagem 67 mostra um exemplo:

Listagem 67: Exemplo em C++ de formatação demasiada de uma classe.

```
//*********************************************************************
//*********************************************************************
// FUNÇÕES MATEMÁTICAS
//
// Esta classe contém as funções matemáticas do programa.
//*********************************************************************
//*********************************************************************

//*********************************************************************
// encontra o máximo aritmético de argl e arg2
//*********************************************************************
int Math::Max( int argl, int arg2 ) {
//*********************************************************************
    if ( argl > arg2 ) {
      return argl;
    }
    else {
      return arg2;
    }
}
//*********************************************************************
```

```
// encontra o máximo aritmético de argl e arg2
//*********************************************************************
 int Math::Min( int argl, int arg2 ) {
//*********************************************************************
     if ( argl < arg2 ) {
       return argl;
     }
     else {
       return arg2;
     }
}
```

Nesse exemplo, são tantas coisas destacadas com asteriscos que nada é realmente enfatizado. O programa se torna uma densa floresta de asteriscos. Embora seja mais um julgamento estético do que técnico, acredite: na formatação, menos é mais.

Se você precisar separar partes de um programa com longas linhas de caracteres especiais, desenvolva uma hierarquia de caracteres (do mais denso para o mais leve), em vez de usar exclusivamente asteriscos. Por exemplo, use asteriscos para divisões de classe, traços para divisões de rotina e linhas em branco para comentários importantes. Evite colocar duas fileiras de asteriscos ou traços juntas. A Listagem 68 mostra um exemplo:

Listagem 68: Exemplo em C++ de boa formatação com restrição.

```
//*********************************************************************
// FUNÇÕES MATEMÁTICAS
//
// Esta classe contém as funções matemáticas do programa.
//*********************************************************************

//---------------------------------------------------------------------
// encontra o máximo aritmético de argl e arg2
//---------------------------------------------------------------------
int Math::Max( int argl, int arg2 ) {
     if ( argl > arg2 ) {
       return argl;
     }
     else {
       return arg2;
     }
}

//---------------------------------------------------------------------
// encontra o máximo aritmético de argl e arg2
//---------------------------------------------------------------------
 int Math::Min( int argl, int arg2 ) {
     if ( argl < arg2 ) {
       return argl;
     }
     else {
       return arg2;
     }
}
```

Essa recomendação sobre como identificar várias classes dentro de um único arquivo aplica-se somente quando sua linguagem restringe o número de arquivos que você pode usar em um programa. Se você estiver usando C++, Java, Visual Basic ou outras

linguagens que aceitam vários arquivos-fonte, coloque apenas uma classe em cada arquivo, a não ser que tenha um bom motivo para fazer o contrário (como incluir algumas classes pequenas que constituam um padrão único). Entretanto, dentro de uma classe, você ainda poderia ter subgrupos de rotinas e poderia agrupá-las usando técnicas como aquelas já mostradas aqui.

Organizando arquivos e programas

Acima das técnicas de formatação de classes está um problema de formatação maior: como você organiza as classes e rotinas dentro de um arquivo e como decide quais classes deve colocar em um arquivo, originalmente?

Coloque uma classe em um arquivo Um arquivo não é apenas um balde contendo código. Caso sua linguagem permita, um arquivo deve conter uma coleção de rotinas que suportam um, e apenas um, propósito. Um arquivo reforça a noção de que uma coleção de rotinas está na mesma classe.

Todas as rotinas dentro de um arquivo constituem a classe. A classe pode ser uma que o programa realmente reconheça como tal ou pode ser apenas uma entidade lógica que você criou como parte do projeto de seu software.

As classes são um conceito semântico da linguagem. Os arquivos são um conceito físico do sistema operacional. A correspondência entre classes e arquivo é coincidente e continua a diminuir com o passar do tempo, à medida que mais ambientes aceitam a colocação de código em bancos de dados ou ocultam de alguma forma a relação entre rotinas, classes e arquivos.

Dê ao arquivo um nome relacionado ao nome da classe A maioria dos projetos tem uma correspondência de um para um entre nomes de classe e nomes de arquivo. Uma classe denominada CustomerAccount teria arquivos denominados CustomerAccount.cpp e CustomerAccount.h, por exemplo.

Separe claramente as rotinas dentro de um arquivo Separe uma rotina das demais usando pelo menos duas linhas em branco. As linhas em branco são tão eficientes quanto grandes fileiras de asteriscos ou traços e são muito mais fáceis de digitar e manter. Use dois ou três para produzir uma diferença visual entre as linhas em branco que fazem parte de uma rotina e as que separam rotinas. Observe o exemplo mostrado na Listagem 69:

Listagem 69: Exemplo em Visual Basic do uso de linhas em branco entre rotinas.

```
' encontra o máximo aritmético de argl e arg2
Function Max( argl As Integer, arg2 As Integer ) As Integer
    If ( argl > arg2 ) Then
      Max = argl;
    Else {
      Max = arg2;
    End If
End Function



' encontra o mínimo aritmético de argl e arg2
Function Min( argl As Integer, arg2 As Integer ) As Integer
    If ( argl < arg2 ) Then
      Min = argl;
    Else {
      Min = arg2;
```

```
    End If
End Function
```

As linhas em branco são mais fáceis de digitar do que qualquer outro tipo de separador e têm uma aparência equivalente. Três linhas em branco foram usadas nesse exemplo para que a separação entre as rotinas seja mais perceptível do que as linhas em branco dentro de cada rotina.

Coloque as rotinas em ordem alfabética Uma alternativa ao agrupamento de rotinas relacionadas em um arquivo é colocá-las em ordem alfabética. Se você não pode dividir um programa em classes ou se o seu editor não permite encontrar funções facilmente, a estratégia da ordem alfabética pode economizar tempo de pesquisa.

Em C++, ordene o arquivo fonte cuidadosamente Aqui está uma ordem típica de conteúdo de arquivo-fonte em C++:

1. Comentário de descrição do arquivo
2. Arquivos #include
3. Definições de constante que se aplicam a mais de uma classe (caso haja mais de uma classe no arquivo)
4. Enumerações que se aplicam a mais de uma classe (caso haja mais de uma classe no arquivo)
5. Definições de função de macro
6. Definições de tipo que se aplicam a mais de uma classe (caso haja mais de uma classe no arquivo)
7. Variáveis globais e funções importadas
8. Variáveis globais e funções exportadas
9. Variáveis e funções privadas do arquivo
10. Classes, incluindo definições de constante, enumerações e definições de tipo dentro de cada classe

Lista de verificação: leiaute

Geral

- A formatação é feita principalmente para tomar mais clara a estrutura lógica do código?
- O esquema de formatação pode ser usado consistentemente?
- O esquema de formatação resulta em um código fácil de manter?
- O esquema de formatação melhora a legibilidade do código?

Estruturas de controle

- O código evita pares de instruções begin-end ou { } duplamente indentados?
- Os blocos sequenciais são separados uns dos outros por linhas em branco?
- As expressões complexas são formatadas de modo a favorecer a legibilidade?
- Os blocos de uma única instrução são formatados consistentemente?
- As instruções case são formatadas de uma maneira compatível com a formatação de outras estruturas de controle?
- a As instruções goto foram formatadas de maneira a tornar seu uso evidente?

Instruções individuais

- São usados espaços em branco para tornar expressões lógicas, referências de array e argumentos de rotina legíveis?
- As instruções incompletas terminam a linha de uma maneira que pareça obviamente incorreta se não forem continuadas?
- As linhas de continuação são indentadas com a quantidade de indentação-padrão?

- Cada linha contém no máximo uma instrução?
- Cada instrução é escrita sem fazer uso de efeitos colaterais?
- Há no máximo uma declaração de dados por linha?

Comentários

- Os comentários são indentados com o mesmo número de espaços que o código que comentam?
- O estilo de comentário é fácil de manter?

Rotinas

- Os argumentos de cada rotina são formatados de modo que cada argumento seja fácil de ler, modificar e comentar?
- São usadas linhas em branco para separar as partes de uma rotina?

Classes, arquivos e programas

- Existe relação de um para um entre classes e arquivos para a maioria das classes e arquivos?
- No caso de um arquivo que contém várias classes, as rotinas de cada classe estão agrupadas e cada classe está claramente identificada?
- As rotinas dentro de um arquivo estão claramente separadas por linhas em branco?
- Na ausência um princípio de organização mais rigoroso, todas as rotinas estão em ordem alfabética?

Pontos-chave

- A principal prioridade do leiaute visual é esclarecer a organização lógica do código. Os critérios usados para avaliar se essa prioridade é alcançada Incluem precisão, consistência, legibilidade e manutenibilidade.

- A boa aparência é um aspecto secundário diante dos outros critérios - um distante segundo lugar. Entretanto, se os outros critérios forem satisfeitos e o código subjacente for bom, o leiaute terá ótima aparência.

- O Visual Basic tem blocos puros e a prática convencional em lava é usar estilo de bloco puro; portanto, você pode usar um leiaute de bloco puro quando programar nessas linguagens. Em C++, tanto a simulação de bloco puro como os limites de bloco begin-end funcionam bem.

- Estruturar um código é importante para seu próprio bem. A convenção especifica que você segue é menos importante do que o fato de seguir rigorosamente alguma convenção. Uma convenção de leiaute seguida de forma inconsistente pode, na verdade, prejudicar a legibilidade.

- Muitos aspectos do leiaute são questões religiosas. Tente separar as preferências objetivas das subjetivas. Use critérios explícitos para ajudar a fundamentar suas discussões sobre preferências de estilo.