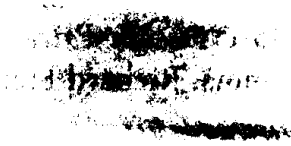
4

Interfaces



Antes de construir uma parede eu perguntaria
O que vai ficar dentro ou fora da parede,
E a quem eu poderia ofender.
Há algo em mim que não gosta de muros,
e os quer abaixo.

Robert Frost, *Mending Wall*

A essência do projeto é o equilíbrio entre objetivos e restrições concorrentes. Embora haja muitas opções a serem feitas quando se escreve um sistema pequeno e restrito, as ramificações das opções em particular permanecem dentro do sistema e afetam apenas o programador individual. Mas quando o código será usado por outras pessoas, as decisões assumem repercussões maiores.

Entre as questões a serem trabalhadas em um projeto estão:

- Interfaces: quais são os serviços e acesso fornecidos? A interface é, na verdade, um contrato entre fornecedor e cliente. O desejo é fornecer serviços que sejam uniformes e convenientes, com funcionalidade suficiente para serem fáceis de usar, mas não muito a ponto de se tornarem difíceis de gerenciar.
- Ocultamento de informações: quais informações estão visíveis e quais são privadas? Uma interface deve fornecer acesso direto aos componentes, ocultando ao mesmo tempo detalhes da implementação para que eles possam ser alterados sem afetar os usuários.
- Gerenciamento de recursos: quem é responsável por gerenciar a memória e outros recursos limitados? Aqui, os problemas principais são a alocação e liberação do armazenamento, e o gerenciamento das cópias compartilhadas de informações.
- Tratamento de erros: quem detecta os erros, quem os reporta e como? Quando um erro é detectado, qual tipo de recuperação é tentado?

No Capítulo 2 vimos as partes individuais – as estruturas de dados – a partir das quais um sistema é construído. No Capítulo 3, vimos como combinar essas partes em um programa pequeno. O tópico agora são as interfaces entre os componentes que podem vir de fontes diferentes. Neste capítulo, ilustramos o

projeto da interface, construindo uma biblioteca e estruturas de dados para uma tarefa comum. Nesse meio tempo, vamos apresentar alguns princípios do projeto. Geralmente há um número enorme de decisões a serem tomadas, mas a maioria é tomada quase que inconscientemente. Sem esses princípios, o resultado quase sempre são interfaces casuais que frustram e impedem os programadores todos os dias.

4.1 Valores separados por vírgulas

Valores separados por vírgulas, ou CSV, é o termo para uma representação natural e amplamente usada para os dados tabulares. Cada linha de uma tabela é uma linha de texto; os campos de cada linha são separados por vírgulas. A tabela no final do capítulo anterior pode começar desta maneira no formato CSV:

```
,"250MHz","400MHz","Lines of"
,"R10000","Pentium II","source code"
C,0.36 sec,0.30 sec,150
Java,4.9,9.2,105
```

Esse formato é lido e gravado por programas como as planilhas. Não é acaso o fato de que eles também aparecem nas páginas da Web de serviços, tais como cotações de preços de ações. Uma página conhecida da Web com cotações de ações apresenta esta exibição:

Símbolo	Último Pregão		Alteração		Volume
LU	2:19PM	86-1/4	+4-1/16	+4.94%	5.804,800
T	2:19PM	60-11/16	-1-3/16	-1.92%	2.468,000
MSFT	2:24PM	106-9/16	+1-3/8	+1.31%	11.474,900

Download Spreadsheet Format

A recuperação de números por meio da interação com um browser da Web é efetiva, porém demorada. É um aborrecimento invocar um browser, aguardar, observar uma infinidade de anúncios, digitar uma lista de ações, esperar, esperar, esperar, depois observar outra infinidade de anúncios, tudo para ter alguns números. Para processar mais os números é preciso mais interação ainda; selecionando o link "Download Spreadsheet Format" você recupera um arquivo contendo grande parte das mesmas informações das linhas de dados CSV como estes (editado para se ajustar):

```
"LU",86.25,"11/4/1998","2:19PM",+4.0625,
        83.9375,86.875,83.625,5804800
"T",60.6875,"11/4/1998","2:19PM",-1.1875,
        62.375,62.625,60.4375,2468000
"MSFT",106.5625,"11/4/1998","2:24PM",+1.375,
        105.8125,107.3125,105.5625,11474900
```

Evidente pela sua ausência nesse processo está o princípio de deixar a máquina trabalhar. Os browsers permitem que o seu computador acesse os dados em um servidor remoto, mas seria mais conveniente recuperar os dados sem a interação forçada. Abaixo de todos os botões que são pressionados há um procedimento puramente textual – o browser lê alguma HTML, você digita algum texto, o browser envia isso para um servidor e lê alguma HTML de volta. Com as ferramentas e a linguagem certas é fácil recuperar as informações automaticamente. Aqui temos um programa na linguagem Tcl para acessar o site na Web de cotações de ações e recuperar os dados CSV no formato acima, precedido por algumas linhas de header:

```
# getquotes.tcl: preços das ações da Lucent, AT&T, Microsoft

set so [socket quote.yahoo.com 80]   ;# conectar ao servidor
set q "/d/quotes.csv?s=LU+T+MSFT&f=sl1d1t1c1ohgv"

puts $so "GET $q HTTP/1.0\r\n\r\n"   ;# enviar solicitação
flush $so
puts [read $so]                      ;# ler e imprimir resposta
```

A seqüência crítica `f=...` que vem depois dos símbolos de verificação é uma string não documentada de controle, como o primeiro argumento de `printf`, que determina quais valores devem ser recuperados. Por experimentação, determinamos que `s` identifica o símbolo da ação, `l1` o último preço, `c1` a alteração desde ontem e assim por diante. Os detalhes não são importantes, pois estão sujeitos a alterações de qualquer forma, mas sim a possibilidade da automação: recuperar as informações desejadas e convertê-las para a forma que precisamos sem nenhuma intervenção humana. Nós podemos deixar a máquina fazer o trabalho.

Geralmente é preciso uma fração de um segundo para executar `getquotes`, bem menos do que a interação com um browser. Depois que tivermos os dados, vamos processá-los um pouco mais. Formatos de dados como CSV funcionam melhor se houver bibliotecas convenientes para converter de e para o formato, talvez aliadas a algum processamento auxiliar, tal como as conversões numéricas. Mas nós não conhecemos uma biblioteca pública para lidar com os CSV; portanto, vamos escrevê-la nós mesmos.

Nas próximas seções, vamos construir três versões de uma biblioteca para ler os dados CSV e convertê-los para uma representação interna. Nesse meio tempo, vamos falar sobre questões que surgem quando se cria software que deve funcionar com outro software. Por exemplo, não parece haver uma definição padrão para o CSV; portanto, a implementação não pode se basear em uma especificação precisa, uma situação comum no projeto das interfaces.

4.2 Uma biblioteca protótipo

É pouco provável que consigamos o projeto de uma biblioteca ou interface na primeira tentativa. Como escreveu Fred Brooks: "planeje jogar uma fora; você vai

acabar jogando mesmo". Brooks estava escrevendo sobre sistemas grandes, mas a idéia é relevante para qualquer software. Isso não é comum até você ter construído e usado uma versão do programa, quando então você já entende as questões o suficiente para projetar certo.

Dentro desse espírito, vamos abordar a construção de uma biblioteca para o CSV, construindo uma para jogar fora, um *protótipo*. Nossa primeira versão vai ignorar muitas das dificuldades de uma biblioteca feita completamente, mas será suficientemente completa para ser útil e permitir que tenhamos uma certa familiaridade com o problema.

Nosso ponto de partida é uma função `csvgetline` que lê uma linha de dados CSV de um arquivo para um buffer, a divide em campos de um array, remove as aspas e retorna o número de campos. Ao longo dos anos, escrevemos código semelhante em quase todas as linguagens que conhecemos; portanto, essa é uma tarefa conhecida. Aqui temos uma versão de protótipo em C. Nós a marcamos como questionável, porque ela é apenas um protótipo:

```
?     char buf[200];     /* buffer de linha de entrada */
?     char *field[20];   /* campos */
?
?     /* csvgetline: lê e analisa a linha, retorna a contagem de campo /
?     /* entrada de exemplo: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
?     int csvgetline(FILE *fin)
?     {
?         int nfield;
?         char *p, *q;
?
?         if (fgets(buf, sizeof(buf), fin) == NULL)
?             return -1;
?         nfield = 0;
?         for (q = buf; (p=strtok(q, ",\n\r")) != NULL; q = NULL)
?             field[nfield++] = unquote(p);
?         return nfield;
?     }
```

O comentário na parte superior da função inclui um exemplo do formato de entrada que o programa aceita; tais comentários são úteis para os programas que analisam entrada confusa.

O formato CSV é complicado demais para ser analisado facilmente por `scanf`, portanto usamos a função da biblioteca padrão `strtok`. Cada chamada de `strtok(p,s)` retorna um ponteiro para o primeiro token dentro de `p`, consistindo nos caracteres que não estão em `s`; `strtok` encerra o token sobrepondo o caractere seguinte da string original com um byte null. Na primeira chamada, o primeiro argumento de `strtok` é a string a ser examinada; as chamadas subseqüentes usam `NULL` para indicar que o exame deve retomar de onde parou na chamada anterior. Essa é uma interface fraca. Como `strtok` armazena uma variável em um lugar secreto entre as chamadas, apenas uma seqüência de chamadas pode estar ativa de cada vez. As chamadas entrelaçadas não-relacionadas interferirão umas nas outras.

Nossa função unquote remove as aspas iniciais e finais que aparecem na entrada de exemplo acima. Ela não lida com as citações aninhadas e, assim, embora seja suficiente para um protótipo ela não é geral.

```
?     /* unquote: remove as aspas iniciais e finais */
?     char *unquote(char *p)
?     {
?         if (p[0] == '"') {
?             if (p[strlen(p)-1] == '"')
?                 p[strlen(p)-1] = '\0';
?             p++;
?         }
?         return p;
?     }
```

Um programa simples de teste ajuda a verificar se csvgetline funciona:

```
?     /* csvtest main: testa a função csvgetline */
?     int main(void)
?     {
?         int i, nf;
?
?         while ((nf = csvgetline(stdin)) != -1)
?             for (i = 0; i < nf; i++)
?                 printf("field[%d] = '%s'\n", i, field[i]);
?         return 0;
?
```

printf inclui os campos dentro de apóstrofos, os quais os demarcam e ajudam a revelar os bugs que lidam incorretamente com o espaço em branco.

Agora podemos executar isso na saída produzida por getquotes.tcl:

```
% getquotes.tcl | csvtest
...
field[0] = 'LU'
field[1] = '86.375'
field[2] = '11/5/1998'
field[3] = '1:01PM'
field[4] = '-0.125'
field[5] = '86'
field[6] = '86.375'
field[7] = '85.0625'
field[8] = '2888600'
field[0] = 'T'
field[1] = '61.0625'
...
```

(Editamos as linhas do header HTTP.)

Agora temos um protótipo que parece funcionar nos dados da classificação mostrada acima. Mas seria prudente experimentá-la também em alguma outra coisa, particularmente se planejamos deixar que os outros a usem. Descobrimos outro site da Web que descarrega cotações de ações e obtivemos um arquivo com informações similares, mas em uma forma diferente: retornos de carro (\r) em vez de linhas novas para separar os registros, e sem retorno de carro de encerramento no final do arquivo. Nós o editamos e formatamos para que ele se ajustasse à página:

```
"Ticker","Price","Change","Open","Prev Close","Day High",
    "Day Low","52 Week High","52 Week Low","Dividend",
    "Yield","Volume","Average Volume","P/E"
"LU",86.313,-0.188,86.000,86.500,86.438,85.063,108.50,
    36.18,0.16,0.1,2946700,9675000,N/A
"T",61.125,0.938,60.375,60.188,61.125,60.000,68.50,
    46.50,1.32,2.1,3061000,4777000,17.0
"MSFT",107.000,1.500,105.313,105.500,107.188,105.250,
    119.62,59.00,N/A,N/A,7977300,16965000,51.0
```

Com essa entrada, nosso protótipo falhou de forma terrível.

Criamos o nosso protótipo após examinar uma fonte de dados, e o testamos originalmente apenas nos dados daquela mesma fonte. Assim sendo, não deveríamos nos surpreender quando o primeiro encontro com uma fonte diferente revelasse falhas sérias. Linhas de entrada longas, muitos campos e separadores inesperados ou faltando, tudo isso causa problemas. Esse frágil protótipo poderia servir para o uso pessoal ou para demonstrar a possibilidade de uma abordagem, mas nada além disso. Está na hora de repensar o projeto antes de experimentar outra implementação.

Tomamos um número grande de decisões no protótipo, tanto implícitas quanto explícitas. Aqui temos algumas das opções que fizemos, nem sempre da melhor maneira para uma biblioteca de fins gerais. Cada uma levanta uma questão que precisa de mais atenção.

- O protótipo não lida com as linhas de entrada longas ou com muitos campos. Ele pode dar respostas erradas ou entrar em pane porque nem mesmo verifica os overflows, muito menos retorna valores sensíveis em caso de erros.
- A entrada é assumida como consistindo em linhas terminadas por caracteres newline.
- Os campos são separados por vírgulas e as aspas são removidas. Não há provisão para a incorporação de aspas ou vírgulas.
- A linha de entrada não é preservada; ela é sobreposta no processo da criação dos campos.
- Nenhum dado é salvo de uma linha de entrada para a próxima. Se algo precisa ser lembrado, deve-se fazer uma cópia.
- O acesso aos campos é feito por meio de uma variável global, o array `field`, o qual é compartilhado por `csvgetline` e pelas funções que o chamam. Não existe nenhum controle do acesso ao conteúdo dos campos ou ponteiros.

Também não existe nenhuma tentativa de evitar o acesso além do último campo.

- As variáveis globais tornam o projeto inadequado para um ambiente multiencadeado ou mesmo para duas seqüências de chamadas entrelaçadas.
- Quem está chamando deve abrir e fechar os arquivos explicitamente; `csvgetline` lê apenas os arquivos abertos.
- A entrada e a divisão estão ligadas de forma emaranhada: cada chamada lê uma linha e a divide em campos, independente do aplicativo precisar daquele serviço.
- O valor de retorno é o número de campos da linha; cada linha deve ser dividida para calcular esse valor. Também não há como distinguir os erros do final do arquivo.
- Não há como alterar nenhuma dessas propriedades sem alterar o código.

Essa longa, porém incompleta, lista ilustra algumas das possíveis opções de projeto. Cada decisão está entrelaçada em todo o código. Ele é bom para uma tarefa rápida, como analisar um formato fixo a partir de uma fonte conhecida. Mas e se o formato mudar, ou aparecer uma vírgula dentro de uma string com aspas, ou se o servidor produzir uma linha longa ou muitos campos?

Isso aparentemente é algo simples de resolver, uma vez que a "biblioteca" é pequena e apenas um protótipo. Imagine, porém, que após ficar alguns meses ou anos na prateleira, o código se torne parte de um programa maior cuja especificação muda com o passar do tempo. Como `csvgetline` vai se adaptar? Se aquele programa for usado por outras pessoas, as opções rápidas feitas no projeto original podem criar problemas que surgirão mais tarde. Esse cenário é representativo da história de muitas interfaces ruins. É triste o fato de que muito código "rápido e sujo" acaba entrando no software de uso amplo, onde ele permanece sujo e quase sempre não tão rápido quando deveria ser.

4.3 Uma biblioteca para as outras pessoas

Usando aquilo que aprendemos com o protótipo, agora queremos construir uma biblioteca de uso geral. O requisito mais óbvio é que devemos tornar `csvgetline` mais robusta, para que ela trate de linhas longas ou de muitos campos. Ela também deve ser mais cuidadosa na análise dos campos.

Para criar uma interface que as outras pessoas possam usar, vamos considerar as questões listadas no início deste capítulo: interfaces, ocultamento de informações, gerenciamento de recursos e tratamento de erros. O relacionamento entre essas questões afeta bastante o projeto. Nossa separação para essas questões é um pouco arbitrária, uma vez que elas estão inter-relacionadas.

Interface. Optamos por três operações básicas:

`char *csvgetline(FILE *)`: lê uma linha CSV nova
`char *csvfield(int n)`: retorna o campo de número `n` da linha atual
`int csvnfield(void)`: retorna o número de campos da linha atual

Qual valor de função deve ser retornado por `csvgetline`? É desejável que se retorne o máximo conveniente de informações úteis, o que sugere o retorno do número de campos, como no protótipo. Mas o número de campos deve ser calculado mesmo quando os campos não estão sendo usados. Outro valor possível é o comprimento da linha de entrada, que é afetado quando a newline do final é preservada. Depois de várias experiências, resolvemos que `csvgetline` retornará um ponteiro para a linha original de entrada, ou `NULL` se o final do arquivo foi atingido.

Vamos remover a newline do final da linha retornada por `csvgetline`, uma vez que ela pode ser facilmente restaurada se for preciso.

A definição de um campo é complicada. Tentamos fazer coincidir aquilo que observamos empiricamente nas planilhas e em outros programas. Um campo é uma seqüência de zero ou mais caracteres. Os campos são separados por vírgulas. Os espaços em branco no início e no final são preservados. Um campo pode estar incluído dentro de caracteres de aspas, caso em que ele pode conter vírgulas. Um campo com aspas pode conter caracteres de aspas, o que é representado por um caractere de aspas; o campo CSV `"x""y"` define a string `x"y`. Os campos podem ser vazios; um campo especificado como `" "` é vazio e idêntico àquele especificado pelas vírgulas adjacentes.

Os campos são numerados a partir do zero. E se o usuário pedir um campo não-existente chamando `csvfield(-1)` ou `csvfield(100000)`? Poderíamos retornar `" "` (a string vazia), porque ela pode ser impressa e comparada. Os programas que processam os números de variável dos campos não teriam de tomar precauções especiais para lidar com os números inexistentes. Mas essa opção não fornece uma maneira de distinguir o vazio do não-existente. Uma segunda opção seria imprimir uma mensagem de erro ou mesmo abortar. Vamos discutir em breve o motivo pelo qual isso não é desejável. Resolvemos retornar `NULL`, o valor convencional para uma string não-existente em C.

Ocultamento de informações. A biblioteca não limitará o comprimento de entrada de linha ou o número de campos. Para conseguir isso, quem chama deve fornecer a memória ou quem é chamado (a biblioteca) deve alocá-la. Quem chama a função de biblioteca `fgets` passa um array e um tamanho máximo. Se a linha for mais longa do que o buffer, ela é dividida em partes. Esse comportamento não é satisfatório para a interface CSV; portanto, nossa biblioteca alocará a memória quando descobrir que mais memória é necessária.

Assim sendo, apenas `csvgetline` sabe sobre o gerenciamento da memória. Nada sobre a maneira como ela organiza a memória pode ser acessado de fora. A melhor maneira de fornecer esse isolamento é usar a interface de função: `csvgetline` lê a linha seguinte, independente do seu tamanho, `csvfield(n)` retorna um ponteiro para os bytes do campo de número `n` da linha atual, e `csvnfield` retorna o número de campos da linha atual.

Precisaremos aumentar a memória à medida que chegarem linhas mais longas ou mais campos. Os detalhes de como isso é feito estão ocultos nas funções `csv`; nenhuma outra parte do programa sabe como isso funciona, por exemplo, se a biblioteca usa arrays pequenos que podem ser aumentados, ou arrays muito grandes,

ou algo completamente diferente. A interface também não revela quando a memória é liberada.

Se o usuário chamar apenas `csvgetline`, não há necessidade de dividi-la em campos; as linhas podem ser divididas conforme a demanda. Outro detalhe da implementação que é oculto do usuário é se a divisão de campos é "ansiosa" (feita imediatamente quando a linha está pronta), "preguiçosa" (feita apenas quando um campo ou contagem forem necessários) ou "muito preguiçosa" (somente o campo solicitado é dividido).

Gerenciamento de recursos. Devemos resolver quem é responsável pelo compartilhamento das informações. `csvgetline` retorna os dados originais ou faz uma cópia? Resolvemos que o valor de retorno de `csvgetline` é um ponteiro para a entrada original, o qual é sobreposto quando a próxima linha é lida. Os campos serão construídos em uma cópia da linha de entrada, e `csvfield` retornará um ponteiro para o campo dentro da cópia. Com essa organização, o usuário deve fazer outra cópia quando uma determinada linha ou um campo devem ser salvos ou alterados, e é responsabilidade do usuário liberar aquele armazenamento quando ele não é mais necessário.

Quem abre e fecha o arquivo de entrada? Seja quem for, ele deve fazer o fechamento correspondente: as tarefas coincidentes devem ser feitas no mesmo nível ou lugar. Vamos assumir que `csvgetline` é chamada com um ponteiro `FILE` para um arquivo já aberto que quem chamou vai fechar, quando o processamento estiver concluído.

O gerenciamento dos recursos compartilhados ou passados pelo limite entre uma biblioteca e aquele de quem chama é uma tarefa difícil, e quase sempre há razões sólidas mas conflitantes para preferir as diversas opções de projeto. Os erros e mal-entendidos sobre as responsabilidades do compartilhamento são fonte freqüente de bugs.

Tratamento de erros. Como `csvgetline` retorna `NULL`, não há uma maneira boa para distinguir o final do arquivo de um erro como falta de memória. Da mesma forma, o acesso a um campo não-existente não causa erro. Por analogia com `ferror`, poderíamos adicionar outra função `csvgeterror` à interface para reportar o erro mais recente, mas, por questões de simplicidade, vamos deixá-la de fora desta versão.

Como princípio, as rotinas de biblioteca não devem simplesmente morrer quando ocorrer um erro. O status de erro deve ser retornado para quem chama para a ação apropriada. As rotinas de biblioteca também não devem imprimir mensagens ou mostrar caixas de diálogo, uma vez que elas podem estar sendo executadas em um ambiente onde uma mensagem interferiria em outra coisa. O tratamento de erros é um tópico que merece uma discussão própria, o que será feito mais tarde neste capítulo.

Especificação. As opções feitas acima devem ser reunidas em um lugar como uma especificação dos serviços fornecidos por `csvgetline` e de como ela deve ser usada. Em um projeto grande, a especificação precede a implementação, porque os

especificadores e implementadores geralmente são pessoas diferentes e podem estar em organizações diferentes. Na prática, porém, o trabalho quase sempre é feito em paralelo, com a especificação e o código evoluindo juntos, embora eventualmente a "especificação" seja escrita apenas depois do fato para descrever aproximadamente o que o código faz.

A melhor abordagem é escrever a especificação no início e revisá-la à medida que a implementação progride. Quanto mais precisa e cuidadosa for uma especificação, maior a chance de que o programa resultante funcione bem. Mesmo nos programas pessoais, é bom preparar uma especificação razoavelmente completa, porque isso encoraja a consideração das alternativas e registra as opções feitas.

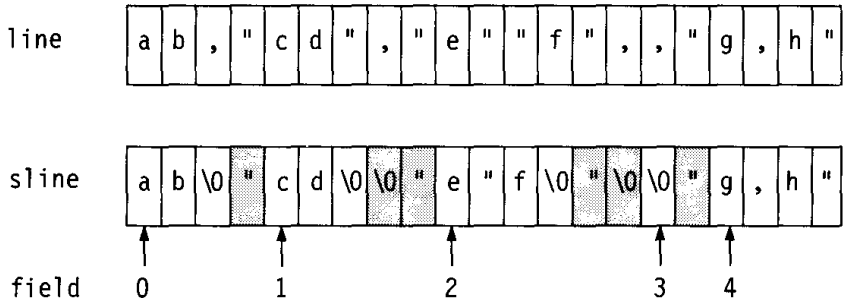
Para nossos propósitos, a especificação incluiria os protótipos de função e uma prescrição detalhada do comportamento, das responsabilidades e suposições:

```
Os campos são separados por vírgulas.
Um campo pode estar entre caracteres de aspas "...".
Um campo com aspas pode conter vírgulas, mas não pode conter newlines.
Um campo com aspas pode conter caracteres de aspas ", representados por
" ".
Os campos podem estar vazios; " " e uma string vazia representam um campo
vazio.
O espaço em branco inicial e final é preservado.

char *csvgetline(FILE *f);
  lê uma linha do arquivo de entrada aberto f;
        assume que as linhas de entrada terminam com \r,\n,\r\n, ou EOF.
  retorna o ponteiro para a linha, com o terminador removido, ou NULL se
    EOF ocorreu.
  a linha pode ter um comprimento arbitrário; retorna NULL se o limite de
  memória é excedido.
  a linha deve ser tratada como um armazenamento somente para leitura;
        quem chama deve fazer uma cópia para preservar ou alterar o
  conteúdo.

char *csvfield(int n);
  os campos são numerados a partir do 0.
  retorna o campo de número n da última linha lida por csvgetline;
        retorna NULL se n < 0 ou além do último campo.
  os campos são separados por vírgulas
  os campos podem estar dentro de " ..."; tais aspas são removidas;
        dentro de " ...", " " são substituídas por " e a vírgula
        não é um separador.
  nos campos sem aspas, as aspas são caracteres regulares.
  pode haver um número arbitrário de campos de qualquer comprimento;
        retorna NULL se o limite de memória é excedido.
  o campo deve ser tratado como um armazenamento somente para leitura;
        quem chama deve fazer uma cópia para preservar ou alterar o
        conteúdo.
  comportamento não definido se chamado antes de csvgetline ser chamada.
```

"e""f",,"g,h" foram processadas. Os elementos sombreados em sline não fazem parte de nenhum campo.



Aqui temos a própria função csvgetline:

```
/* csvgetline: obtém uma linha, aumenta conforme a necessidade */
/* entrada de exemplo: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
char *csvgetline(FILE *fin)
{
    int i, c;
    char *newl, *news;

    if (line == NULL) {       /* aloca na primeira chamada */
        maxline = maxfield = 1;
        line = (char *) malloc(maxline);
        sline = (char *) malloc(maxline);
        field = (char **) malloc(maxfield*sizeof(field[0]));
        if (line == NULL || sline == NULL || field == NULL) {
            reset();
            return NULL;      /* sem memória */
         }
    }
    for (i=0; (c=getc(fin))!=EOF && !endofline(fin,c); i++) {
        if (i >= maxline-1) { /* aumentar linha */
            maxline *= 2; /* dobrar o tamanho atual */
            newl = (char *) realloc(line, maxline);
            news = (char *) realloc(sline, maxline);
            if (newl == NULL || news == NULL) {
                reset();
                return NULL;  /* sem memória */
            }
            line = newl;
            sline = news;
        }
        line[i] = c;
    }
    line[i] = '\0';
    if (split() == NOMEM) {
```

```
            reset();
            return NULL; /* sem memória */
        }
        return (c == EOF && i == 0) ? NULL : line;
    }
```

Uma linha recebida é acumulada em `line`, a qual é aumentada conforme a necessidade por uma chamada de `realloc`; o tamanho é dobrado a cada aumento, como na Seção 2.6. O array `sline` é mantido com o mesmo tamanho de `line`; `csvgetline` chama `split` para criar os ponteiros de campo de um `field` de array separado, o qual também é aumentado conforme a necessidade.

Como estamos personalizando, começamos o array muito pequeno e o aumentamos conforme a demanda, para garantir que o código de aumento de array seja exercido. Se a alocação falhar, chamamos `reset` para restaurar os globais ao seu estado inicial, de modo que uma chamada subseqüente a `csvgetline` tenha chances de sucesso:

```
/* reset: define as variáveis de volta aos valores iniciais */
static void reset(void)
{
    free(line); /* free(NULL) permitido pela ANSI C */
    free(sline);
    free(field);
    line = NULL;
    sline = NULL;
    field = NULL;
    maxline = maxfield = nfield = 0;
}
```

A função `endofline` trata do problema de uma linha de entrada ser encerrada com um retorno de carro, uma newline, ambos, ou mesmo com um `EOF`:

```
/* endofline: verificar e consumir \r, \n, \r\n, or EOF */
static int endofline(FILE *fin, int c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        c = getc(fin);
        if (c != '\n' && c != EOF)
            ungetc(c, fin); /* ler bem adiante, colocar c de volta */
    }
    return eol;
}
```

Uma função separada é necessária, uma vez que as funções padrão de entrada não lidam com a rica variedade de formatos teimosos encontrados nas entradas reais.

Nosso protótipo usou `strtok` para encontrar o próximo token pesquisando um caractere separador, normalmente uma vírgula, mas isso impossibilitou o tratamento das vírgulas com aspas. Uma grande alteração na implementação de `split` se faz necessária, embora sua interface não precise mudar. Pense nestas linhas de entrada:

```
"",,""
,"",
,,
```

Cada linha tem três campos vazios. Verificar se `split` os analisa e às outras entradas corretamente é algo que complica as coisas de forma significativa, um exemplo de como os casos especiais e as condições-limite podem dominar um programa.

```
/* split: divide a linha em campos */
static int split(void)
{
    char *p, **newf;
    char *sepp; /* ponteiro para caractere separador temporário */
    int sepc;   /* caractere separador temporário */

    nfield = 0;
    if (line[0] == '\0')
        return 0;
    strcpy(sline, line);
    p = sline;

    do  {
        if (nfield >= maxfield) {
            maxfield *= 2;          /* dobra o tamanho atual */
            newf = (char **) realloc(field,
                        maxfield * sizeof(field[0]));
            if (newf == NULL)
                return NOMEM;
            field = newf;
        }
        if (*p == '"')
            sepp = advquoted(++p);  /* saltar aspas iniciais */
        else
            sepp = p + strcspn(p, fieldsep);
        sepc = sepp[0];
        sepp[0] = '\0';             /* encerrar campo */
        field[nfield++] = p;
        p = sepp + 1;
    } while (sepc == ',');

    return nfield;
}
```

O loop aumenta o array dos ponteiros de campo se for preciso, e depois chama uma das duas funções para localizar e processar o próximo campo. Se o campo começar com aspas, `advquoted` encontra o campo e retorna um ponteiro para o separador que encerra o campo. Caso contrário, para localizar a próxima vírgula usamos a função de biblioteca `strcspn(p,s)`, que pesquisa em uma string `p` a próxima ocorrência de qualquer caractere da string `s`; ela retorna o número de caracteres que foi saltado.

As aspas dentro de um campo são representadas por duas aspas adjacentes, de modo que `advquoted` as espreme em uma única; ela também remove as aspas que cercam o campo. Alguma complexidade é adicionada por uma tentativa de lidar com entradas plausíveis que não coincidem com a especificação, tais como `"abc"def`. Em tais casos, anexamos o que vier depois das segundas aspas até o próximo separador como parte desse campo. O Microsoft Excel parece usar um algoritmo semelhante.

```
/* advquoted: campo com aspas; retornar ponteiro para o próximo
   separador */
static char *advquoted(char *p)
{
    int i, j;

    for (i = j = 0; p[j] != '\0'; i++, j++) {
        if (p[j] == '"' && p[++j] != '"') {
            /* copiar até o próximo separador ou \0 */
            int k = strcspn(p+j, fieldsep);
            memmove(p+i, p+j, k);
            i += k;
            j += k;
            break;
        }
        p[i] = p[j];
    }
    p[i] = '\0';
    return p + j;
}
```

Como a linha de entrada já está dividida, `csvfield` e `csvnfield` são triviais:

```
/* csvfield: retorna ponteiro para o nº campo */
char *csvfield(int n)
{
    if (n < 0 || n >= nfield)
        return NULL;
    return field[n];
}

/* csvnfield: retorna o número de campos */
int csvnfield(void)
```

```
{
    return nfield;
}
```

Finalmente, podemos modificar o driver de teste para exercitar essa versão da biblioteca. Como ele mantém uma cópia da linha de entrada, o que o protótipo não faz, ele pode imprimir a linha original antes de imprimir os campos:

```
/* csvtest main: testa a biblioteca CSV */
int main(void)
{
    int i;
    char *line;

    while ((line = csvgetline(stdin)) != NULL) {
        printf("line = `%s'\n", line);
        for (i = 0; i < csvnfield(); i++)
            printf("field[%d] = `%s'\n", i, csvfield(i));
    }
    return 0;
}
```

Isso conclui nossa versão em C. Ela lida arbitrariamente com entradas grandes e faz algo sensível até mesmo com os dados mais teimosos. O preço é que ela é mais do que quatro vezes maior do que o primeiro protótipo e parte do código é complicada. Tal expansão no tamanho e complexidade é um resultado típico da passagem do protótipo para a produção. □

Exercício 4-1. Há vários graus de "preguiça" na divisão de campos. Entre as possibilidades está dividir tudo de uma vez, mas somente quando alguns campos são solicitados, dividir apenas o campo solicitado ou dividir até o campo solicitado. Enumere as possibilidades, avalie a dificuldade e os benefícios em potencial e depois escreva-os e meça suas velocidades. □

Exercício 4-2. Adicione um recurso para que os separadores sejam alterados (a) para uma classe arbitrária de caracteres; (b) para separadores diferentes para campos diferentes; (c) para uma expressão regular (consulte o Capítulo 9). Como deve ser a interface? □

Exercício 4-3. Preferimos usar a inicialização estática fornecida pela C como a base de um deslocamento feito de uma só vez: quando um ponteiro é `NULL` na entrada, a inicialização é realizada. Outra possibilidade é exigir que o usuário chame uma função explícita de inicialização, a qual pode incluir os tamanhos iniciais sugeridos para os arrays. Implemente uma versão que combina o melhor de ambas. Qual é o papel de `reset` em sua implementação? □

Exercício 4-4. Projete e implemente uma biblioteca para criar dados formatados como CSV. A versão mais simples pode tomar um array de strings e imprimi-los com aspas e vírgulas. Uma versão mais sofisticada usaria uma string de formato semelhante a `printf`. O Capítulo 9 traz algumas sugestões de notação. □

4.4 Uma implementação em C++

Nesta seção, vamos escrever uma versão em C++ da biblioteca CSV para abordar algumas das limitações restantes da versão em C. Isso exigirá algumas alterações na especificação, das quais a mais importante é que as funções lidarão com as strings da C++ em vez dos arrays de caracteres da C. O uso das strings da C++ resolverá automaticamente algumas das questões de gerenciamento do armazenamento, uma vez que as funções de biblioteca gerenciarão a memória para nós. Em particular, as rotinas de campo retornarão strings que podem ser modificadas por quem chama, um projeto mais flexível do que as versões anteriores.

Uma classe `csv` define a face pública, enquanto oculta as variáveis e funções da implementação. Como um objeto de classe contém todo o estado de uma instância, podemos instanciar diversas variáveis `csv`; cada uma é independente das outras, de modo que vários fluxos de entrada CSV podem operar ao mesmo tempo.

```
class Csv { // lê e analisa os valores separados por vírgulas
    // entrada de exemplo: "LU",86.25,"11/4/1998","2:19PM",+4.0625

  public:
    Csv(istream& fin = cin, string sep = ",") :
        fin(fin), fieldsep(sep) {}

    int getline(string&);
    string getfield(int n);
    int getnfield() const { return nfield; }

  private:
    istream& fin;         // ponteiro para o arquivo de entrada
    string line;          // linha de entrada
    vector<string> field; // strings de campo
    int nfield;           // número de campos
    string fieldsep;      // caracteres separadores

    int split();
    int endofline(char);
    int advplain(const string& line, string& fld, int);
    int advquoted(const string& line, string& fld, int);
};
```

Os parâmetros padrão do construtor são definidos para que um objeto csv padrão seja lido a partir do fluxo padrão de entrada e use o separador de campo normal. Qualquer um dos dois pode ser substituído por valores explícitos.

Para gerenciar as strings, a classe usa as classes `string` e `vector` padrão da C++, em vez das strings no estilo da C. Não há estado não-existente para uma `string`: "empty" significa apenas que o comprimento é zero, e não há equivalente para NULL, portanto nós não podemos usá-lo como um sinal de final de arquivo. Assim sendo, `Csv::getline` retorna a linha de entrada por meio de um argumento por referência, reservando o próprio valor de função para o final de arquivo e relatórios de erro.

```
// getline: obtém uma linha, aumenta conforme a necessidade
int Csv::getline(string& str)
{
    char c;

    for (line = ""; fin.get(c) && !endofline(c); )
        line += c;
    split();
    str = line;
    return !fin.eof();
}
```

O operador += é sobrecarregado para anexar um caractere a uma string.

Pequenas alterações são necessárias em `endofline`. Novamente, temos de ler a entrada um caractere de cada vez, pois nenhuma das rotinas de entrada padrão pode lidar com a variedade de entradas.

```
// endofline: verifica e consome \r, \n, \r\n, or EOF
int Csv::endofline(char c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        fin.get(c);
        if (!fin.eof() && c != '\n')
            fin.putback(c); // read too far
    }
    return eol;
}
```

Aqui temos a versão nova de `split`:

```
// split: divide as linhas em campos
int Csv::split()
{
    string fld;
```

```
    int i, j;

    nfield = 0;
    if (line.length() == 0)
        return 0;
    i = 0;

    do {
        if (i < line.length() && line[i] == '"')
            j = advquoted(line, fld, ++i); // saltar aspas
        else
            j = advplain(line, fld, i);
        if (nfield >= field.size())
            field.push_back(fld);
        else
            field[nfield] = fld;
        nfield++;
        i = j + 1;
    } while (j < line.length());

    return nfield;
}
```

Como strcspn não funciona nas strings da C++, nós devemos alterar tanto split quanto advquoted. A nova versão de advquoted usa a função padrão da C++ find_first_of para localizar a próxima ocorrência de um caractere de separador. A chamada s.find_first_of(fieldsep,j) pesquisa na string s a primeira instância de qualquer caractere em fieldsep que ocorre em ou após a posição j. Se ela não encontrar uma instância, retorna um índice além do final da string, de modo que devemos colocá-la de novo dentro do intervalo. O loop interno for que se segue anexa os caracteres até o separador para o campo que está sendo acumulado em fld.

```
// advquoted: campo com aspas; retorna o índice do próximo separador
int Csv::advquoted(const string& s, string& fld, int i)
{
    int j;

    fld = "";
    for (j = i; j < s.length(); j++) {
        if (s[j] == '"' && s[++j] != '"') {
            int k = s.find_first_of(fieldsep, j);
            if (k > s.length()) // nenhum separador encontrado
                k = s.length();
            for (k -= j; k-- > 0; )
                fld += s[j++];
            break;
        }
```

```
            fld += s[j];
        }
        return j;
    }
```

A função `find_first_of` também é usada em uma função nova `advplain`, a qual avança sobre um campo simples sem aspas. Novamente, essa alteração é necessária porque as funções string da C como `strcspn` não podem ser aplicadas às strings da C++, as quais são tipos de dados totalmente diferentes.

```
// advplain: campo sem aspas; retorna o índice do próximo separador
int Csv::advplain(const string& s, string& fld, int i)
{
    int j;

    j = s.find_first_of(fieldsep, i);   // procura o separador
    if (j > s.length())                 // nenhum encontrado
        j = s.length();
    fld = string(s, i, j-i);
    return j;
}
```

Como antes, `Csv::getfield` é comum, enquanto que `Csv::getnfield` é tão curto que é implementado na definição da classe.

```
// getfield: retorna o campo nº
string Csv::getfield(int n)
{
    if (n < 0 || n >= nfield)
        return "";
    else
        return field[n];
}
```

Nosso programa de teste é uma variante simples daquela anterior:

```
// Csvtest main: testa a classe Csv
int main(void)
{
    string line;
    Csv csv;

    while (csv.getline(line) != 0) {
        cout << "line = '" << line <<"'\n";
        for (int i = 0; i < csv.getnfield(); i++)
            cout << "field[" << i << "] = '"
                 << csv.getfield(i) << "'\n";
    }
```

```
    return 0;
}
```

O uso é diferente daquele da versão em C, embora somente até certo ponto. Dependendo do compilador, a versão em C++ é cerca de 40% a quatro vezes mais lenta do que a versão em C em um arquivo de entrada grande com 30.000 linhas e cerca de 25 campos por linha. Como vimos ao comparar as versões da `markov`, essa variação é um reflexo da maturidade da biblioteca. O programa-fonte em C++ é cerca de 20% mais curto.

Exercício 4-5. Aperfeiçoe a implementação em C++ para sobrecarregar o subscripting com `operator[]` para que os campos possam ser acessados como `csv[i]`. □

Exercício 4-6. Escreva uma versão em Java da biblioteca CSV, depois compare as três implementações quanto à clareza, robustez e velocidade. □

Exercício 4-7. Faça novamente o package da versão em C++ do código CSV como um STL iterator. □

Exercício 4-8. A versão em C++ permite que várias instâncias independentes `csv` operem simultaneamente sem interferência, um benefício do encapsulamento de todo o estado em um objeto que pode ser instanciado várias vezes. Modifique a versão em C para conseguir o mesmo efeito substituindo as estruturas de dados globais pelas estruturas que são alocadas e inicializadas por uma função `csvnew` explícita. □

4.5 Princípios da interface

Nas seções anteriores vimos os detalhes de uma interface, que é o limite detalhado entre o código que fornece um serviço e o código que o usa. Uma interface define aquilo que alguma parte do código faz para seus usuários, como as funções e talvez os membros de dados podem ser usados pelo restante do programa. Nossa interface CSV fornece três funções – lê uma linha, obtém um campo e retorna o número de campos – que são as únicas operações que podem ser executadas.

Para prosperar, uma interface deve estar bem adaptada à sua tarefa – simples, geral, regular, previsível, robusta – e deve se adaptar bem aos seus usuários e às alterações nas suas implementações. As boas interfaces seguem um conjunto de princípios. Estes não são independentes, nem mesmo consistentes, mas eles nos ajudam a descrever o que acontece na fronteira entre dois softwares.

Ocultar os detalhes da implementação. A implementação por trás da interface deve ser oculta do restante do programa, para que ele possa ser alterado sem afetar ou quebrar nada. Existem diversos termos para esse tipo de princípio de

organização: ocultamento de informações, encapsulamento, abstração, modularização e outros que se referem todos às idéias relacionadas. Uma interface deve ocultar os detalhes da implementação que são irrelevantes para o cliente (usuário) da interface. Os detalhes que são invisíveis podem ser alterados sem afetar o cliente, talvez para estender a interface, torná-la mais eficiente ou mesmo substituir toda a sua implementação.

As bibliotecas básicas da maioria das linguagens de programação fornecem exemplos conhecidos, embora nem sempre eles sejam bem projetados. A biblioteca padrão de E/S da C está entre as mais conhecidas: algumas dezenas de funções que abrem, fecham, lêem, gravam e manipulam os arquivos de outras formas. A implementação da E/S de arquivo é oculta por trás de um tipo de dados `FILE*`, cujas propriedades podem ser vistas (porque quase sempre elas são declaradas em `<stdio.h>`), mas não se deve explorá-las.

Se o arquivo header não incluir a declaração real de estrutura mas apenas o nome da estrutura, isso também é chamado de *tipo opaco*, uma vez que suas propriedades não são visíveis e todas as operações ocorrem por meio de um ponteiro para qualquer objeto real que esteja se escondendo.

Evite as variáveis globais. Sempre que possível é melhor passar as referências para todos os dados por meio dos argumentos de função.

Não recomendamos os dados publicamente visíveis em todos os formulários; fica muito difícil manter a consistência dos valores quando os usuários podem alterar as variáveis à vontade. As interfaces de função facilitam a implantação das regras de acesso, mas esse princípio quase sempre é violado. Os fluxos de E/O como `stdin` e `stdout` quase sempre são definidos como elementos de um array global de estruturas `FILE`:

```
extern FILE      __iob[_NFILE];
#define stdin    (&__iob[0])
#define stdout   (&__iob[1])
#define stderr   (&__iob[2])
```

Isso torna a implementação totalmente visível. Isso também significa que é possível atribuir `stdin`, `stdout` ou `stderr`, muito embora elas se pareçam com variáveis. O nome peculiar `__iob` usa a convenção ANSI C de dois caracteres de sublinhado na frente para nomes privados que devem ser visíveis, o que faz com que os nomes tenham menos chances de causar conflitos em um programa.

As classes da C++ e Java são mecanismos melhores para ocultar as informações; elas são importantes para o uso adequado daquelas linguagens. As classes de contêiner da C++ Standard Template Library que usamos no Capítulo 3 levam isso ainda mais adiante: além de algumas garantias de desempenho não há informações sobre a implementação e os criadores da biblioteca podem usar qualquer mecanismo que quiserem.

Selecionar um conjunto ortogonal pequeno de primitivos. Uma interface deve fornecer o máximo de funcionalidade necessária, mas nada além disso, e as funções não devem se sobrepor excessivamente em suas capacidades. Muitas funções

podem tornar a biblioteca mais fácil de usar – tudo de que alguém precisa está lá para ser usado. Mas uma interface grande é mais difícil de escrever e atualizar, e um tamanho muito grande também pode torná-la difícil de aprender e usar. As "interfaces de programas aplicativos", ou APIs, eventualmente são tão imensas que não se espera que nenhum mortal possa dominá-las.

Em nome da conveniência, algumas interfaces fornecem várias maneiras de fazer a mesma coisa, uma tendência à qual devemos resistir. A biblioteca de E/S padrão da C fornece pelo menos quatro funções diferentes que gravarão um único caractere em um fluxo de saída:

```
char c;
putc(c, fp);
fputc(c, fp);
fprintf(fp, "%c", c);
fwrite(&c, sizeof(char), 1, fp);
```

Se o fluxo for `stdout`, existem várias outras possibilidades. Elas são convenientes, mas nem todas são necessárias.

As interfaces menores devem ser usadas em vez das grandes, pelo menos até que se tenham fortes evidências da necessidade de mais funções. Faça uma coisa, e faça-a bem. Não aumente uma interface só porque é possível fazer isso, e não conserte a interface quando os problemas estão na implementação. Por exemplo, em vez de ter `memcpy` por questões de velocidade e `memmove` por questões de segurança, seria melhor ter uma função que sempre foi segura e que é rápida quando pode ser.

Não saia do alcance do usuário. Uma função de biblioteca não deve escrever arquivos e variáveis secretos ou alterar os dados globais, e deve ser circunspecta quanto à modificação dos dados em seu chamador. A função `strtok` falha em vários desses critérios. É um pouco surpreendente que `strtok` escreva bytes null no meio de sua string de entrada. Seu uso do ponteiro null como um sinal para retomar de onde parou implica dados secretos mantidos entre as chamadas, uma fonte provável de bugs, e exclui os usos simultâneos da função. Um projeto melhor seria fornecer uma única função que faz o token de uma string de entrada. Por questões semelhantes, nossa segunda versão da C não pode ser usada para dois fluxos de entrada. Consulte o Exercício 4-8.

O uso de uma interface não deve exigir outra apenas para ser conveniente para o criador ou implementador da interface. Em vez disso, faça a interface autocontida, ou na sua falta, seja explícito sobre quais serviços externos são requeridos. Caso contrário, a carga da manutenção fica por conta do cliente. Um exemplo óbvio é o aborrecimento de gerenciar listas enormes de arquivos header na fonte em C e C++; os arquivos header podem ter milhares de linhas de comprimento e incluir dezenas de outros headers.

Faça a mesma coisa igual em todos os lugares. A consistência e a regularidade são importantes. Coisas relacionadas devem ser realizadas por meios relacionados.

As funções `str...` básicas da biblioteca C são fáceis de usar sem documentação, porque elas todas se comportam mais ou menos igual: os dados fluem da direita para a esquerda, na mesma direção que em uma declaração de atribuição, e todas retornam a string resultante. Por outro lado, na biblioteca C Standard I/O fica difícil prever a ordem dos argumentos para as funções. Algumas têm o argumento `FILE*` primeiro, outras por último. Outras têm diversas ordens para tamanho e número de elementos. Os algoritmos para os contêineres STL apresentam uma interface bastante uniforme, portanto é fácil prever como usar uma função desconhecida.

A consistência externa, comportando-se como alguma outra coisa, também é um objetivo. Por exemplo, as funções `mem...` foram criadas depois das funções `str...` na C, mas pediram emprestado seu estilo. As funções de E/S padrão `fread` e `fwrite` seriam mais fáceis de lembrar se fossem parecidas com as funções `read` e `write` nas quais se basearam. As opções da linha de comandos do Unix são introduzidas com um sinal de menos, mas determinada letra de opção pode significar coisas completamente diferentes, mesmo entre programas relacionados.

Se os caracteres curingas, como o * de `*.exe`, forem todos expandidos por um intérprete de comandos, o comportamento é uniforme. Se eles forem expandidos por programas individuais, provavelmente o comportamento será uniforme. Os browsers da Web usam um único clique do mouse para seguir um link, mas outros aplicativos usam dois cliques para iniciar um programa e seguir um link. O resultado é que muitas pessoas clicam automaticamente duas vezes.

Esses princípios são mais fáceis de seguir em alguns ambientes do que em outros, mas eles continuam valendo para ambos. Por exemplo, é difícil ocultar os detalhes de implementação na C, mas um bom programador não os explora, porque isso torna os detalhes parte da interface e viola o princípio do ocultamento de informações. Comentários nos arquivos header, nomes com formas especiais (tais como `__iob`) e outros são maneiras de incentivar o bom comportamento quando ele não for implantado.

Não há limite para aquilo que podemos fazer para criar uma boa interface. Mesmo as melhores interfaces de hoje podem eventualmente se tornar os problemas de amanhã, mas o bom projeto pode fazer com que o amanhã demore um pouco mais para chegar.

4.6 Gerenciamento de recursos

Um dos problemas mais difíceis da criação da interface de uma biblioteca (ou de uma classe ou de um pacote) é gerenciar os recursos que são propriedade da biblioteca ou que são compartilhados pela biblioteca e por aqueles que a chamam. O recurso mais óbvio deles é a memória – quem é responsável por alocar e liberar o armazenamento? – mas outros recursos compartilhados exibem os arquivos abertos e o estado das variáveis cujos valores são de interesse comum. De forma geral, as questões se classificam nas categorias da inicialização, manutenção do estado, compartilhamento, cópia e limpeza.

O protótipo de nosso pacote CSV usou a inicialização estática para definir os valores iniciais de ponteiros, contadores e outros. Mas essa opção é limitante, uma vez que ela evita a reinicialização das rotinas a seu estado inicial depois que uma das funções foi chamada. Uma alternativa é fornecer uma função de inicialização que define todos os valores internos com os valores iniciais corretos. Isso permite reinicializar, mas depende do usuário para chamar a inicialização explicitamente. A função `reset` da segunda versão poderia ser tornada pública para esse fim.

Em C++ e Java, os construtores são usados para inicializar os membros de dados das classes. Construtores definidos adequadamente garantem que todos os membros de dados sejam inicializados e que não haja como criar um objeto de classe não-inicializada. Um grupo de construtores pode suportar diversos tipos de inicializadores. Nós podemos fornecer `csv` com um construtor que tome um nome de arquivo e outro que tome um fluxo de entrada.

E quanto às cópias das informações gerenciadas por uma biblioteca, tal como as linhas de entrada e os campos? Nosso programa `csvgetline` em C fornece acesso direto às strings de entrada (linhas e campos), retornando ponteiros para elas. Esse acesso não-restrito tem diversas desvantagens. É possível para o usuário sobrepor memória de modo a tornar as outras informações inválidas. Por exemplo, uma expressão como

```
strcpy(csvfield(1), csvfield(2));
```

falharia de várias maneiras, principalmente sobrepondo o início do campo 2 se ele fosse mais longo do que o campo 1. O usuário da biblioteca deve fazer uma cópia de todas as informações a serem preservadas além da próxima chamada de `csvgetline`; na seqüência abaixo, o ponteiro seria inválido no final se a segunda `csvgetline` causasse uma realocação de seu buffer de linha.

```
char *p;

csvgetline(fin);
p = csvfield(1);
csvgetline(fin);
/* p poderia ser inválido aqui */
```

A versão em C++ é mais segura porque as strings são cópias que podem ser alteradas à vontade.

A Java usa referências para se referir aos objetos, ou seja, toda entidade diferente dos tipos básicos como `int`. Isso é mais eficiente do que fazer uma cópia, mas é possível se enganar e achar que uma referência é uma cópia; tivemos um bug desse tipo numa primeira versão de nosso programa `markov` em Java, e essa questão é fonte constante de bugs que envolvem strings em C. Os métodos clones fornecem uma maneira de fazer uma cópia quando for necessário.

O outro lado da inicialização de nossa construção é a finalização ou destruição – limpar e recuperar recursos quando alguma entidade não é mais necessária. Isso

é particularmente importante para a memória, uma vez que um programa que falha na recuperação de memória não utilizada eventualmente não será executado. O software mais moderno tem uma embaraçosa tendência a apresentar essa falha. Problemas relacionados ocorrem quando os arquivos abertos devem ser fechados: se os dados estão em buffer, este pode ter de ser esvaziado (e sua memória ter de ser retomada). Para as funções de biblioteca da C padrão, o "esvaziamento" acontece automaticamente quando o programa é encerrado normalmente, mas de outra forma ele deve ser programado. A função padrão da C e C++ `atexit` fornece uma maneira de obter o controle imediatamente antes do encerramento normal de um programa; os implementadores de interface podem usar esse recurso para programar a limpeza.

Liberar um recurso na mesma camada em que foi alocado. Uma maneira de controlar a alocação e retomada de recursos é fazer com que a mesma biblioteca, pacote, ou interface que aloca um recurso seja responsável pela sua liberação. Outra maneira de dizer isso é que o estado de alocação de um recurso não deve ser alterado em toda a interface. Nossas bibliotecas CSV lêem dados de arquivos que já foram abertos, de modo que elas os deixam abertos quando terminam. Aquele que chama a biblioteca precisa fechar os arquivos.

Os construtores e desconstrutores da C++ ajudam a implantar essa regra. Quando uma instância de classe sai do escopo ou é explicitamente destruída, o destrutor é chamado. Ele pode esvaziar os buffers, recuperar memória, redefinir valores e fazer tudo o mais que for necessário. A Java não fornece um mecanismo equivalente. Embora seja possível definir um método de finalização para uma classe, não há garantia de que ele será executado, muito menos em determinada hora, portanto as ações de limpeza não podem ter ocorrência garantida, embora quase sempre seja razoável supor que elas ocorrerão.

A Java fornece ajuda considerável com o gerenciamento da memória porque ela tem a *coleta de lixo* incorporada. À medida que um programa é executado, ele aloca objetos novos. Não há como desalocá-los explicitamente, mas o sistema do tempo de execução controla quais objetos ainda estão em uso e quais não estão, e retorna periodicamente aqueles não-utilizados para o pool de memória disponível.

Há uma variedade de técnicas para se fazer a coleta de lixo. Alguns esquemas controlam o número de usos de cada objeto, sua *contagem de referência*, e liberam um objeto quando sua contagem de referência chega a zero. Essa técnica pode ser usada explicitamente em C e C++ para gerenciar os objetos compartilhados. Os outros algoritmos seguem periodicamente uma trilha a partir do pool de alocação para todos os objetos referenciados. Os objetos que são encontrados dessa forma ainda estão em uso; os objetos que não são referidos por outro objeto não estão em uso e podem ser retomados.

A existência da coleta automática de lixo *não* significa que não haja questões de gerenciamento da memória em um projeto. Ainda temos de determinar se as interfaces retornam referências para os objetos compartilhados ou cópias deles, e isso afeta todo o programa. A coleta de lixo não é grátis – há um custo para manter as informações e retomar a memória não-utilizada, e a coleta pode ocorrer em momentos imprevisíveis.

Todos esses problemas tornam-se mais complicados quando uma biblioteca deve ser usada em um ambiente onde mais de um encadeamento de controle pode estar executando suas rotinas ao mesmo tempo, como em um programa Java multiencadeado.

Para evitar problemas, é preciso escrever código *reentrante*, o que significa que ele funciona independente do número de execuções simultâneas. O código reentrante evita as variáveis globais, variáveis locais estáticas e todas as outras variáveis que poderiam ser modificadas enquanto outro encadeamento as estivesse usando. O segredo do bom projeto de multiencadeamento é separar os componentes de modo que eles compartilhem nada, exceto por meio de interfaces bem definidas. As bibliotecas que inadvertidamente expõem as variáveis ao compartilhamento destroem o modelo. (Em um programa multiencadeado, `strtok` é um desastre, assim como para as outras funções da biblioteca C que armazenam valores na memória estática interna.) Se as variáveis pudessem ser compartilhadas, elas deveriam estar protegidas por algum tipo de mecanismo de bloqueio para garantir que apenas uma thread de cada vez as acessasse. As classes são uma grande ajuda aqui porque elas fornecem um foco para discutir o compartilhamento e bloqueio de modelos. Os métodos sincronizados em Java fornecem uma maneira de um encadeamento bloquear toda uma classe ou instância de uma classe contra a modificação simultânea feita por algum outro encadeamento; os blocos sincronizados permitem que apenas um encadeamento de cada vez execute uma seção do código.

O multiencadeamento aumenta a complexidade das questões de programação, além de ser um tópico grande demais para ser discutido com detalhes aqui.

4.7 Abort, Retry, Fail?

Nos capítulos anteriores, usamos funções como `eprintf` e `estrdup` para lidar com os erros exibindo uma mensagem antes de encerrar a execução. Por exemplo, `eprintf` se comporta como `fprintf(stderr,...)`, mas sai do programa com um status de erro depois de reportá-lo. Ele usa o header `<stdarg.h>` e a rotina de biblioteca `vfprintf` para imprimir os argumentos representados pelo `...` do protótipo. A biblioteca `stdarg` deve ser inicializada por uma chamada para `va_start` e encerrada por `va_end`. Vamos usar mais dessa interface no Capítulo 9.

```
#include <stdarg.h>
#include <string.h>
#include <errno.h>

/* eprintf: imprime mensagem de erro e sai */
void eprintf(char *fmt, ...)
{
    va_list args;

    fflush(stdout);
    if (progname() != NULL)
```

```
        fprintf(stderr, "%s: ", progname());

    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);

    if (fmt[0] != '\0' && fmt[strlen(fmt)-1] == ':')
        fprintf(stderr, " %s", strerror(errno));
    fprintf(stderr, "\n");
    exit(2); /* valor convencional para execução falha */
}
```

Se o argumento de formato terminar com dois pontos, eprintf chama a função padrão da C strerror, a qual retorna uma string contendo todas as informações adicionais de erro do sistema que possam estar disponíveis. Também escrevemos weprintf, semelhante a eprintf, que exibe um aviso mas não sai. A interface do tipo printf é conveniente para construir strings que podem ser impressas ou exibidas em uma caixa de diálogo.

Da mesma forma, estrdup tenta fazer uma cópia de uma string e sai com uma mensagem (por meio de eprintf) se ela ficar sem memória:

```
/* estrdup: duplica uma string, reporta se houver erro */
char *estrdup(char *s)
{
    char *t;

    t = (char *) malloc(strlen(s)+1);
    if (t == NULL)
        eprintf("estrdup(\"%.20s\") failed:", s);
    strcpy(t, s);
    return t;
}
```

e emalloc fornece um serviço semelhante para as chamadas a malloc:

```
/* emalloc: malloc e reporta se houver erro */
void *emalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p == NULL)
        eprintf("malloc of %u bytes failed:", n);
    return p;
}
```

Um arquivo header coincidente chamado eprintf.h declara essas funções:

```
/* eprintf.h: funções error wrapper */
```

```
extern   void   eprintf(char *, ...);
extern   void   weprintf(char *, ...);
extern   char   *estrdup(char *);
extern   void   *emalloc(size_t);
extern   void   *erealloc(void *, size_t);
extern   char   *progname(void);
extern   void   setprogname(char *);
```

Esse header é incluído em todo arquivo que chama uma das funções de erro. Cada mensagem de erro também inclui o nome do programa se ele foi definido por quem chama; isso é definido e recuperado pelas funções comuns `setprogname` e `progname`, declaradas no arquivo header e definidas no arquivo-fonte com `eprintf`:

```
static char *name = NULL; /* nome do programa para as mensagens */

/* setprogname: define nome armazenado do programa */
void setprogname(char *str)
{
    name = estrdup(str);
}

/* progname: retorna nome armazenado do programa */
char *progname(void)
{
    return name;
}
```

O uso típico se parece com o seguinte:

```
int main(int argc, char *argv[])
{
    setprogname("markov");
    ...
    f = fopen(argv[i], "r");
    if (f == NULL)
        eprintf("can't open %s:", argv[i]);
    ...
}
```

que imprime saída assim:

```
markov: can't open psalm.txt: No such file or directory
```

Achamos que essas funções de wrapper são convenientes para nosso uso próprio, uma vez que elas unificam o tratamento de erros, e sua própria existência nos encoraja a pegar os erros, em vez de ignorá-los. Não há nada de especial em nosso projeto, porém, e você pode preferir alguma variante para seus próprios programas.

Suponhamos que, em vez de escrever funções para nosso uso próprio, nós estejamos criando uma biblioteca para outras pessoas usarem em seus programas. O que uma função dessa biblioteca deveria fazer se ocorresse um erro irrecuperável? As funções que escrevemos anteriormente neste capítulo exibem uma mensagem e morrem. Esse é um comportamento aceito em muitos programas, particularmente nas ferramentas isoladas pequenas e nos aplicativos. Nos outros programas, porém, sair é errado, uma vez que isso evita que o restante do programa tente alguma recuperação. Por exemplo, um processador de texto deve se recuperar dos erros para não perder o documento que você está digitando. Em algumas situações uma rotina de biblioteca não deve nem exibir uma mensagem, uma vez que o programa pode estar sendo executado em um ambiente no qual uma mensagem vai interferir com os dados exibidos ou desaparecerá sem deixar vestígios. Uma alternativa útil é registrar a saída de diagnóstico em um "arquivo de registro" explícito, onde ela pode ser monitorada de forma independente.

Detectar erros em um nível baixo, lidar com eles em um nível alto. Como princípio geral, os erros devem ser detectados no nível mais baixo possível, mas tratados em um nível alto. Na maioria dos casos, quem chama deve determinar como lidar com os erros, não quem é chamado. As rotinas de biblioteca podem ajudar nisso através da falha. Esse raciocínio nos levou a retornar `NULL` para um campo não-existente em vez de abortar. Do mesmo modo, `csvgetline` retorna `NULL` independente do número de vezes que ela é chamada após o primeiro final de arquivo.

Os valores de retorno apropriados nem sempre são tão óbvios, como vimos na discussão anterior sobre o que `csvgetline` deveria retornar. Queremos retornar o máximo possível de informações úteis, mas em uma forma cujo uso seja fácil para o restante do programa. Em C, C++ e Java, isso significa retornar alguma coisa como o valor da função, e talvez outros valores por meio dos argumentos de referência (ponteiros). Muitas funções de biblioteca dependem da habilidade de distinguir os valores normais dos valores de erro. As funções de entrada `getchar` retornam uma `char` para os dados válidos, e algum valor não `char` como `EOF` para o final de arquivo ou erro.

Esse mecanismo não funciona se os valores de retorno legais da função ocuparem todos os valores possíveis. Por exemplo, uma função matemática como `log` pode retornar qualquer número de ponto flutuante. No ponto flutuante IEEE, um valor especial chamado `NaN` ("not a number") indica um erro e pode ser retornado como um sinal de erro.

Algumas linguagens, tais como a Perl e Tcl, fornecem uma maneira barata de agrupar dois ou mais valores em um `tuple`. Em tais linguagens, um valor de função e qualquer estado de erro podem ser facilmente retornados juntos. A C++ STL fornece um tipo de dados `pair` que também pode ser usado dessa forma.

Se possível, é desejável distinguir os diversos valores excepcionais como final de arquivo e estados de erro, em vez de colocá-los juntos em um único valor. Se os valores não puderem ser separados facilmente, outra opção seria retornar um valor único de "exceção" e fornecer outra função que retornasse mais detalhes sobre o último erro.

Essa é a abordagem usada pelo Unix e na biblioteca padrão C, onde muitas chamadas de sistema e funções de biblioteca retornam -1 mas também definem uma variável global chamada errno que codifica o erro específico; strerror retorna uma string associada ao número do erro. Em nosso sistema, este programa:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <math.h>

/* errno main: testa errno */
int main(void)
{
    double f;
    errno = 0; /* limpa estado de erro */
    f = log(-1.23);
    printf("%f %d %s\n", f, errno, strerror(errno));
    return 0;
}
```

imprime:

```
nan0x10000000 33 Domain error
```

Como mostramos, errno deve ser declarado primeiro. Depois se ocorrer um erro, errno será definido com um valor diferente de zero.

Usar exceções somente nas situações excepcionais. Algumas linguagens fornecem *exceções* para pegar situações incomuns e fazer a recuperação. Elas fornecem um fluxo alternativo de controle quando algo de ruim acontece. As exceções não devem ser usadas para lidar com valores de retorno esperados. A leitura de um arquivo eventualmente produzirá um final de arquivo; isso deve ser tratado com um valor de retorno e não por uma exceção.

Em Java, escrevemos:

```
String fname = "someFileName";
try {
    FileInputStream in = new FileInputStream(fname);
    int c;
    while ((c = in.read()) != -1)
        System.out.print((char) c);
    in.close();
} catch (FileNotFoundException e) {
    System.err.println(fname + " not found");
} catch (IOException e) {
    System.err.println("IOException: " + e);
    e.printStackTrace();
}
```

O loop lê os caracteres até o final do arquivo, um evento esperado que é sinalizado por um valor de retorno de -1 de read. Se o arquivo não puder ser aberto, ele levanta uma exceção, em vez de definir o fluxo de entrada como NULL, como seria feito na C ou C++. Finalmente, se algum outro erro de E/S ocorrer no bloco try, ele também é excepcional, e é pego pela cláusula IOException.

As exceções quase sempre são superutilizadas. Como elas distorcem o fluxo do controle, elas podem levar a construções complicadas que estão propensas a bugs. É excepcional a abertura de um arquivo falhar; a geração de uma exceção neste caso nos parece excesso de engenharia. As exceções são melhor reservadas para os eventos verdadeiramente inesperados, tais como o enchimento total dos sistemas de arquivos ou os erros de ponto flutuante.

Nos programas em C, o par de funções setjmp e longjmp fornece um serviço de nível bem mais inferior sobre o qual um mecanismo de exceção pode ser construído, mas ele é suficientemente antigo para não termos de falar dele.

E a recuperação dos recursos quando ocorre um erro? Uma biblioteca deve tentar uma recuperação quando algo sai errado? Nem sempre esse é o caso, mas isso seria bom pois teríamos certeza de que ela deixaria no estado mais limpo e inalterado possível. Certamente o armazenamento não utilizado seria retomado. Se as variáveis ainda pudessem ser acessadas, elas deveriam ser definidas em valores sensitivos. Uma fonte comum de bugs é tentar usar um ponteiro que aponta para armazenamento liberado. Se o código de tratamento de erros definir ponteiros em zero depois de liberar aquilo para o qual eles apontam, isso não passará desapercebido. A função reset da segunda versão da biblioteca CSV foi uma tentativa de abordar essas questões. Em geral, elas visam manter a biblioteca utilizável após a ocorrência de um erro.

4.8 Interfaces de usuário

Até aqui falamos principalmente sobre as interfaces entre os componentes de um programa ou entre os programas. Mas há outro tipo importante de interface, aquela entre um programa e seus usuários humanos.

A maioria dos programas de exemplo deste livro se baseia no texto; portanto, suas interfaces de usuário tendem a ser diretas. Como discutimos na seção anterior, os erros deveriam ser detectados e reportados, e a recuperação tentada onde isso fizer sentido. A saída de erros deve incluir todas as informações disponíveis e deve ser o mais significativa possível tendo em vista o contexto. Um diagnóstico não deve dizer:

```
estrdup failed
```

quando ele poderia dizer

```
markov: estrdup("Derrida") failed: Memory limit reached
```

Não custa nada adicionar as informações extras como fizemos em estrdup, e isso pode ajudar o usuário a identificar um problema ou fornecer entrada válida.

Os programas devem exibir informações sobre o uso adequado quando um erro for cometido, como mostram funções como:

```
/* usage: imprime mensagem de uso e sai */
void usage(void)
{
    fprintf(stderr, "usage: %s [-d] [-n nwords]"
        " [-s seed] [files ...]\n", progname());
    exit(2);
}
```

O nome do programa identifica a fonte da mensagem, o que é particularmente importante se essa ela for parte de um processo maior. Se um programa apresentar uma mensagem que diz apenas `syntax error` ou `estrdup failed`, o usuário pode não ter a menor idéia de quem disse isso.

O texto das mensagens de erro, avisos e caixas de diálogo deve declarar a forma da entrada válida. Não diga que um parâmetro é grande demais; reporte o intervalo válido dos valores. Sempre que possível, o texto deve ser uma entrada válida por si mesmo, tal como a linha de comando completa com o parâmetro definido adequadamente. Além de direcionar os usuários quanto ao uso adequado, tal saída pode ser capturada em um arquivo ou por um movimento do mouse e depois usada para executar algum outro processo. Isso indica uma fraqueza das caixas de diálogo: seu conteúdo é difícil de entender para uso futuro.

Uma maneira efetiva de criar uma boa interface de usuário para entrada é criar uma linguagem especializada para definir parâmetros, controlar ações e assim por diante. Uma boa notação pode tornar um programa fácil de usar, e ajudar a organizar uma implementação. As interfaces baseadas na linguagem são o assunto do Capítulo 9.

A programação defensiva, ou seja, ter certeza de que um programa não está vulnerável à entrada ruim, é importante tanto para proteger os usuários contra si mesmos e também como um mecanismo de segurança. Isso é discutido com mais detalhes no Capítulo 6, o qual fala sobre os testes de programas.

Para a maioria das pessoas, as interfaces gráficas são *a* interface de usuário para seus computadores. As interfaces gráficas de usuário são um tópico enorme, portanto vamos dizer apenas algumas coisas relativas a este livro. Primeiro, as interfaces gráficas são difíceis de criar e ficarem "certas", uma vez que sua adequação e sucesso dependem muito do comportamento e das expectativas humanas. Em segundo lugar, e em termos práticos, quando um sistema tem uma interface de usuário, geralmente existe mais código para lidar com a interação do usuário do que nos algoritmos que fazem o trabalho.

No entanto, princípios conhecidos se aplicam tanto ao projeto externo quanto à implementação interna do software da interface gráfica. Do ponto de vista do usuário, as questões de estilo como simplicidade, clareza, regularidade, unifor-

midade, familiaridade e restrição todas contribuem para uma interface que é fácil de usar. A falta de tais propriedades geralmente acompanha as interfaces desagradáveis ou esquisitas.

A uniformidade e regularidade são desejáveis, incluindo o uso consistente de termos, unidades, formatos, layouts, fontes, cores, tamanhos e todas as outras opções que um sistema grafico disponibiliza. Quantas palavras diferentes são usadas para sair de um programa ou fechar uma janela? As opções variam de Abandonar a control-Z, com pelo menos uma dúzia delas no meio. Essa falta de consistência é confusa para um nativo da língua e frustrante para os outros.

Dentro do código de gráficos, as interfaces são particularmente importantes, uma vez que esses sistemas são grandes, complicados e orientados por um modelo de entrada diferente da digitalização de texto seqüencial. A programação orientada para o objeto excede nas interfaces gráficas de usuário, uma vez que fornece uma maneira de encapsular todo o estado e os comportamentos das janelas, usando a herança para combinar as similaridades nas classes base, separando as diferenças nas classes derivadas.

Leitura suplementar

Embora poucos de seus detalhes técnicos estejam atualizados, *The Mythical Man Month*, de Frederick P. Brooks, Jr. (Addison-Wesley, 1975; edição de aniversário 1995), é uma leitura deliciosa e contém idéias sobre o desenvolvimento de software que são tão valiosas hoje quanto na época da publicação original.

Quase todos os livros sobre programação têm alguma coisa útil a dizer sobre o projeto de interface. Um livro prático baseado em experiência conseguida a duras penas é o *Large-Scale C++ Software Design*, de John Lakos (Addison-Wesley, 1996), o qual discute como construir e gerenciar programas verdadeiramente grandes em C++. O livro *C Interfaces and Implementations*, de David Hanson (Addison-Wesley, 1997), é um bom tratamento para os programas em C.

O livro *Rapid Development* (Microsoft Press, 1996), de Steve McConnell, é uma descrição excelente de como construir software em equipes, com ênfase para o papel da "prototipação".

Existem vários livros interessantes sobre o projeto das interfaces gráficas de usuário, com uma variedade de perspectivas diferentes. Sugerimos os livros *Designing Visual Interfaces: Communication Oriented Techniques*, de Kevin Mullet e Darrell Sano (Prentice Hall, 1995), *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, de Ben Shneiderman (3ª edição, Addison-Wesley, 1997), *About Face: The Essentials of User Interface Design*, de Alan Cooper (IDG, 1995) e *User Interface Design*, de Harold Thimbleby (Addison-Wesley, 1990).