Capítulo 3

Estruturas de dados em C++

Num ambiente de pressão e tempo restrito como a Maratona de Programação, a Standard Template Library (STL) de C++ é essencialmente o que se deseja. Primeiro, porque o reuso de estruturas de dados complexas como sets, multisets e maps é indispensável. Segundo, porque você precisa simplesmente confiar na eficiência de uma estrutura de dados bem construída do que fazê-la em um tempo tão curto quanto o da competição.

Este capítulo dá várias dicas e exemplifica o uso maciço da STL C++. O maratonista que domina a STL tem sua produtividade muito ampliada, por isso é tão importante esse estudo prévio. Para um maior aproveitamento da leitura deste capítulo é necessário possuir conhecimento prévio sobre as estruturas de dados apresentadas.

3.1 Pilhas

A pilha é uma das estruturas de dados mais simples: ela representa um container de dados onde se coloca qualquer quantidade de valores e sempre se retira o último valor que foi colocado (o mais recente). Esse regime é chamado LIFO (Last-In, First-Out).

Uma aplicação direta para pilhas é inverter elementos de uma seqüência (como um vetor).

Exemplo 3.1.1: Receba como entrada um número N seguido de N inteiros. Imprima esses inteiros na ordem inversa que eles foram informados na entrada.

```
1:  #include <stack>
2:  #include <iostream>
3:  using namespace std;
4:
5:  int main() {
6:     long N, num;
7:     stack<long> s;
8:     cin >> N;
9:     while(N--) {
10:       cin >> num;
11:       s.push(num);
12:    }
13:    while(!s.empty()) {
14:       cout << s.top() << endl;
15:       s.pop();
16:    }
17: }
```

Código 3.1.1: Invertendo inteiros lidos da entrada.

A linha 1 inclui a biblioteca de pilhas da STL. A linha 2 inclui a biblioteca de entrada e saída. A linha 3 torna visível para seu fonte o namespace padrão. Há três primitivas básicas de uma pilha: o push (linha 11), que insere novos valores na pilha; o pop (linha 15), que remove o valor mais recente inserido na pilha; e o empty (linha 13), que verifica se há algum valor na pilha. Para acessar o elemento mais recente na pilha, utiliza-se a primitiva top (linha 14).

O uso de pilhas por vezes torna certas implementações bem elegantes.

Exemplo 3.1.2: "Shellsort", questão 10152 do UVa (http://online-judge.uva.es/p/v101/10152.html). É dada uma lista de strings inicial e a lista de strings como deve ser ordenada. Dado que a única operação permitida para se ordenar a lista é mover uma string para o início, deseja-se saber qual a seqüência de operações de menor tamanho que deve ser feita para ordenar a lista inicial.

Exercício 3.1.1: Releia o enunciado da questão. Depois de ter certeza que entendeu o enunciado, marque 10 minutos. Durante esse tempo, tente elaborar um algoritmo para resolver a questão (não codifique nada, apenas idealize um algoritmo). Só continue lendo depois de fazê-lo.

Há um algoritmo elegante e simples para resolver esta questão. Suponha que se percorram as duas listas de trás para frente. Ao percorrê-las, se os elementos da lista inicial e da ordenada forem os mesmos, não é necessário realizar movimento algum. Se forem encontrados elementos diferentes quando percorrendo as duas listas, significa que aquele elemento da lista inicial precisa ser enviado para o topo. Dessa forma, são identificados todos os elementos da lista inicial que precisam ser movidos para o topo.

O que não se sabe a princípio é qual a melhor ordem para enviar os elementos que precisam ser movidos para o topo. Acontece que a própria lista ordenada indica isso: entre todos os elementos que precisam ser movidos, o elemento que aparece por último na lista deve ser enviado para o topo primeiro (para que os demais sejam ordenados sobre ele). Isso prossegue recursivamente até chegar à string no topo da pilha. Esse algoritmo é apresentado a seguir.

Para percorrer as duas listas de strings de trás para frente, uma idéia interessante é usar pilhas para representar as duas listas, "initial" e "ordered" (linha 15). Nas linhas 8 e 9 é lida a primeira linha da entrada com o número de casos de teste. O laço das linhas 11 a 37 garante que se repita o algoritmo para todos os casos de teste. Cada caso de teste começa por um inteiro **n** em uma linha isolada indicado o tamanho de cada lista de strings (o que é lido da entrada nas linhas 12 e 13). Seguem então **n** strings da lista inicial que são armazenadas na pilha "initial" (linhas 16 a 19) e mais **n** strings da lista ordenada que são armazenadas na pilha "ordered" (linhas 20 a 23).

Nas linhas 25 a 30 está a idéia central do algoritmo. O laço percorre a lista inicial, removendo um elemento da lista ordenada sempre o topo da lista inicial coincidir com o topo da lista ordenada (linhas 26 a 28). Após esse laço, a pilha ordenada vai possuir apenas os elementos que precisam ser movidos. Como essa pilha já representa a ordem final dos elementos, significa que a ordem em que os elementos devem ser movidos para o topo da pilha é a mesma ordem da pilha ordenada. Isso é explorado nas linhas 32 a 36 para imprimir como número mínimo de movimentos o que restar na pilha ordenada.

Nesse exemplo são usadas pilhas para tornar mais elegante a representação e percurso das listas de strings. Há várias outras aplicações interessantes para pilhas. Uma delas é a avaliação de expressões e do uso correto de parênteses.

```
1:    #include <iostream>
2:    #include <string>
3:    #include <stack>
4:    using namespace std;
5:    int main() {
6:       string name;
7:       int cases, n, i;
8:       getline(cin, name);
9:       cases = atoi(name.c_str());
10:
11:      while(cases--) {
12:         getline(cin, name);
13:         n = atoi(name.c_str());
14:
15:         stack<string> initial, ordered;
16:         for(i=0; i<n; i++) {
17:            getline(cin, name);
18:            initial.push(name);
19:         }
20:         for(i=0; i<n; i++) {
21:            getline(cin, name);
22:            ordered.push(name);
23:         }
24:
25:         while(!initial.empty()) {
26:            if( initial.top() == ordered.top() ) {
27:               ordered.pop();
28:            }
29:            initial.pop();
30:         }
31:
32:         while(!ordered.empty()) {
33:            cout << ordered.top() << endl;
34:            ordered.pop();
35:         }
36:         cout << endl;
37:      }
38:   }
```

Código 3.1.2: Shellsort

Exemplo 3.1.3: “Parentheses Balance”, questão 673 da Uva (http://online-judge.uva.es/p/v6/673.html). Dada uma string composta por uma seqüência dos quatro caracteres “()[]” (aspas para tornar mais claro), o objetivo é informar se os parênteses e colchetes estão empregados corretamente (todos os colchetes e parênteses abertos são fechados pelo respectivo caractere).

A solução proposta para este problema é simples. A idéia é percorrer a expressão dada como entrada da esquerda para a direita e empilhar o caractere se ele for '(' ou '['. Se o caractere encontrado for ')' ou ']', basta verificar se o caractere no topo da pilha corresponde ao respectivo caractere.


```
39:   #include <cstdio>
40:   #include <stack>
41:   using namespace std;
42:
43:   bool balance(char *str) {
44:      int i;
45:      stack<char> s;
46:      for(i=0; str[i]!='\0'; i++) {
47:         if(str[i]=='(' || str[i]=='[') {
48:            s.push( str[i] );
49:         }
50:         else if(str[i]==')') {
51:            if(s.empty() || s.top()!='(') return false;
52:            s.pop();
53:         }
54:         else if(str[i]==']') {
55:            if(s.empty() || s.top()!='[') return false;
56:            s.pop();
57:         }
58:      }
59:      return s.empty();
60:   }
61:
62:   int main() {
63:      int cases;
64:      char s[129];
65:      gets(s);
66:      cases = atoi(s);
67:      while(cases--) {
68:         gets(s);
69:         if(balance(s)) printf("Yes\n");
70:         else printf("No\n");
71:      }
72:   }
```

Código 3.1.3: Parentheses Balance

Exercício 3.1.2: A linha 21 garante que a função *balance* só retorna verdadeiro se a pilha estiver vazia depois de percorrida toda a expressão. Se esta linha fosse alterada para "**return true;**", haveria alguns casos em que o programa informaria um falso verdadeiro. Dê um exemplo de expressão errada que este código a indicaria como correta se fosse feita esta alteração.

Exercício 3.1.3: Utilize pilhas para implementar "Camel Trading", a questão 10700 do UVa (http://online-judge.uva.es/p/v107/10700.html). O objetivo dessa questão é informar qual o menor e o maior valor possível para uma expressão modificando-se a ordem de avaliação dos operadores da mesma. Dica: o que acontece se todas as somas forem feitas antes de todas as multiplicações? E se todas as multiplicações forem feitas antes de todas as somas?

3.2 Filas

Filas são containers de dados que seguem o regime LIFO (Last-In, First-Out), ou seja, o próximo valor removido da fila é sempre aquele que está armazenado há mais tempo.

Exemplo 3.2.1: The Dole Queue, questão 133 do UVa (http://online-judge.uva.es/p/v1/133.html). Dada uma fila circular com N elementos numerados de 1 a N e dois valores inteiros K e M, o objetivo é simular o seguinte algoritmo. Um juiz inicia no começo da fila e caminha K posições na ordem crescente dos números até chegar ao K-ésimo elemento. Outro juiz inicia no final da fila e caminha M posições na ordem decrescente dos números até chegar ao M-ésimo elemento. O K-ésimo e o M-ésimo elementos devem ser removidos da fila. O procedimento continua até a fila esvaziar.

```
1:  #include <cstdio>
2:  #include <queue>
3:  #include <iostream>
4:  using namespace std;
5:
6:  queue<int> removeX(queue<int> q, int x) {
7:     queue<int> ret;
8:     while(!q.empty()) {
9:        if(q.front()!=x) ret.push( q.front() );
10:       q.pop();
11:    }
12:    return ret;
13: }
14:
15: int main() {
16:     int N, k, m, i, d1, d2;
17:     while(true) {
18:        scanf("%d%d%d", &N, &k, &m);
19:        if(!N&&!k&&!m) return 0;
20:        queue<int> qk, qm;
21:        for(i=1; i<=N; i++) qk.push(i);
22:        for(i=N; i>=1; i--) qm.push(i);
23:        while(!qk.empty()) {
24:           for(i=0; i<k; i++) {
25:              d1 = qk.front();
26:              qk.pop();
27:              if(i!=k-1) qk.push(d1);
28:           }
29:           for(i=0; i<m; i++) {
30:              d2 = qm.front();
31:              qm.pop();
32:              if(i!=m-1) qm.push(d2);
33:           }
34:           if(d1==d2) {
35:              printf("%3d", d1);
36:              if(!qk.empty()) putchar(',');
37:           }
38:           else {
39:              qk = removeX(qk, d2);
40:              qm = removeX(qm, d1);
41:              printf("%3d%3d", d1, d2);
42:              if(!qk.empty()) putchar(',');
43:           }
44:        }
45:     }
46: }
```

Código 3.2.1: The Dole Queue.

A função removeX nas linhas 6 a 13 serve para remover todas as ocorrências de um valor x de uma fila q, retornando a fila resultante. Isso é útil para remover da fila o K-ésimo ou M-ésimo elemento da fila.

Para esta questão são usadas as variáveis N, k e m lidas da entrada, assim como a variável contadora i e as variáveis d1 e d2 que indicam respectivamente o k-ésimo e o m-ésimo elemento da fila (linha 16). O laço principal que inicia na linha 17 lê os três inteiros da entrada (linha 18) e termina quando os três forem informados iguais a zero (linha 19).

A solução faz uso de duas filas de inteiros, chamadas qk e qm na linha 20. Elas são iniciadas nas linhas 21 e 22 respectivamente com os números de 1 a N em ordem crescente e decrescente. As linhas 23 a 44 apresentam o laço para cada caso de teste. Nas linhas 24 a 28 é determinado o k-ésimo elemento da fila, que é armazenado na variável d1. Nas linhas 29 a 33 é determinado o m-ésimo elemento da fila, que é armazenado na variável d2. Tanto o elemento d1 quanto o elemento d2 não são devolvidos de volta para as respectivas filas (linhas 27 e 32).

Nas linhas 34 a 43 é exibida a saída. Se o k-ésimo e o m-ésimo elemento forem iguais, esse elemento é apenas exibido sem a necessidade de removê-lo das filas (já que ele não foi devolvido para a mesma). Caso eles sejam diferentes, d2 é removido da fila qk (linha 39) e d1 é removido da fila qm (linha 40). Depois disso, os elementos d1 e d2 são impressos. As linhas 36 e 42 imprimem a vírgula exigida pelo enunciado após o par de números impressos.

Filas são muito utilizadas para simulação e tornam algumas implementações bem mais sucintas. Elas também são usadas para criarem algoritmos que constroem soluções em profundidade.

Exemplo 3.2.2: Knight Moves, questão 439 do UVa (http://online-judge.uva.es/p/v4/439.html). Dada a posição inicial e final de um cavalo em um tabuleiro 8x8 de xadrez cujas linhas são indicadas de “a” a “h” e colunas numeradas de 1 a 8, o objetivo é indicar qual o menor número possível de lances de cavalo para se mover da posição inicial até a posição final. Para quem não joga xadrez, o cavalo anda em forma de L, duas colunas e uma linha adiante ou ainda uma coluna e duas linhas adiante.

O algoritmo usado como solução é uma busca simples em profundidade. A idéia é usar uma fila, inicialmente contendo apenas a posição inicial do cavalo. Faz-se um laço removendo o elemento da frente da fila e tentando as oito possíveis casas que o cavalo pode se movimentar, incrementando em um o número de movimentos para chegar àquela casa e incluindo de volta essa posição ao final da fila. A primeira vez que se retirar da fila a posição final, trata-se do número mínimo de movimentos para chegar àquela casa.

Exercício 3.2.1: Por que o regime da fila garante que a primeira vez que for removida da fila a posição final, isso será feito com o menor número possível de movimentos?

Nas linhas 1 a 3 são incluídas as bibliotecas de entrada/saída e fila. Nas linhas 5 a 7 é definida uma estrutura “position” representando uma posição com três inteiros indicando linha, coluna e número de movimentos. As strings initial e final guardam a entrada (linha 10). Nas linhas 16 a 20 é colocada a posição inicial do cavalo na fila. Nas linhas 22 a 41 é feito o laço principal do algoritmo.

O elemento da frente da fila é retirado nas linhas 23 e 24. A primitiva front retorna o elemento da frente da fila e a primitiva pop retira o elemento da frente da fila. As linhas 26 a 31 testam se o elemento retirado da fila equivale à posição final lida da entrada. Caso seja a posição final, é exibida a respectiva saída e o laço é encerrado. Nas linhas 33 a 40 são calculadas as oito casas em que o cavalo pode se movimentar a partir da casa na frente da fila. O vetor de inteiros "neighs" é um artifício para reduzir o tamanho do código ao calcular as oito movimentações possíveis. Quando o movimento conduzir a uma casa válida, ele é devolvido para o final da fila.

```
1:    #include <iostream>
2:    #include <queue>
3:    using namespace std;
4:
5:    typedef struct position {
6:       int lin, col, moves;
7:    };
8:
9:    int main() {
10:       string initial, final;
11:       int i, j, neighs[8][2] =
12:       {{-2,-1},{-2,1},{-1,-2},{-1,2},{1,-2},{1,2},{2,-1},{2,1}};
13:       position p, n;
14:
15:       while(cin >> initial >> final) {
16:          p.lin = initial[0]-'a';
17:          p.col = initial[1]-'1';
18:          p.moves = 0;
19:          queue<position> q;
20:          q.push(p);
21:
22:          while(!q.empty()) {
23:             p = q.front();
24:             q.pop();
25:
26:             if(p.lin==final[0]-'a' && p.col==final[1]-'1') {
27:                cout << "To get from " << initial << " to "
28:                     << final << " takes " << p.moves
29:                     << " knight moves." << endl;
30:                break;
31:             }
32:
33:             n.moves = p.moves+1;
34:             for(i=0; i<8; i++) {
35:                n.lin = p.lin + neighs[i][0];
36:                n.col = p.col + neighs[i][1];
37:                if(n.lin>=0 && n.lin<8 && n.col>=0 && n.col<8) {
38:                   q.push(n);
39:                }
40:             }
41:          }
42:       }
43:    }
```

Código 3.2.1: Knight Moves.

Exercício 3.2.2: É possível tornar esse algoritmo bem mais eficiente evitando que se insiram na fila posições do tabuleiro que já tenham sido visitadas. A idéia é usar uma matriz booleana 8x8, digamos "visited", inicialmente com todas as posições marcadas com falso. Só são inseridas novas posições na fila depois que se verificar se aquela posição ainda não foi visitada.

Exercício 3.2.3: Australian Voting, questão 10142 do UVa (http://online-judge.uva.es/p/v101/10142.html). O objetivo é indicar qua(is) o(s) vencedor(es) de uma eleição em que participam **n** candidatos e **m** eleitores. Cada eleitor vota em n candidatos, na ordem de sua preferência. Se um candidato obtiver mais do que a metade dos votos dos eleitores como primeira preferência, ele é eleito. Caso contrário, os votos dos candidatos empatados em último lugar devem ser ignorados e o voto daquele eleitor específico deve passar para a próxima preferência. O procedimento prossegue até que um candidato tenha mais de 50% dos votos ou até que haja um empate entre todos os candidatos ainda não eliminados (caso em que todos eles devem ser impressos como vencedores). Os exemplos ajudam no entendimento e solução dessa questão.

Exemplo de entrada expandido:	**Respectiva Saída:**
2 4 AAA BBB CCC DDD 1 2 3 4 1 3 2 4 4 1 2 3 2 1 3 4 2 3 1 4 3 3 3 3 2 AAA BBB 1 2 2 1	AAA AAA BBB

3.3 Conjuntos

Conjuntos são muito úteis em várias situações. A template class "set" oferece todas as funcionalidades desejadas com conjuntos.

Exemplo 3.3.1: Receba como entrada um número N (menor que 200000) e N inteiros variando de 1 a 2147483648 e informe quantos números diferentes há entre esses N elementos.

Nas linhas 1 e 2 estão sendo incluídas as bibliotecas set (para conjuntos) e iostream (para entrada/saída). A linha 3 torna visível para seu fonte o namespace padrão. As variáveis N e K são declaradas na linha 5, assim como o conjunto s de elementos do tipo unsigned long long int na linha 6. Note o poder dos templates: a mesma estrutura de dados *set* pode

ser aplicada a qualquer tipo (inclusive outros templates) sem mudar nada em sua interface. O número de elementos N é lido da entrada na linha 7.

O laço das linhas 8 a 11 é repetido N vezes. Como isso funciona? Em C/C++, uma expressão inteira que resulta em 0 (zero) é tratada como falso e qualquer valor diferente é verdadeiro. Sendo assim, quando N zerar, o laço terá sido executado exatamente N vezes (Importante: se N for menor do que zero você terá um laço infinito, o que nunca ocorre nas entradas de Maratonas). Note como é interessante essa construção "while(N--)" porque ela dispensa o uso de uma variável adicional para contagem (como em "for(i=0; i<N; i++)"). Use esse truque sempre que você não precisar mais do valor de N.

A linha 9 lê da entrada um inteiro e armazena em k. Na linha 10, esse valor é adicionado ao conjunto s. O que ocorre se já houver aquele valor no conjunto? Assim como em um conjunto matemático, esse valor não é armazenado duas vezes no conjunto. Isso é exatamente o comportamento que se deseja para esse problema (e muitos outros).

Como o enunciado deseja que se exiba o número de elementos diferentes, basta imprimir o tamanho do conjunto (linha 12).

```
1:    #include <set>
2:    #include <iostream>
3:    using namespace std;
4:    int main() {
5:       unsigned long long int N, k;
6:       set<unsigned long long int> s;
7:       cin >> N;
8:       while(N--) {
9:          cin >> k;
10:         s.insert(k);
11:      }
12:      cout << s.size() << endl;
13:   }
```

Código 3.3.1: Contando o número de elementos distintos da entrada

Exemplo 3.3.2: "Jolly Jumpers", questão 10038 da UVa (http://acm.uva.es/p/v100/10038.html). Dado um vetor com n elementos, o objetivo dessa questão é indicar se o valor absoluto da diferença entre números consecutivos no vetor possui todos os números de 1 a n-1.

Casos de teste: {1 4 2 3}. |1-4| = 3, |4-2| = 2, |2-3| = 1. Assim, temos todos os números de 1 a n-1 (1, 2 e 3) como diferença entre elementos consecutivos. No caso de teste {1 4 2 -1 6}, temos |1-4|=3, |4-2|=2, |2-(-1)|=3, |-1-6|=7, ou seja, não há todos os números de 1 a 4.

Não há novidade até a linha 6, salvo que está sendo incluída a biblioteca math.h para a função abs. Na linha 7 é lido o inteiro n. Se a entrada terminar (o arquivo de testes chegar no EOF), o laço vai parar de executar já que cin retornaria falso nesse caso.

A solução proposta não utilize um vetor. Ao invés disso, faz uso do fato que só interessa o elemento atual e o anterior para calcular as diferenças e os armazena nas variáveis ant e atual. Sabendo disso, por que é lido um elemento inicialmente na variável ant na linha 10? Por que o laço de leitura só executa n-1 vezes (inicia em i=1)? Por que a linha 15?

A lógica novamente é parecida com o exemplo anterior. Sempre que o valor absoluto da diferença entre dois elementos estiver no intervalo entre 1 e n-1, essa diferença é inserida

no conjunto s (linha 14). Ao final, se o número de elementos no conjunto for igual a n-1, então a saída deve ser “Jolly”, caso contrário deve ser “Not jolly” (linhas 17 e 18).

Note que é importante testar se a diferença está no intervalo exigido (linha 14). Você conseguiria criar um caso de teste em que o programa responderia com um falso Jolly se não houvesse o teste na linha 14 (e ela fosse simplesmente “s.insert(dif)”)?

```
1:    #include <set>
2:    #include <iostream>
3:    #include <math.h>
4:    using namespace std;
5:    int main() {
6:       int n;
7:       while( cin >> n ) {
8:          int ant, atual, dif;
9:          set<int> s;
10:         cin >> ant;
11:         for(int i=1; i<n; i++) {
12:            cin >> atual;
13:            dif = abs(atual-ant);
14:            if(dif>=1 && dif<=n-1) s.insert(dif);
15:            ant = atual;
16:         }
17:         if(s.size()==n-1) cout << "Jolly" << endl;
18:         else cout << "Not jolly" << endl;
19:      }
20:   }
```

Código 3.3.2: Jolly Jumpers.

Percorrendo os elementos de um conjunto

A STL permite que você itere (percorra) os elementos contidos em qualquer uma de suas estruturas de dados. Isso é feito de uma forma padrão.

Exemplo 3.3.3: “Andy's First Dictionary”, questão 10815 da UVa (http://acm.uva.es/p/v108/10815.html). Dado um texto qualquer, você deve identificar palavras nesse texto (definidas como seqüências de letras) e exibir em ordem alfabéticas todas as palavras encontradas (sem repetição) e em minúsculo.

Das linhas 15 a 30 é feito um laço que lê a entrada linha a linha até que se alcance o fim da entrada. A cada linha lida (armazenada na variável line), é feito um laço (linha 16) nos caracteres dessa linha. Cada caractere é inicialmente convertido para minúsculo e armazenado na variável c (linha 17). Quando c for uma letra, ele vai compondo a string str (linhas 18 a 20) enquanto quando ele não for uma letra e a variável str não estiver vazia, essa string é inserida no conjunto s (linha 22) e esvaziada (linha 23). É preciso ter cuidado em colocar no conjunto s inclusive uma string que possa ter sido montada com letras até o último caractere da linha (linhas 26 a 29).

Se o enunciado não pedisse a saída dada em ordem alfabética, bastaria exibir todos os elementos do conjunto s. Como não é o caso, os elementos do conjunto s devem ser copiados para um vetor (linhas 35 a 37) e então esse vetor é ordenado (linha 38). Enfim, depois do vetor estar organizado alfabeticamente, deve ser exibido (linhas 41 a 44).

Atenção especial deve ser dada às linhas 34 e 35 e às linhas 41 e 42. Essa é a forma de iterar sobre estruturas de dados da STL. Note que o conjunto é uma estrutura de acesso aleatório enquanto o vetor é seqüencial, no entanto ambas as estruturas são percorridas da mesma forma, o que é bastante desejável para facilitar o aprendizado da STL.

```
1:    #include <iostream>
2:    #include <string>
3:    #include <vector>
4:    #include <set>
5:    #include <ctype.h> //para tolower e alpha
6:    #include <algorithm> //para sort
7:    using namespace std;
8:
9:    int main() {
10:      char c;
11:      string str, line;
12:      set<string> s;
13:
14:      //lendo o texto linha a linha e armazenando em s
15:      while( getline(cin, line) ) {
16:         for(unsigned int i=0; i<line.length(); i++) {
17:            c = tolower(line[i]);
18:            if(isalpha(c)) {
19:               str += c;
20:            }
21:            else if(str!="") {
22:               s.insert(str);
23:               str = "";
24:            }
25:         }
26:         if(str!="") {
27:            s.insert(str);
28:            str = "";
29:         }
30:      }
31:
32:      //copiando os elementos do conjunto para um vetor
33:      vector<string> v;
34:      set<string>::iterator it;
35:      for(it=s.begin(); it!=s.end(); it++) {
36:         v.push_back(*it);
37:      }
38:      sort(v.begin(), v.end());
39:
40:      //exibindo os elementos do vetor
41:      vector<string>::iterator i;
42:      for(i=v.begin(); i!=v.end(); i++) {
43:         cout << *i << endl;
44:      }
45:   }
```

Exercício 3.3.1. Implemente a questão Hartals, problema 10050 da UVa (http://acm.uva.es/p/v100/10050.html). Algumas festas ocorrem de t em t dias, mas nunca as sextas ou sábados. Dada a freqüência em que cada festa ocorre e quantos dias serão analisados, o objetivo é informar em quantos dias diferentes ocorreu pelo menos uma festa.

