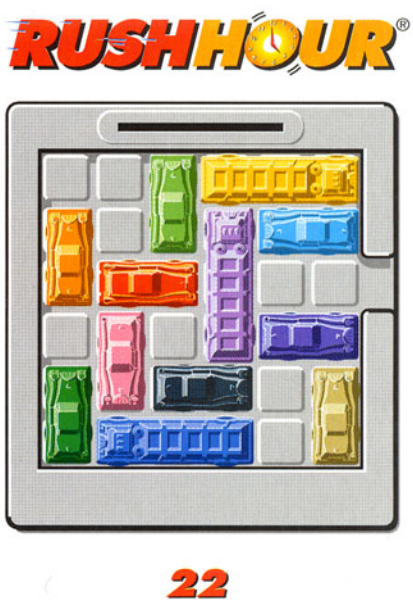

A *hora do rush* é um jogo cujo objetivo consiste a tirar um veículo de um estacionamento. Os veículos se deslocam em uma grade 6x6, seja horizontalmente ou verticalmente. Cada veículo ocupa 2 ou 3 casas de comprimento. Os veículos não podem sair do estacionamento, à exceção do carro vermelho que pode sair pela saída do estacionamento; este é o objetivo do jogo. Na figura abaixo temos um exemplo de configuração inicial:



O objetivo desta prática é escrever um programa que encontra a solução de tal problema em um número mínimo de deslocamentos.

Se você quiser se familiarizar com este jogo, ou testar as soluções encontradas, acesse o link `http://www.thinkfun.com/mathcounts/play-rush-hour`.

Duas classes C++ são fornecidas para vocês completarem com o código pedido nesta prática. As duas classes encontram-se no SIGAA.

1. Representação do problema

Adota-se a seguinte representação do problema. As 6 linhas são numeradas de cima para baixo, de 0 a 5, e as 6 colunas da esquerda para direita, de 0 a 5. A classe `RushHour` contem os quatro campos a seguir:

`nbcars` : o número de veículos ;

`color` : um vetor fornecendo a cor de cada veículo;

`horiz` : um vetor binário indicando para cada veículo se ele se desloca horizontalmente;

`len` : um vetor indicando o comprimento de cada veículo (`len[i]` representa o número de casas ocupadas pelo veículo i, vale 2 ou 3);

`moveon` : um vetor indicando em que linha (resp. coluna) se desloca um veículo horizontalmente (resp. verticalmente).

Por convenção, o carro vermelho é o veículo de índice 0 e tem comprimento 2.

O estado inicial do estacionamento em um momento qualquer do jogo é representado por um objeto da classe `State` que contem os campos a seguir:

`pos` : um vetor indicando a posição de cada veículo (a coluna mais à esquerda de um veículo horizontal ou a linha mais acima para um veículo vertical);

`c,d` e `prev` : indicam que este estado foi obtido a partir do estado `prev` deslocando o veículo `c` de `d` casas (`d` vale -1 ou +1: -1 indica um deslocamento para a esquerda ou para cima, +1 indica um deslocamento para a direita ou para baixo).

Complete o método `sucess` da classe `State` que indica se o estado trata-se do estado final do jogo (i.e., o carro vermelho está situado imediatamente na frente da saída).

Complete o construtor `State(State* s, int c, int d)` da classe `State` que constrói um estado a partir de `s` deslocando o veículo `c` de `d` casas (`d` vale -1 ou +1). Nós supomos que este deslocamento é possível. **Atenção**: é necessário construir um novo vetor `pos` para o novo estado, pois nós vamos querer conservar os estados precedentes.

Você poderá testar o seu código com a função a seguir:

```
void test1() {
  int positioning[] = {1,0,1,4,2,4,0,1};
  vector<int> start(positioning, positioning+8);
  State* s0 = new State(start);
  cout << (!s0->success()) << endl;
  State* s = new State(s0, 1, 1);
  cout << (s->prev == s0) << endl;
  cout << s0->pos[1] << " " << s->pos[1] << endl;
  s = new State(s,6,1);
  s = new State(s,1,-1);
  s = new State(s,6,-1);
  cout << s->equals(s0) << endl;
  s = new State(s0,1,1);
  s = new State(s,2,-1);
  s = new State(s,3,-1);
  s = new State(s,4,-1); s = new State(s, 4, -1);
  s = new State(s,5,-1); s = new State(s,5,-1); s = new State(s,5,-1);
  s = new State(s,6,1); s = new State(s, 6, 1); s = new State(s, 6, 1);
```

```
  s = new State(s,7,1); s = new State(s, 7, 1);
  s = new State(s,0,1); s = new State(s,0,1); s = new State(s,0,1);
  cout << (s.success()) << endl;
}
```

cujo resultado deve ser

```
true
true
0 1
true
true
```

2. Deslocamentos possíveis

A partir de agora, vamos trabalhar com a classe `RushHour`.

Nesta questão, nós vamos completar o método `list<State*> moves(State* s)` que determina o conjunto de estados que podemos atingir a partir do estado `s` efetuando um único deslocamento (de apenas uma casa).

Nós vamos utilizar uma matriz 6x6 `boolean free[6][6]` indicando quais são as casas do estacionamento que estão livres. Complete o método `initFree(State* s)` que inicializa a matriz `free` em função de um estado `s` dado, com a convenção que `true` designa uma casa livre. Adotaremos a convenção de que `free[i][j]` representa a casa (i, j) do estacionamento.

Atenção : todas as casas da matriz `free` devem ser inicializadas por `initFree`, pois trata-se de uma matriz que será utilizada várias vezes.

Utilize a função abaixo para testar o código produzido (a função é implementada dentro da classe `RushHour`):

```
void test2() {
        nbcars = 8;
        bool horiz1[] = {true, true, false, false, true, true, false, false};
        horiz.assign(horiz1, horiz1+8);
        int len1[] = {2,2,3,2,3,2,3,3};
        len.assign(len1,len1+8);
        int moveon1[] = {2,0,0,0,5,4,5,3};
        moveon.assign(moveon1,moveon1+8);
        int start1[] = {1,0,1,4,2,4,0,1};
        vector<int> start(start1,start1+8);
        State* s = new State(start);
        initFree(s);
        for (int i = 0; i < 6; i++) {
```

```
            for (int j = 0; j < 6; j++)
                cout << free[i][j] + "\t";
            cout << endl;
        }
    }
```

cujo resultado deve ser

```
false false true true true false
false true true false true false
false false false false true false
false true true false true true
false true true true false false
false true false false false true
```

Complete o método `moves` que retorna o conjunto de estados que podem ser atingidos a partir do estado `s`. Este conjunto é representado por uma `list<State*>`; a ordem nesta lista não é importante.

Podemos testar o código através do método a seguir:

```
static void test3(){

  nbcars = 12;
  bool horiz1[] = {true, false, true, false, false, true, false, true,
  false, true, false, true};
  horiz.assign(horiz1, horiz1+nbcars);
  int len1[] = {2,2,3,2,3,2,2,2,2,2,2,3};
  len.assign(len1,len1+12);
  int moveon1[] = {2,2,0,0,3,1,1,3,0,4,5,5};
  moveon.assign(moveon1,moveon1+nbcars);
  int start1[] = {1,0,3,1,1,4,3,4,4,2,4,1};
  vector<int> start(start1,start1+nbcars);
  State* s = new State(start);
  int start02[] = {1,0,3,1,1,4,3,4,4,2,4,2};
  vector<int> start2(start02,start02+nbcars);
  State* s2 = new State(start2);

  int n = 0;
  for (list<State*> L = moves(s); !L.empty(); n++) L.pop_front();
  cout << n << endl;
  n = 0;
  for (list<State*> L = moves(s2); !L.isEmpty(); n++) L.pop_front();
  cout << n << endl;
}
```

que deve fornecer o resultado a seguir (existem 5 deslocamentos possíveis a partir do estado `s` que representa a figura mostrada no início do trabalho, depois 6 a partir do estado `s2` quando o caminho azul de desloca para a direita):

```
5
6
```

3. À procura de uma solução

Nós vamos agora em busca de uma solução. Existem duas ideias principais para atingirmos este propósito:

- É preciso memorizar os estados que já encontramos, e para isto utilizaremos um conjunto hash (trata-se da variável `visited` no código, do tipo `hash_set<State*>`: consulte o link `http://www.sgi.com/tech/stl/hash_set.html` a fim de entender como se utiliza esta estrutura de dados).
- Estamos interessados na solução mais curta, então é preciso enumerar os estados por ordem crescente do número de deslocamentos que eles representam, para isso utilizaremos uma fila.

Se representamos o conjunto de estados por uma árvore cuja raiz é o estado inicial e cada nó `s` tem como filhos os estados que podemos atingir a partir de `s` realizando um deslocamento, desejamos percorrer os nós desta árvore *por níveis*. Para isto, é suficiente utilizarmos uma fila. Inicialmente a fila contem o estado inicial. Enquanto ela não está vazia, extraímos o primeiro estado da fila. Se um dos filhos deste estado corresponde a uma solução, terminamos o algoritmo. Caso contrário, todos os filhos, ainda não visitados, são colocados na fila e no conjunto `visited`.

Complete o método `State* solve(State* s)` para que ele implemente este algoritmo. O método deve retornar o estado correspondente a uma solução (aquela em que o carro vermelho está situado imediatamente na frente da saída do estacionamento).

Poderemos usar como teste o método a seguir:

```
void test4() {
    nbcars = 12;
    string color1[] = {"vermelho","verde claro","amarelo","laranja",
    "violeta claro","azul ceu","rosa","violeta","verde","preto","bege","azul"};
    color.assign(color1, color1+nbcars);
    bool horiz1[] = {true, false, true, false, false, true, false,
    true, false, true, false, true};
    horiz.assign(horiz1, horiz1+nbcars);
    int len1[] = {2,2,3,2,3,2,2,2,2,2,2,3};
```

```
    len.assign(len1,len1+nbcars);
    int moveon1[] = {2,2,0,0,3,1,1,3,0,4,5,5};
    moveon.assign(moveon1,moveon1+nbcars);
    int start1[] = {1,0,3,1,1,4,3,4,4,2,4,1};
    vector<int> start(start1,start1+nbcars);
    State* s = new State(start);
    int n = 0;
    for (s = solve(s); s.prev != null; s = s.prev) n++;
    cout << n << endl;
}
```

que deve resultar em (existem 46 deslocamentos a fazer para resolver o problema dado na figura do começo do trabalho) :

```
46
```

4. Recuperando uma solução

Complete o método `void printSolution(State* s)` que imprime uma solução, dado que o estado `s` corresponde a uma solução (o estado final). Note que os estados que formam uma solução são encadeados a partir de `s` seguindo-se o campo `prev`. Nós precisamos imprimir a solução na ordem correta.

Este método deve também imprimir o número total de deslocamentos. Podemos utilizar o campo `nbMoves` para isto.

Com o código a seguir, correspondente à configuração ilustrada no início da prática,

```
void solve22() {
    nbcars = 12;
    string color1[] = {"vermelho","verde claro","amarelo","laranja",
    "violeta claro","azul ceu","rosa","violeta","verde","preto","bege","azul"};
    color.assign(color1, color1+nbcars);
    bool horiz1[] = {true, false, true, false, false, true, false,
    true, false, true, false, true};
    horiz.assign(horiz1, horiz1+nbcars);
    int len1[] = {2,2,3,2,3,2,2,2,2,2,2,3};
    len.assign(len1,len1+nbcars);
    int moveon1[] = {2,2,0,0,3,1,1,3,0,4,5,5};
    moveon.assign(moveon1,moveon1+nbcars);
    int start1[] = {1,0,3,1,1,4,3,4,4,2,4,1};
    vector<int> start(start1,start1+nbcars);
    State* s = new State(start);
    s = solve(s);
```

```
    printSolution(s);
}
```

devemos obter o resultado a seguir:

```
46 deslocamentos
veiculo azul para a direita
veiculo preto para a direita
veiculo verde para cima
veiculo rosa para baixo
veiculo laranja para cima
veiculo vermelho para esquerda
veiculo verde claro para baixo
veiculo amarelo para esquerda
veiculo amarelo para esquerda
veiculo verde claro para baixo
veiculo verde claro para baixo
veiculo vermelho para direita
veiculo laranja para baixo
veiculo amarelo para esquerda
veiculo violeta claro para cima
veiculo violeta para esquerda
veiculo bege para cima
veiculo azul para direita
veiculo bege para cima
veiculo preto para direita
veiculo verde claro para baixo
veiculo violeta para esquerda
veiculo violeta para esquerda
veiculo violeta claro para baixo
veiculo violeta claro para baixo
veiculo azul ceu para esquerda
veiculo bege para cima
veiculo bege para cima
veiculo azul ceu para esquerda
veiculo azul ceu para esquerda
veiculo violeto claro para cima
veiculo violeto claro para cima
veiculo violeta para direita
veiculo violeta para direita
veiculo violeta para direita
veiculo rosa para cima
veiculo violeta claro para baixo
veiculo violeta claro para baixo
```

```
veiculo verte claro para cima
veiculo azul para esquerda
veiculo azul para esquerda
veiculo violeta claro para baixo
veiculo vermelho para direita
veiculo vermelho para direita
veiculo vermelho para direita
```

Nota: A sua solução pode diferir da apresentada aqui (isto vai depender do modo como a lista é construída dentro do método `moves`), porém a sua solução deve ter o mesmo número de deslocamentos (46).

A seguir temos dois outros problemas para que você teste o seu programa. O primeiro deve ter 16 deslocamentos:

```
void solve1() {
    nbcars = 8;
    string color1[] = {"vermelho","verde claro","violeta",
    "laranja","verde","azul ceu","amarelo","azul"};
    color.assign(color1, color1+nbcars);
    bool horiz1[] = {true, true, false, false, true,
    true, false, false};
    horiz.assign(horiz1, horiz1+nbcars);
    int len1[] = {2,2,3,2,3,2,3,3};
    len.assign(len1,len1+nbcars);
    int moveon1[] = {2,0,0,0,5,4,5,3};
    moveon.assign(moveon1,moveon1+nbcars);
    int start1[] = {1,0,1,4,2,4,0,1};
    vector<int> start(start1,start1+nbcars);
    State* s = new State(start);
    s = solve(s);
    printSolution(s);
}
```

e o segundo 81 deslocamentos:

```
void solve40() {
    nbcars = 13;
    string color1[] = {"vermelho","amarelo","verde claro","laranja","azul claro",
    "rosa","violeta claro","azul","violeta","verde","preto","bege","amarelo claro"};
    color.assign(color1, color1+nbcars);
    bool horiz1[] = {true, false, true, false, false, false, false,
    true, false, false, true, true, true};
    horiz.assign(horiz1, horiz1+nbcars);
    int len1[] = {2,3,2,2,2,2,3,3,2,2,2,2,2};
```

```
    len.assign(len1,len1+nbcars);
    int moveon1[] = {2,0,0,4,1,2,5,3,3,2,4,5,5};
    moveon.assign(moveon1,moveon1+nbcars);
    int start1[] = {3,0,1,0,1,1,1,0,3,4,4,0,3};
    vector<int> start(start1,start1+nbcars);
    State* s = new State(start);
    s = solve(s);
    printSolution(s);
}
```

*Este trabalho prático é de autoria de Steve Oudot e Jean-Christophe Filliâtre (Poly, France)