Não há problema que não possa ser solucionado pela paciência.

Chico Xavier

1 Motivação - a realidade

Bons programadores de jogos são preguiçosos (qualquer bom programador é preguiçoso). Não é que eles não queiram trabalhar, é que eles não desejam refazer o trabalho que já foi feito, especialmente se tiver sido bem feito. A linguagem C++ contém uma poderosa relação de trabalhos de programação prontos (e bem feitos) que podemos utilizar para facilitar a nossa vida. A STL (*Standard Template Library*) fornece um conjunto de contentores, algoritmos e iteradores, entre outras coisas.

E como contentores (leia-se vetores) pode nos ajudar a escrever programas melhores ? Bem, estes contentores permitem armazenar grandes quantidades de valores. Sim, vetores também permitem, mas os contentores oferecem mais flexibilidade e facilidade que um simples vetor.

Os algoritmos são funções comuns que os programadores de jogos utilizam repetidamente com grupos de valores. Os algoritmos disponíveis na STL facilitam atividades de pesquisa, ordenação, cópia, inserção e outras *mágicas*.

E, finalmente, os iteradores são objetos que identificam os elementos contidos nestes contentores que podem ser manipulados para se mover entre os diversos elementos. Os iteradores são a base do funcionamento para os algoritmos disponíveis na STL.

2 Revisitando o programa do inventário - um pouco mais do mesmo, *again*

Já trabalhamos com vetores anteriormente e não existe muito mistério no seu entendimento. O objeto *vector* permite implementar um *vetor dinâmico*. Um *vetor tradicional* tem o seu tamanho definido no momento da escrita do programa (tamanho fixo), já um vetor dinâmico permite incluir e remover elementos do seu vetor. O programa 1 implementa um vetor dinâmico:

Programa 1: Trabalhando com *vectors*.

```
1  // uma nova versão para o programa de inventário
2  // programa_001.cpp
3  #include "biblaureano.h"
4
5  int main() {
6     //declaração do vetor
7     vector<string> inventario;
8
9     //acrescentando itens ao final
10    inventario.push_back("espada");
11    inventario.push_back("armadura");
12    inventario.push_back("escudo");
13
14    //verificando a quantidade de itens do inventário
15    cout << "Seu inventário tem " << inventario.size() << " items." << endl;
16
17    //percorrendo todos os itens do inventário
18    cout << "O inventário tem os seguintes itens:" << endl;
19    for( int i = 0; i < inventario.size(); ++i) {
20       cout << "\t" << inventario[i] << endl;
21    }
22
```

```
23    //substituindo itens no inventario
24    inventario[0]= "machado de guerra";
25
26    //percorrendo todos os itens do inventário
27    cout << "O inventário tem os seguintes itens:" << endl;
28    for( int i = 0; i < inventario.size(); ++i) {
29       cout << "\t" << inventario[i] << endl;
30    }
31
32    //acessando um método string
33    cout << "O item " << inventario[0]
34         << " tem "  << inventario[0].size() << " letras!!" << endl;
35
36    //destruindo um item
37    cout << "Seu " << inventario.back() << " quebrou durante a luta..." << endl;
38    inventario.pop_back();
39
40    //percorrendo todos os itens do inventário
41    cout << "O inventário tem os seguintes itens:" << endl;
42    for( int i = 0; i < inventario.size(); ++i) {
43       cout << "\t" << inventario[i] << endl;
44    }
45
46    //apagando tudo
47    cout << "Você foi roubado..." << endl;
48    inventario.clear();
49
50    if( inventario.empty() ) {
51       cout << "Você não possue mais itens..." << endl;
52    }
53    else {
54       cout << "Você ainda possui alguns itens..." << endl;
55    }
56
57    cout << "Game over!!!" << endl;
58    return 0;
59 }
```

2.1 Entendendo o programa

Inicialmente, um vetor de *strings* é declarado para uso. Repare que não foi definido o seu tamanho.

```
1    vector<string> inventario;
```

Cuidado com a sintaxe de uso do objeto *vector*. Repare que utilizamos o comando *vector* e entre < e > o tipo da variável. Seria o equivalente a declarar um vetor tradicional da seguinte forma:

```
1    string inventario[tamanho do vetor];
```

Só que neste caso, não precisamos definir o tamanho inicial, pois trata-se de um vetor dinâmico e podemos incluir quantos itens quisermos na sequência. Claro, se quisermos já declarar com um tamanho definido:

```
1    vector<string> inventario(10);
```

Neste caso nossa variável já começa com 10 itens alocados (mas vazios). Se quisermos preencher o vetor no momento da declaração:

```
1    vector<string> inventario(10, "slot disponível");
```

Desta forma todos os 10 itens terão o conteúdo *slot disponível*.

Após a definição da variável que conterá o nosso novo inventário, precisamos incluir itens no vetor. Para tal está disponível o método *push_back()* que permite incluir novos elementos ao final do vetor:

```
1  //acrescentando itens ao final
2  inventario.push_back("espada");
3  inventario.push_back("armadura");
4  inventario.push_back("escudo");
```

E a partir deste momento o nosso vetor contém um conjunto de métodos que podemos utilizar. Qualquer semelhança com algumas coisas de string não é mera coincidência. Por exemplo, para determinar o tamanho do vetor, utilizamos o método *size()*:

```
1  //verificando a quantidade de itens do inventário
2  cout << "Seu inventário tem " << inventario.size() << " itens." << endl;
```

Para acessar elementos do vetor dinâmico, basta utilizar a mesma sintaxe para uso dos vetores tradicionais:

```
1  //percorrendo todos os itens do inventário
2  cout << "O inventário tem os seguintes itens:" << endl;
3  for( int i = 0; i < inventario.size(); ++i) {
4     cout << "\t" << inventario[i] << endl;
5  }
6
7  //substituindo itens no inventario
8  inventario[0]= "machado de guerra";
```

Como estamos trabalhando com um vetor de *strings*, podemos acessar um método da string de dentro do vetor:

```
1  //acessando um método string
2  cout << "O item " << inventario[0]
3       << " tem "  << inventario[0].size() << " letras!!" << endl;
```

Também é possível retirar (destruir) um elemento do final do vetor, basta utilizar o método *pop_back()* do objeto *vector*:

```
1  //destruindo um item
2  cout << "Seu escudo quebrou durante a luta..." << endl;
3  inventario.pop_back();
```

Lembre-se quando falamos que qualquer semelhança com uma *string* não é mera coincidência ? Veja como eliminar todos os itens do vetor:

```
1  //apagando tudo
2  cout << "Você foi roubado..." << endl;
3  inventario.clear();
```

E verificar o que houve com o seu inventário:

```
1  if( inventario.empty() ) {
2     cout << "Você não possui mais itens..." << endl;
3  }
4  else {
5     cout << "Você ainda possui alguns itens..." << endl;
6  }
```

3 Caminhando nos elementos do inventário - o uso de iteradores

Praticamente todos os recursos disponíveis para uso do contentor *vector* e dos algoritmos disponíveis exigem que sejam realizados através de um *iterator*. Com um iterador é possível percorrer uma sequência de elementos, acessar elementos entre outras atividades.

Iteradores são valores que identificam um determinado elemento em um contentor. Dado um iterador, você pode acessar o valor do elemento. Dado o tipo de iterador, você pode alterar o valor. Iteradores também podem mover-se entre os elementos através dos operadores aritméticos. O programa 2 demonstra como utilizar os iteradores:

Programa 2: Trabalhando com *vectors* e *iterators*.

```
// outras coisas que você pode fazer no seu inventário
// programa_002.cpp
#include "biblaureano.h"

int main()
{
   //declaração do vetor
   vector<string> inventario;

   //declaração do iterator
   vector<string>::iterator meuIterador;

   //acrescentando itens ao final
   inventario.push_back("espada");
   inventario.push_back("armadura");
   inventario.push_back("escudo");

   //pegando o último item
   cout << "Último item adicionado foi "
        << inventario.back()
        << "." << endl;

   cout << "O inventário tem "
        << inventario.size()
        << " itens!" << endl;

   for( meuIterador = inventario.begin(); meuIterador != inventario.end(); ++meuIterador) {
      cout << "\t " << *meuIterador << endl;
   }

   //inserindo um item no inventário na segunda posição
   meuIterador = inventario.begin();
   *meuIterador = "xaxo";
   //lembre-se que o operador de pré-incremento ++
   //incrementa primeiro a posição
   inventario.insert( ++meuIterador, "punhal");
   //inventario.insert( inventario.begin()+1, "punhal");

   cout << "O inventário tem "
        << inventario.size()
        << " itens!" << endl;

   for( meuIterador = inventario.begin(); meuIterador != inventario.end(); ++meuIterador) {
      cout << "\t " << *meuIterador << endl;
   }

   //inserindo um item no começo
   inventario.insert(inventario.begin(),"faca sem gume");

   cout << "O inventário tem "
        << inventario.size()
        << " itens!" << endl;

   for( meuIterador = inventario.begin(); meuIterador != inventario.end(); ++meuIterador) {
      cout << "\t " << *meuIterador << endl;
   }

   //pegando o primeiro item do inventário
   cout << "Primeiro item disponível no inventário é "
        << inventario.front() << "." << endl;

   //alterando o primeiro elemento do vetor
   meuIterador = inventario.begin();
   *meuIterador = "peixeira";

   //pegando o primeiro item do inventário
```

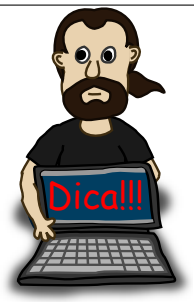
```
67      cout << "Agora o primeiro disponível no inventário é "
68           << inventario.front() << "." << endl;
69
70      //apagando o terceiro elemento
71      cout << "Destruindo uma arma do inventário..." << endl;
72      inventario.erase(inventario.begin()+2);
73
74      cout << "O inventário tem "
75           << inventario.size()
76           << " itens!" << endl;
77
78      for( meuIterador = inventario.begin(); meuIterador != inventario.end(); ++meuIterador) {
79         cout << "\t " << *meuIterador << endl;
80      }
81
82      cout << "Game over!!!" << endl;
83      return 0;
84  }
```

3.1 Entendendo o programa - e vendo algumas novidades

Inicialmente são declaradas as variáveis de uso, inclusive a variável iteradora que será utilizada para manipular os elementos do vetor:

```
1       //declaração do vetor
2       vector<string> inventario;
3
4       //declaração do iterator
5       vector<string>::iterator meuIterador;
```

Um iterador deve ser declarado com o mesmo tipo do vetor. Por exemplo, não é possível declarar um vetor para inteiros e manipular com um iterador de strings.



Podemos ter dois tipos de iteradores: um apenas para manipulação das informações (somente leitura) e que pode ser declarado como tal e outra que permite a manipulação dos dados. Um iterador apenas para leitura deve ser declarado como:

```
1       vector<string>::const_iterator meuIterador;
```

É possível retornar o último elemento inserido no vetor. Para tal basta utilizar o método *back()*:

```
//pegando o último item
cout << "Último item adicionado foi "
     << inventario.back()
     << "." << endl;
```

E chegou o momento de utilizar iteradores. Neste exemplo, a variável *meuIterador* é utilizada para percorrer todos os elementos do vetor. O método *begin()* retorna o primeiro elemento do vetor e o método *end()* permite verificar que se chegou ao final do vetor. Para caminhar pelo vetor, podemos utilizar qualquer operador aritmético convencional (neste caso está sendo o utilizado operador de pré-incremento ++). E finalmente, atenção no uso do iterador, repare no uso do $*$ (asterisco) antes do nome da variável *meuIterador*. O esquecimento do uso do asterisco causará erros no funcionamento do programa.

```
for( meuIterador = inventario.begin(); meuIterador != inventario.end(); ++meuIterador) {
   cout << "\t " << *meuIterador << endl;
}
```

O método *end()* não retorna o último elemento do vetor e sim um indicativo que se chegou ao final do vetor.

Na sequência incluímos um novo elemento no meio do vetor. Devemos indicar a posição para a inserção de um novo valor, só para que indicar a posição não informamos simplesmente um número, como 2 para indicar a segunda posição, por exemplo. Devemos indicar o elemento onde queremos inserir um novo valor. Para indicar o elemento, devemos utilizar um iterador. Neste caso, posicionamos no início com o método *begin()* e incrementamos uma posição para indicar posicionar na segunda posição do vetor e depois fazemos a inclusão com o método *insert()*:

```
//inserindo um item no inventário na segunda posição
meuIterador = inventario.begin();
//lembre-se que o operador de pré-incremento ++
//incrementa primeiro a posição
inventario.insert( ++meuIterador, "punhal");
```

Para incluir um novo elemento na primeira posição do vetor, basta utilizar o método *insert()* em conjunto com o método *begin()*:

```
//inserindo um item no começo
inventario.insert(inventario.begin(),"faca sem gume");
```

E como verificação do novo conteúdo, utilizamos o método *front()* que retorna o conteúdo da primeira posição:

```
//pegando o primeiro item do inventário
cout << "Primeiro item disponível no inventário é "
     << inventario.front() << "." << endl;
```

Podemos alterar um elemento do vetor utilizando o iterador. Lembre-se que declaramos o nosso iterador possibilitando alterações. Novamente preste atenção na sintaxe de uso e repare no uso do $*$ (asterisco) antes do nome da variável:

```
//alterando o primeiro elemento do vetor
meuIterador = inventario.begin();
*meuIterador = "peixeira";
```

O método *front()* é utilizado novamente apenas mostrar que o conteúdo do vetor foi alterado:

```
1    //pegando o primeiro item do inventário
2    cout << "Agora o primeiro disponível no inventário é "
3         << inventario.front() << "." << endl;
```

Finalmente, podemos excluir um elemento em qualquer posição do vetor. Basta indicar a posição, utilizando iteradores, e chamar o método *erase()*:

```
1    //apagando o terceiro elemento
2    cout << "Destruindo uma arma do inventário..." << endl;
3    inventario.erase(inventario.begin()+2);
```

A utilização do método *erase()* modifica as posições dos iteradores. Por exemplo, o trecho de código a seguir não funcionaria adequadamente:

```
1  for( meuIterador = inventario.begin();
2       meuIterador != inventario.end();
3       ++meuIterador) {
4     inventario.erase(meuIterador);
5  }
```

4 Como ser preguiçoso - O uso dos algoritmos disponíveis para simplificar a nossa vida

Como comentado anteriormente, é possível utilizarmos vários algoritmos já disponíveis para uso na STL. Vários destes algoritmos precisam do uso conjunto de iteradores para a correta manipulação das informações. O programa 3 implementa alguns destes algoritmos. Lembre-se que podemos utilizar estes algoritmos para implementar alguns jogos.

Programa 3: Trabalhando com *iterators* e funções da *algorithm*.

```
1  // números mágicos e outros recursos
2  // programa_003.cpp
3  #include "biblaureano.h"
4
5  void mostraNumeros( const vector<int> numeros);
6
7  int main()
8  {
9     //declaração do vetor
10    vector<int> numeros;
11
12    //declaração do iterator local
13    vector<int>::const_iterator meuIterador;
14
15    //números aleatórios
16    numeros.push_back(randomico(1,20));
17    numeros.push_back(randomico(1,20));
18    numeros.push_back(randomico(1,20));
19    numeros.push_back(randomico(1,20));
20    numeros.push_back(randomico(1,20));
21
22    cout << "Números sorteados:" << endl;
23    mostraNumeros(numeros);
24
```

```
25      int seuNumero;
26      seuNumero = readInt("Entre com o seu número:");
27
28      meuIterador = find(numeros.begin(),numeros.end(),seuNumero);
29      //se não chegou no final
30      if( meuIterador != numeros.end()) {
31         cout << "Seu número está na relação..." << endl;
32      }
33      else {
34         cout << "Seu número não está na relação!!!" << endl;
35      }
36
37      //verificando a quantidade de ocorrências
38      seuNumero = readInt("Entre com outro seu número:");
39
40      int qtdOcorrencias = count( numeros.begin(), numeros.end(), seuNumero);
41      cout << seuNumero << " apareceu " << qtdOcorrencias << " vezes." << endl;
42
43      //Revertendo os números
44      cout << "Revertendo os números!" << endl;
45      reverse(numeros.begin(), numeros.end());
46      mostraNumeros(numeros);
47
48      //ordenando os números
49      cout << "Ordenando os números!" << endl;
50      sort(numeros.begin(), numeros.end());
51      mostraNumeros(numeros);
52
53      //incluindo outros números
54      //declaração do vetor
55      cout << "Criando outros números..." << endl;
56      vector<int> outrosNumeros;
57      outrosNumeros.push_back(randomico(1,20));
58      outrosNumeros.push_back(randomico(1,20));
59      outrosNumeros.push_back(randomico(1,20));
60      outrosNumeros.push_back(randomico(1,20));
61      mostraNumeros(outrosNumeros);
62
63      //realizando o merge
64      cout << "Juntando todos os números..." << endl;
65      vector<int> todosNumeros( numeros.size() + outrosNumeros.size());
66      merge( numeros.begin(),numeros.end(),
67             outrosNumeros.begin(), outrosNumeros.end(),
68             todosNumeros.begin());
69      mostraNumeros(todosNumeros);
70
71      //bagunçando os números
72      cout << "Embaralhando os números..." << endl;
73      srand(time(0)); //esta linha é necessário para a próxima funcionar
74      random_shuffle(todosNumeros.begin(), todosNumeros.end());
75      mostraNumeros(todosNumeros);
76
77      //pegando max e min
78      meuIterador = max_element(todosNumeros.begin(), todosNumeros.end());
79      cout << "Maior número: " << *meuIterador << endl;
80
81      meuIterador = min_element(todosNumeros.begin(), todosNumeros.end());
82      cout << "Menor número: " << *meuIterador << endl;
83
84      cout << "Game over!!!" << endl;
85      return 0;
86  }
87
88  void mostraNumeros( const vector<int> numeros) {
89      //declaração do iterator local
90      vector<int>::const_iterator iterador;
91      for( iterador = numeros.begin(); iterador != numeros.end(); ++iterador) {
92         cout << "\t " << *iterador << endl;
93      }
94      return;
95  }
```

4.1 Entendendo o programa - nada de novo no *front*

Inicialmente, declaramos as variáveis iniciais e incluímos alguns valores na variável:

```
1   //declaração do vetor
2   vector<int> numeros;
3
4   //declaração do iterator local
5   vector<int>::const_iterator meuIterador;
6
7   //números aleatórios
8   numeros.push_back(randomico(1,20));
9   numeros.push_back(randomico(1,20));
10  numeros.push_back(randomico(1,20));
11  numeros.push_back(randomico(1,20));
12  numeros.push_back(randomico(1,20));
```

Repare que a variável iteradora foi declarada como *const_iterator*, pois neste programa não altera nenhum dado utilizando a variável iteradora.

A primeira novidade, uma chamada a função *mostraNumeros()*. Esta função recebe o vetor dinâmico e apenas imprime na tela. O trecho de código a seguir:

```
1   cout << "Números sorteados:" << endl;
2   mostraNumeros(numeros);
```

Resulta na execução da função. Neste caso o argumento de entrada também é declarado como constante, pois o vetor não sofre nenhum alteração ou seja é utilizada somente para leitura:

```
1  void mostraNumeros( const vector<int> numeros)
2  {
3     //declaração do iterator local
4     vector<int>::const_iterator Iterador;
5     for( iterador = numeros.begin(); iterador != numeros.end(); ++iterador) {
6        cout << "\t " << *iterador << endl;
7     }
8     return;
9  }
```

A função *mostraNumeros* também poderia ser escrita utilizando notação de vetores:

```
1 void mostraNumeros( const vector<int> numeros) {
2    for( int i = 0;  i < numeros.size(); ++i) {
3       cout << "\t " << numeros[i] << endl;
4    }
5    return;
6 }
```

Da mesma forma que era possível pesquisar uma letra ou sequência de letras numa string, é possível pesquisar se existe um determinado valor dentro do nosso contentor. Para tal, basta utilizar o algoritmo *find()*. Neste caso, se o valor não está contido é retornado o final da lista para o nosso iterador:

```
1  int seuNumero;
2  seuNumero = readInt("Entre com o seu número:");
3
4  meuIterador = find(numeros.begin(),numeros.end(),seuNumero);
5  //se não chegou no final
6  if( meuIterador != numeros.end()) {
7     cout << "Seu número está na relação..." << endl;
8  }
9  else {
10    cout << "Seu número não está na relação!!!" << endl;
11 }
```

Também é possível contar quantas ocorrências existem de um valor no nosso contentor. O algoritmo *count()* informa a quantidade de repetições entre o inicio e fim informados:

```
1  //verificando a quantidade de ocorrências
2  seuNumero = readInt("Entre com outro seu número:");
3
4  int qtdOcorrencias = count( numeros.begin(), numeros.end(), seuNumero);
5  cout << seuNumero << " apareceu " << qtdOcorrencias << " vezes." << endl;
```

O algoritmo *reverse()* inverte o conteúdo de um vetor. A última posição vira primeira e assim por diante:

```
1  cout << "Revertendo os números!" << endl;
2  reverse(numeros.begin(), numeros.end());
3  mostraNumeros(numeros);
```

Já o algoritmo *sort()* organiza a relação de valores em ordem crescente (menor para o maior):

```
1  //ordenando os números
2  cout << "Ordenando os números!" << endl;
3  sort(numeros.begin(), numeros.end());
4  mostraNumeros(numeros);
```

Se quisermos ordenar um vetor em ordem descrente, basta utilizar os algoritmos *sort()* e *reverse()* em conjunto. Veja o exemplo:

```
1    sort(numeros.begin(), numeros.end());
2    reverse(numeros.begin(), numeros.end());
3    mostraNumeros(numeros);
```

Criamos outros números, apenas para ilustrar a utilização do algoritmo *merge()*:

```
1    //incluindo outros números
2    //declaração do vetor
3    cout << "Criando outros números..." << endl;
4    vector<int> outrosNumeros;
5    outrosNumeros.push_back(randomico(1,20));
6    outrosNumeros.push_back(randomico(1,20));
7    outrosNumeros.push_back(randomico(1,20));
8    outrosNumeros.push_back(randomico(1,20));
9    mostraNumeros(outrosNumeros);
```

Para então chamarmos o algoritmo *merge()*. Este algoritmo permite somar duas sequências de um vetor. Basta informar a sequência de cada vetor (neste exemplo realizamos a soma do conteúdo integral de ambos os vetores) e indicar o vetor (e a posição inicial do vetor) que irá receber a soma das sequências. Repare que o vetor *todosNumeros* teve o seu tamanho previamente definido:

```
1    //realizando o merge
2    cout << "Juntando todos os números..." << endl;
3    vector<int> todosNumeros( numeros.size() + outrosNumeros.size());
4    merge( numeros.begin(),numeros.end(),
5           outrosNumeros.begin(), outrosNumeros.end(),
6           todosNumeros.begin());
7    mostraNumeros(todosNumeros);
```

O algoritmo *merge()* não cria novas sequências no novo vetor. O vetor que receberá o valor deve ter espaço suficiente para receber os novos valores. Por isto no exemplo o vetor *todosNumeros* teve o seu tamanho definido inicialmente.

É possível embaralhar o conteúdo do contentor. O algoritmo *random_shuffle()* faz exatamente isto. É necessário realizar uma chamada a função *srand()* antes do uso do algoritmo *random_shuffle()* (já fizemos no início do programa, só colocamos novamente por questões didáticas):

```
1    cout << "Embaralhando os números..." << endl;
2    srand(time(0)); //esta linha é necessário para a próxima funcionar
3    random_shuffle(todosNumeros.begin(), todosNumeros.end());
4    mostraNumeros(todosNumeros);
```

Para utilizar a função *random_shuffle()* é necessário chamar ao menos uma vez o gerador de sementes aleatórios:

```
1 srand(time(0));
```

Esta chamada não precisa ser realizada a todo instante, basta chamar uma vez no inicio do seu código:

```
1  int main(){
2      //inicio do código main
3      srand(time(0));
4      ...
5      //outros comandos
6      ...
7      return 0;
8  }
```

Finalmente, também podemos pegar o maior e o menor elemento do nosso contentor. Basta utilizar os algoritmos *max_element* e *min_element*:

```
1    //pegando max e min
2    meuIterador = max_element(todosNumeros.begin(), todosNumeros.end());
3    cout << "Maior número: " << *meuIterador << endl;
4
5    meuIterador = min_element(todosNumeros.begin(), todosNumeros.end());
6    cout << "Menor número: " << *meuIterador << endl;
```

Todos os algoritmos vistos trabalham com sequências. Neste exemplo sempre informamos o início (*begin()*) e o fim (*end()*). Para trabalhar com sequências, basta informar um iterador para o inicio e outra para o fim. Por exemplo, se quisermos ordenar apenas os elementos 2,3,4 e 5 de uma relação qualquer:

```
1     sort(numeros.begin()+1, numeros.begin()+5);
```

Neste exemplos vimos apenas uma pequena amostra dos algoritmos disponíveis. A biblioteca STL dispõem de outros algoritmos e que podemos utilizar em nossos programas.

5 Uma aplicação do uso de vetores em jogos - aproveitando para revermos alguns conceitos antigos

O programa 4 ilustra uma aplicação para o conhecido Jogo da Velha (figura 1). Este programa, além do uso de vetores, ilustra também a aplicação de vários outros conceitos vistos até o momento, tais como: uso de constantes, uso de funções, retorno de funções, aplicações com vetores, programação modular, etc.

```
Bem-vindo ao último confronto homem-máquina: Velha 100 AC.
-- onde cérebro humano é combaterá o processador de silício.

Faça a sua jogada! Introduza um número, 0 - 8.  O número
corresponde à posição desejada do tabuleiro, tal como ilustrado:

       0 | 1 | 2
       ---------
       3 | 4 | 5
       ---------
       6 | 7 | 8

Prepare-se, humano. A batalha está prestes a começar.

Você quer o primeiro movimento?:[sn]s

Pegue o primeiro movimento, você irá precisar mesmo...

          |   |                       0 | 1 | 2
         ---------                     ---------
          |   |                       3 | 4 | 5
         ---------                     ---------
          |   |                       6 | 7 | 8

Qual o seu movimento? (0 - 8):2
Bom...

          |   | X                     0 | 1 | 2
         ---------                     ---------
          |   |                       3 | 4 | 5
         ---------                     ---------
          |   |                       6 | 7 | 8

Eu escolho o espaço 4

          |   | X                     0 | 1 | 2
         ---------                     ---------
          | 0 |                       3 | 4 | 5
         ---------                     ---------
          |   |                       6 | 7 | 8

Qual o seu movimento? (0 - 8):
```

Figura 1: O emocionante jogo da velha.

Programa 4: Um exemplo completo de jogo da velha.

```
// jogo da velha
// utilizando o conceitos de vectores
// e aplicando vários conceitos já vistos até o momento
// funções, constantes, estruturas de repetição e condição
// programa_004.cpp
#include "biblaureano.h"

// constantes globais
const char X = 'X';
const char O = 'O';
const char VAZIO = ' ';
const char EMPATE = 'E';
const char NINGUEM = 'N';

// protótipos
void instrucoes();
string simOuNao(string questao);
int numero(string questao, int alto, int baixo=0);
char pecaHumana();
char oponente(char peca);
void mostrarTabuleiro(const vector<char> tabuleiro);
inline bool eLegal(int movimento, const vector<char> tabuleiro);
int movimentoHumano(const vector<char> tabuleiro, char humano);
int movimentoComputador(vector<char> tabuleiro, char computador);
char vencedor(const vector<char> tabuleiro);
void anuncioVencedor(char vencedor, char computador, char humano);

// programa principal
int main() {
   int movimento;
   const int NUM_SQUARES = 9;
   vector<char> tabuleiro(NUM_SQUARES, VAZIO);

   instrucoes();
   char humano = pecaHumana();
   char computador = oponente(humano);
   char turno = X;
   mostrarTabuleiro(tabuleiro);

   //enquanto não houver vencedor
   while (vencedor(tabuleiro) == NINGUEM){
      if (turno == humano) {
         movimento = movimentoHumano(tabuleiro, humano);
         tabuleiro[movimento] = humano;
      }
      else {
         movimento = movimentoComputador(tabuleiro,computador);
         tabuleiro[movimento] = computador;
      }

      mostrarTabuleiro(tabuleiro);
      turno = oponente(turno);
   }

   anuncioVencedor(vencedor(tabuleiro), computador, humano);

   return 0;
}

// funções
void instrucoes() {
   cout << "Bem-vindo ao último confronto homem-máquina: Velha 100 AC." << endl;
   cout << "-- onde cérebro humano é combaterá o processador de silício." << endl << endl;

   cout << "Faça a sua jogada! Introduza um número, 0 - 8. O número" << endl;
   cout << "corresponde à posição desejada do tabuleiro, tal como ilustrado:" << endl << endl;

   cout << "         0 | 1 | 2" << endl;
   cout << "        -----------" << endl;
   cout << "         3 | 4 | 5" << endl;
   cout << "        -----------" << endl;
```

```
72    cout << "        6 | 7 | 8" << endl << endl;
73
74    cout << "Prepare-se, humano. A batalha está prestes a começar." << endl << endl;
75    return;
76 }
77
78 string simOuNao(string questao) {
79    string resposta;
80    questao += ":[sn]";
81    do {
82       resposta = readString(questao);
83    } while (resposta != "s" && resposta != "n");
84
85    return resposta;
86 }
87
88 int numero(string questao, int alto, int baixo) {
89    int numero;
90    questao += " (" + numeroToString(baixo) + " - " + numeroToString(alto) + "):";
91    do {
92       numero = readInt(questao);
93    } while (numero> alto || numero < baixo);
94
95    return numero;
96 }
97
98 //verifica a peça (X ou O) do jogador humano
99 char pecaHumana() {
100    string primeiro = simOuNao("Você quer o primeiro movimento?");
101    if (primeiro == "s") {
102       cout << endl << "Pegue o primeiro movimento, você irá precisar mesmo..." << endl;
103       return X;
104    }
105    else {
106       cout << endl << "Sua coragem será a sua ruína... Eu vou primeiro!" << endl;
107       return O;
108    }
109 }
110
111 //verifica qual o próximo a jogar
112 char oponente(char peca) {
113    if (peca == X) {
114       return O;
115    }
116    else {
117       return X;
118    }
119 }
120
121 //mostra a situação atual do jogo e a referência do tabuleiro
122 void mostrarTabuleiro(const vector<char> tabuleiro) {
123    cout << "\n\t" << tabuleiro[0] << " | " << tabuleiro[1] << " | " << tabuleiro[2] << "\t\t\t0 | 1 | 2";
124    cout << "\n\t" << "---------"  << "\t\t\t---------";
125    cout << "\n\t" << tabuleiro[3] << " | " << tabuleiro[4] << " | " << tabuleiro[5] << "\t\t\t3 | 4 | 5";;
126    cout << "\n\t" << "---------"  << "\t\t\t---------";
127    cout << "\n\t" << tabuleiro[6] << " | " << tabuleiro[7] << " | " << tabuleiro[8] << "\t\t\t6 | 7 | 8";;
128    cout << endl << endl;
129    return;
130 }
131
132 //verifica se houve vencedor
133 char vencedor(const vector<char> tabuleiro) {
134     // combinações possíveis de vencedores
135     const int SEQUENCIAS_VENCEDORAS[24] = { 0, 1, 2,
136                                             3, 4, 5,
137                                             6, 7, 8,
138                                             0, 3, 6,
139                                             1, 4, 7,
140                                             2, 5, 8,
141                                             0, 4, 8,
142                                             2, 4, 6 };
143    for(int i = 0; i<24; i+=3) {
144       if( (tabuleiro[ SEQUENCIAS_VENCEDORAS[i] ]   != VAZIO) &&
```

```
145             (tabuleiro[ SEQUENCIAS_VENCEDORAS[i] ]   == tabuleiro[ SEQUENCIAS_VENCEDORAS[i+1] ]) &&
146             (tabuleiro[ SEQUENCIAS_VENCEDORAS[i+1] ] == tabuleiro[ SEQUENCIAS_VENCEDORAS[i+2] ])) {
147            return tabuleiro[ SEQUENCIAS_VENCEDORAS[i]];
148         }
149
150      }
151      // se ninguém ganhou, checando para ver se ainda existem espaços vazios no tabuleiro
152      if (count(tabuleiro.begin(), tabuleiro.end(), VAZIO) == 0) {
153         return EMPATE;
154      }
155
156      // ninguém ganhou e ainda existem espaços vazios, o jogo não acabou
157      return NINGUEM;
158   }
159
160   //verifica se o movimento é válido
161   inline bool eLegal(int movimento, const vector<char> tabuleiro) {
162      return (tabuleiro[movimento] == VAZIO);
163   }
164
165   //retorna o movimento do jogador
166   int movimentoHumano(const vector<char> tabuleiro, char humano) {
167      int movimento = numero("Qual o seu movimento?", (tabuleiro.size() - 1));
168      while (!eLegal(movimento, tabuleiro)) {
169         cout << endl << "Este espaço está ocupado, humano tolo." << endl;
170         movimento = numero("Qual o seu movimento?", (tabuleiro.size() - 1));
171      }
172      cout << "Bom..." << endl;
173      return movimento;
174   }
175
176   //retorna o movimento do computador
177   int movimentoComputador(vector<char> tabuleiro, char computador) {
178      cout << "Eu escolho o espaço ";
179
180      // se o computador pode ganhar na próxima jogada...
181      for(int movimento = 0; movimento < tabuleiro.size(); ++movimento) {
182         if (eLegal(movimento, tabuleiro)) {
183
184            tabuleiro[movimento] = computador;
185
186            if(vencedor(tabuleiro) == computador) {
187               cout << movimento << endl;
188               return movimento;
189            }
190             // movimento verificado, tenho que desfaze-lo
191            tabuleiro[movimento] = VAZIO;
192         }
193      }
194
195      // se o jogador humano for ganhar na próxima jogada, é necessário bloquea-lo
196      char humano = oponente(computador);
197      for(int movimento = 0; movimento < tabuleiro.size(); ++movimento)
198      {
199         if (eLegal(movimento, tabuleiro)) {
200
201            tabuleiro[movimento] = humano;
202
203            if (vencedor(tabuleiro) == humano) {
204               cout << movimento << endl;
205               return movimento;
206            }
207            // movimento verificado, tenho que desfaze-lo
208            tabuleiro[movimento] = VAZIO;
209         }
210      }
211
212      // a melhor sequência de jogadas
213      const int MELHORES_MOVIMENTOS[]={4, 8, 6, 0, 1, 3, 5, 7};
214      // se não posso ganhar nesta jogada, tento escolher a melhor jogada possível
215      for(int i = 0; i < tabuleiro.size(); ++i) {
216         int movimento = MELHORES_MOVIMENTOS[i];
217         if (eLegal(movimento, tabuleiro)){
```

```
218             cout << movimento << endl;
219             return movimento ;
220         }
221     }
222 }
223
224 void anuncioVencedor(char vencedor, char computador, char humano) {
225     if(vencedor == computador ) {
226         cout << vencedor << " ganhou!" << endl;
227         cout << "Como eu previ, humano, eu sou vitorioso mais uma vez -- a prova" << endl;
228         cout << "que os computadores são superiores aos seres humanos em todos os aspectos." << endl;
229     }
230
231   else if (vencedor == humano) {
232         cout << vencedor << " ganhou!" << endl;
233         cout << "Não, não! Não pode ser! De alguma forma, você me enganou, humano." << endl;
234         cout << "Mas esta será a última vez, eu prometo!!" << endl;
235     }
236   else{
237         cout << "Temos um empate!" << endl;
238         cout << "Você tem sorte, humano, e de alguma forma conseguiu empatar comigo." << endl;
239         cout << "Celebre... pois é o melhor que irá conseguir de mim." << endl;;
240   }
241   return;
242 }
```

5.1 Entendendo o programa passo-a-passo

Inicialmente temos as declarações da variáveis globais (constantes) que serão utilizadas em vários pontos do jogo:

```
1 // constantes globais
2 const char X = 'X';
3 const char O = 'O';
4 const char VAZIO = ' ';
5 const char EMPATE = 'E';
6 const char NINGUEM = 'N';
```

Na sequências temos a definição de todos os protótipos (assinaturas) das funções escritas para o jogo:

```
1 // protótipos
2 void instrucoes();
3 string simOuNao(string questao);
4 int numero(string questao, int alto, int baixo=0);
5 char pecaHumana();
6 char oponente(char peca);
7 void mostrarTabuleiro(const vector<char> tabuleiro);
8 inline bool eLegal(int movimento, const vector<char> tabuleiro);
9 int movimentoHumano(const vector<char> tabuleiro, char humano);
10 int movimentoComputador(vector<char> tabuleiro, char computador);
11 char vencedor(const vector<char> tabuleiro);
12 void anuncioVencedor(char vencedor, char computador, char humano);
```

O programa principal (função *main()*) contém toda a estrutura do nosso jogo. Inicialmente, é criado o tabuleiro do jogo (*vector* de caracateres). Após as instruções do jogo serem apresentadas na tela (podem ser observadas na figura 1), é perguntado ao jogador se ele deseja iniciar o jogo (função *pecaHumana()*) e é definido o símbolo do oponente (função *oponente(humano)*).

A cada passagem, é verificado se existe algum vencedor do jogo. Não havendo vencedores, verifica-se a vez (turno) e determina-se o movimento do jogador ou do computador. Finalmente é anunciado o vencedor do jogo ou se houve um empate.

```
1 // programa principal
2 int main() {
3     int movimento;
```

```
4     const int NUM_SQUARES = 9;
5     vector<char> tabuleiro(NUM_SQUARES, VAZIO);
6
7     instrucoes();
8     char humano = pecaHumana();
9     char computador = oponente(humano);
10    char turno = X;
11    mostrarTabuleiro(tabuleiro);
12
13    //enquanto não houver vencedor
14    while (vencedor(tabuleiro) == NINGUEM){
15        if (turno == humano) {
16            movimento = movimentoHumano(tabuleiro, humano);
17            tabuleiro[movimento] = humano;
18        }
19        else {
20            movimento = movimentoComputador(tabuleiro,computador);
21            tabuleiro[movimento] = computador;
22        }
23
24        mostrarTabuleiro(tabuleiro);
25        turno = oponente(turno);
26    }
27
28    anuncioVencedor(vencedor(tabuleiro), computador, humano);
29
30    return 0;
31 }
```

Como pode ser observado no trecho de código anterior, o programa principal contém estritamente o necessário para que o jogo funcione adequadamente. Todos os movimentos e controles são realizados por funções. Esta abordagem permite modularizar o programa e simplificar o entendimento do mesmo.

No código referente a função *instrucoes()* são passadas algumas informações ao jogador. Embora esta função seja chamada apenas uma única vez, é uma boa prática de programação separar trechos de códigos distintos em pequenas funções.

```
1  void instrucoes() {
2     cout << "Bem-vindo ao último confronto homem-máquina: Velha 100 AC." << endl;
3     cout << "— onde cérebro humano é combaterá o processador de silício." << endl << endl;
4
5     cout << "Faça a sua jogada! Introduza um número, 0 - 8. O número" << endl;
6     cout << "corresponde à posição desejada do tabuleiro, tal como ilustrado:" << endl << endl;
7
8     cout << "       0 | 1 | 2" << endl;
9     cout << "       ---------" << endl;
10    cout << "       3 | 4 | 5" << endl;
11    cout << "       ---------" << endl;
12    cout << "       6 | 7 | 8" << endl << endl;
13
14    cout << "Prepare-se, humano. A batalha está prestes a começar." << endl << endl;
15    return;
16 }
```

As funções *simOuNao()* e *numero()* foram escritas para centralizar e simplificar a interação do jogo com jogador humano.

```
1  string simOuNao(string questao) {
2     string resposta;
```

```
    questao += ":[sn]";
    do {
        resposta = readString(questao);
    } while (resposta != "s" && resposta != "n");

    return resposta;
}

int numero(string questao, int alto, int baixo) {
    int numero;
    questao += " (" + numeroToString(baixo) + " - " + numeroToString(alto) + "):";
    do {
        numero = readInt(questao);
    } while (numero> alto || numero < baixo);

    return numero;
}
```

Assim como a função *instrucoes()*, a função *pecaHumana()* é chamada apenas uma vez (no código principal, *main()*), mas como ela tem um objetivo bem específico é melhor separar este código do código principal do jogo:

```
//verifica a peça (X ou O) do jogador humano
char pecaHumana() {
    string primeiro = simOuNao("Você quer o primeiro movimento?");
    if (primeiro == "s") {
        cout << endl << "Pegue o primeiro movimento, você irá precisar mesmo..." << endl;
        return X;
    }
    else {
        cout << endl << "Sua coragem será a sua ruína... Eu vou primeiro!" << endl;
        return O;
    }
}
```

```
//verifica qual o próximo a jogar
char oponente(char peca) {
    if (peca == X) {
        return O;
    }
    else {
        return X;
    }
}
```

O código da função *oponente()* poderia ser substituído pelo trecho:

```
char oponente(char peca) {
  return (peca == X ? O : X);
}
```

Basta lembrar que o operador ? é um operador condicional com a seguinte sintaxe:

```
condição ? resultado 1 : resultado 2;
```

Ou seja, se a condição for verdadeira, pega-se o resultado 1, senão o resultado 2. Veja o exemplo:

```
7==5 ? 4 : 3      // retorna 3, pois 7 não é igual a 5.
7==5+2 ? 4 : 3    // retorna 4, pois 7 é igual a 5+2.
5>3 ? a : b       // retorna o valor de a, pois 5 é maior que 3.
a>b ? a : b       // retorna a se a for maior que b, senão retorna b.
```

Este trecho de código apenas apresenta o tabuleiro e a dica para o jogador:

```
//mostra a situação atual do jogo e a referência do tabuleiro
void mostrarTabuleiro(const vector<char> tabuleiro) {
    cout << "\n\t" << tabuleiro[0] << " | " << tabuleiro[1] << " | " << tabuleiro[2] << "\t\t\t0 | 1 | 2";
    cout << "\n\t" << "_________"  << "\t\t\t_________";
    cout << "\n\t" << tabuleiro[3] << " | " << tabuleiro[4] << " | " << tabuleiro[5] << "\t\t\t3 | 4 | 5";;
    cout << "\n\t" << "_________"  << "\t\t\t_________";
    cout << "\n\t" << tabuleiro[6] << " | " << tabuleiro[7] << " | " << tabuleiro[8] << "\t\t\t6 | 7 | 8";;
    cout << endl << endl;
    return;
}
```

A função *vencedor()* verifica se houve um vencedor no jogo. Esta função recebe uma constante para o tabuleiro (a constante garante que o contéudo do *vector tabuleiro* não será modificado)

Declarar parâmetros de funções como constantes é uma ótima forma de garantir que o conteúdo dos parâmetros não serão modificados. Em códigos maiores (onde várias pessoas podem alterar o código do jogo) é um mecanismo de documentação para os demais programadores, pois está claro que o programador original NÃO quer que o conteúdo dos parâmetros sofram qualquer tipo de alteração (principalmente as alterações intencionais).

Inicialmente é criado um vetor contendo todas as coordenadas vencedores possíveis (vetor *SEQUENCIAS_VENCEDORAS*). Esta estratégia de verificação não permite verificações para tabuleiros maiores (5x5, 7x7, etc), mas simplifica a lógica de verificação se algum jogador ganhou a partida. A verificação é dada pela posição inicial *i* (considerando a primeira sequência, seria o valor 0) com o conteúdo de *i+1* (valor 1) e do conteúdo de *i+1* com o conteúdo de *i+2* (valor 2), se a primeira posição não estiver vazia e todas as posições tiverem o mesmo valor então temos um vencedor.

A próxima verificação diz a respeito a quantidade de espaços em branco disponíveis no tabuleiro. Usando a função *count()* (um dos algorítmos disponíveis para aplicações em *vectors*) contamos quantos espaços estão disponíveis no tabuleiro, não existindo espaços disponíveis, temos um empate no jogo. No caso de ainda haver espaços disponíveis, a função *vencedor()* retorna que ninguém ganhou o jogo ainda.

```
1  //verifica se houve vencedor
2  char vencedor(const vector<char> tabuleiro) {
3      // combinações possíveis de vencedores
4      const int SEQUENCIAS_VENCEDORAS[24] = { 0, 1, 2,
5                                              3, 4, 5,
6                                              6, 7, 8,
7                                              0, 3, 6,
8                                              1, 4, 7,
9                                              2, 5, 8,
10                                             0, 4, 8,
11                                             2, 4, 6 };
12     for(int i = 0; i<24; i+=3) {
13        if( (tabuleiro[ SEQUENCIAS_VENCEDORAS[i] ] != VAZIO) &&
14            (tabuleiro[ SEQUENCIAS_VENCEDORAS[i] ] == tabuleiro[ SEQUENCIAS_VENCEDORAS[i+1] ]) &&
15            (tabuleiro[ SEQUENCIAS_VENCEDORAS[i+1] ] == tabuleiro[ SEQUENCIAS_VENCEDORAS[i+2] ])) {
16           return tabuleiro[ SEQUENCIAS_VENCEDORAS[i] ];
17        }
18
19     }
20     // se ninguém ganhou, checando para ver se ainda existem espaços vazios no tabuleiro
21     if (count(tabuleiro.begin(), tabuleiro.end(), VAZIO) == 0) {
22        return EMPATE;
23     }
24
25     // ninguém ganhou e ainda existem espaços vazios, o jogo não acabou
26     return NINGUEM;
27 }
```

Na função *eLegal()* temos um ótimo exemplo de aplicação de funções *inline*, lembrando que o código das funções *inline* são *copiados* no momento da compilação para o local onde a função é chamada. Embora o código seja simples,

apenas está se comparando se o tabuleiro na posição indicada está disponível para receber uma marcação, o uso de funções *inline* simplifica alterações futuras no código, visto que basta apenas alterar o código da função e o restante do programa continuará inalterado.

```
//verifica se o movimento é válido
inline bool eLegal(int movimento, const vector<char> tabuleiro) {
   return (tabuleiro[movimento] == VAZIO);
}
```

No controle do movimento do jogador humano, dado pela função *movimentoHumano()*, é relativamente simples. Basta apenas verificar se a posição informada pelo jogador está livre para receber o movimento. Enquanto o movimento for considerador ilegal, o jogador é convidado a informar uma posição válida:

```
//retorna o movimento do jogador
int movimentoHumano(const vector<char> tabuleiro, char humano) {
   int movimento = numero("Qual o seu movimento?", (tabuleiro.size() - 1));
   while (!eLegal(movimento, tabuleiro)) {
      cout << endl << "Este espaço está ocupado, humano tolo." << endl;
      movimento = numero("Qual o seu movimento?", (tabuleiro.size() - 1));
   }
   cout << "Bom..." << endl;
   return movimento;
}
```

A função responsável pelo movimento do computador (*movimentoComputador()*) pode ser dividido em três grandes blocos:

1. Verificar se o computador pode ganhar na próxima jogada: a lógica para verificação se o computador pode ganhar o jogo baseia-se em força bruta. Para cada posição vazio do tabuleiro, o computador coloca sua peça e verifica se é possível ganhar (chamando a função *vencedor()*), se não for possível ganhar, a jogada é desfeita para que o tabuleiro volte a situação original.

```
//retorna o movimento do computador
int movimentoComputador(vector<char> tabuleiro, char computador) {
   cout << "Eu escolho o espaço ";

   // se o computador pode ganhar na próxima jogada...
   for(int movimento = 0; movimento < tabuleiro.size(); ++movimento) {
      if (eLegal(movimento, tabuleiro)) {

         tabuleiro[movimento] = computador;

         if(vencedor(tabuleiro) == computador) {
            cout << movimento << endl;
            return movimento;
         }
          // movimento verificado, tenho que desfaze-lo
         tabuleiro[movimento] = VAZIO;
      }
   }
```

2. Impedir uma possível vitória do jogador humano: se o computador não pode ganhar a partida, ele deve impedir uma possível vitória do seu oponente. Novamente, utilizando técnica de força bruta, o computador verifica cada posição vazia e simula a jogada do oponente. Caso o oponente consiga a vitória em alguma tentativa, o computador escolhe a posição para bloquear a jogada. Caso o oponente não vença, o tabuleiro é retornado ao conteúdo original.

```
   // se o jogador humano for ganhar na próxima jogada, é necessário bloquea-lo
   char humano = oponente(computador);
   for(int movimento = 0; movimento < tabuleiro.size(); ++movimento)
   {
      if (eLegal(movimento, tabuleiro)) {

```

```
            tabuleiro[movimento] = humano;

            if (vencedor(tabuleiro) == humano) {
                cout << movimento << endl;
                return movimento;
            }
            // movimento verificado, tenho que desfaze-lo
            tabuleiro[movimento] = VAZIO;
        }
    }
```

3. Escolher a melhor posição para se jogar: bom, o computador não tem nenhuma jogada vencedora e também não foi necessário bloquear o oponente. Neste caso é necessário escolher uma posição para jogar. Poderia-se utilizar técnicas baseada em estudo de árvores binárias para *prever* a melhor jogada, mas o nosso código é mais simplificado neste caso. Cria-se um vetor de *boas* jogadas em sequência e verifica no tabuleiro qual a primeira posição livre correspondente a jogada. Por exemplo: se o meio do tabuleiro (posição 4) estiver vazio, o computador irá *escolher* esta posição, senão verifica todos os cantos (posições 0, 2, 6 e 8) para depois escolher as posições mais centrais (1, 3, 5 e 7):

```
    // a melhor sequência de jogadas
    const int MELHORES_MOVIMENTOS[]={4, 0, 2, 6, 8, 1, 3, 5, 7};
    // se não posso ganhar nesta jogada, tento escolher a melhor jogada possível
    for(int i = 0; i < tabuleiro.size(); ++i) {
        int movimento = MELHORES_MOVIMENTOS[i];
        if (eLegal(movimento, tabuleiro)){
            cout << movimento << endl;
            return movimento ;
        }
    }
}
```

E finalmente, a função *anuncioVencedor()* anuncia o ganhador do jogo. Como as peças (símbolos X ou O) não são fixos para os jogadores, é necessário saber quem ganhou o jogo e qual símbolo cada jogador está utilizando para fazer o anúncio correto:

```
void anuncioVencedor(char vencedor, char computador, char humano) {
    if(vencedor == computador ){
        cout << vencedor << " ganhou!" << endl;
        cout << "Como eu previ, humano, eu sou vitorioso mais uma vez -- a prova" << endl;
        cout << "que os computadores são superiores aos seres humanos em todos os aspectos." << endl;
    }

    else if (vencedor == humano) {
        cout << vencedor << " ganhou!" << endl;
        cout << "Não, não! Não pode ser! De alguma forma, você me enganou, humano." << endl;
        cout << "Mas esta será a última vez, eu prometo!!" << endl;
    }
    else{
        cout << "Temos um empate!" << endl;
        cout << "Você tem sorte, humano, e de alguma forma conseguiu empatar comigo." << endl;
        cout << "Celebre... pois é o melhor que irá conseguir de mim." << endl;;
    }
    return;
}
```

6 Animação com sprites e vetores de *strings*

Um *sprite* é um elemento gráfico que é desenhado contra um cenário de fundo num jogo. Os *sprites* foram inventados originalmente como um método rápido de animação de várias imagens agrupadas numa tela, em jogos de computador bidimensionais.

Uma animação de um jogo em duas dimensões, por exemplo, é representada por uma sequência de *sprites* sendo exibidos em sucessão (de forma temporizada). Por exemplo, uma caminhada pode ser representada através de uma alternância entre um *sprite*, que exibe o personagem pisando com o pé direito, seguido por outro *sprite* no qual o mesmo personagem esteja pisando com o pé esquerdo, dando uma sensação de que ele está caminhando.

A figura 2 apresenta um exemplo de *sprites* utilizados para simular um inventor mostrando língua. Bastaria encaixar a sequência de figuras para obter tal efeito.

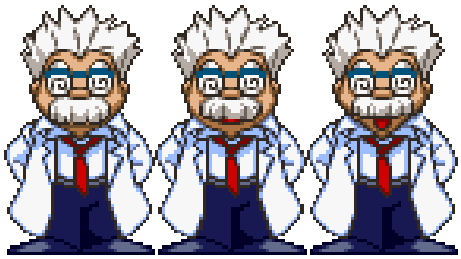


Figura 2: Um exemplo de sprite com o inventor maluco.

Como ainda não estamos trabalhando com efeitos gráficos (nem é o objetivo deste material), temos que *desenhar no braço* as nossas animações. A figura 3 apresenta um exemplo de várias animações obtidas com uso de vetores (resultado do programa 5).

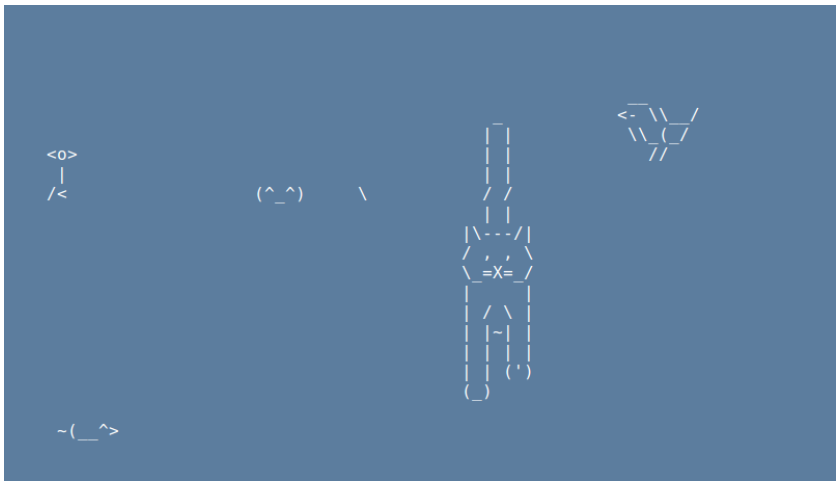


Figura 3: O que o programa 5 mostra na tela.

Programa 5: Brincando com sprites e animação.

```
1  //sprites (animação com vetores)
2  //programa_005.cpp
3  #include "biblaureano.h"
4
5  const int SPRITES=4;
6
7  int main(){
8
9     int i = 0;
```

```
10    int m = 0;
11    desligaCursor(true);
12
13    vector <string> macarena = retornaArquivoSprites("macarena.txt");
14    vector <string> passarinho = retornaArquivoSprites("passarinho.txt");
15    vector <string> gatinho = retornaArquivoSprites("gatinho.txt");
16
17    const string olhos[SPRITES] = {"(-_-)","(^_^)", "(-_-)", "(v_v)"};
18    const string ampulheta[SPRITES] = {"-","\\", "|","/"};
19    const string ratinho[SPRITES]={"~(__^>", " ~(__^>", "  ~(__^>", "   ~(__^>"};
20
21    while(true){
22       imprimeSprite(olhos[i],25,10);
23       imprimeSprite(macarena[m],5,8);
24       imprimeSprite(ampulheta[i],35,10);
25       imprimeSprite(ratinho[i],5,22);
26       imprimeSprite(passarinho[i],60,5);
27       imprimeSprite(gatinho[i],45,6);
28       espera(50);
29       limparTela();
30
31       if(kbhit()) break;
32       if( ++i == SPRITES ) i = 0;
33       if( ++m == macarena.size() ) m =0;
34    }
35    cout << "Game Over" << endl;
36    return 0;
37 }
```

6.1 Entendendo o programa

Inicialmente são declaradas duas variáveis (*i* para controle das animações com 4 imagens e *m* para controle da animação com 9 imagens) e o cursor é desabilitado para evitar o efeito piscante na tela.

```
1    ...
2    int i = 0;
3    int m = 0;
4    desligaCursor(true);
5    ...
```

São criados o vetores/*vectors* que conterão nossas animações. As primeiras animações são carregadas de arquivos utilizando a função *retornaArquivoSprites()* (disponível na biblioteca de funções). A função retorna um *vector* de *strings* contendo todas as imagens geradas no arquivo.

O protótipo da função *retornaArquivoSprites()* é:

```
1 vector<string> retornaArquivoSprites(string nomeArquivo, string separador="*????????*");
```

Repare que o parâmetro *string separador* é opcional. O separador é utilizado para separar uma imagem da outra dentro do mesmo arquivo, caso você opte por utilizar outra sequência de caracteres como separador de imagens será necessário informa-la no momento da chamada da função. O formato do arquivo texto é parecido com o abaixo:

```
(-_-)
*????????*
(^_^)
*????????*
(-_-)
*????????*
(v_v)
*????????*
```

O arquivo acima poderia ser utilizado para representar a animação dos olhos. No exemplo são 4 animações possíveis.

As animações dos olhos, ampulheta e o ratinho foram declaradas diretamente como vetores por serem animações simples (apenas uma única linha).

```
1  ...
2  vector <string> macarena = retornaArquivoSprites("macarena.txt");
3  vector <string> passarinho = retornaArquivoSprites("passarinho.txt");
4  vector <string> gatinho = retornaArquivoSprites("gatinho.txt");
5
6  const string olhos[SPRITES] = {"(-_-)","(^_^)", "(-_-)", "(v_v)"};
7  const string ampulheta[SPRITES] = {"-","\\", "|","/"};
8  const string ratinho[SPRITES]={"~(__^>", " ~(__^>", "  ~(__^>", "   ~(__^>"};
9  ...
```

As animações consistem de um *loop* infinito que será interrompido quando o usuário pressionar qualquer tecla (função *kbhit()*). A impressão da imagem é obtida utilizando a função *imprimeSprite()*, passando a posição do vetor e as coordenadas da imagem. Na sequência as variáveis *i* e *m* são incrementadas e comparadas com o tamanho de imagens, caso o tamanho máximo tenha sido alcançado, a variável recomeça com o valor 0 (que indica a primeira imagem da sequência).

```
1  ...
2  while(true){
3      imprimeSprite(olhos[i],25,10);
4      imprimeSprite(macarena[m],5,8);
5      imprimeSprite(ampulheta[i],35,10);
6      imprimeSprite(ratinho[i],5,22);
7      imprimeSprite(passarinho[i],60,5);
8      imprimeSprite(gatinho[i],45,6);
9      espera(50);
10     limparTela();
11
12     if(kbhit()) break;
13     if( ++i == SPRITES ) i = 0;
14     if( ++m == macarena.size() ) m =0;
15 }
16 ...
```

Lembre-se do operador de pré-incremento *++variável*, neste caso a variável primeiro é incrementada e depois estará disponível para ser utilizada (com um novo valor). Exemplo:

```
int i = 0;
++i; //i vale 1
int m = ++i; //m e i estão com o valor 2
int n = i++; //n está com o valor 2 e i com valor 3, pois foi utilizado o operador de pós-incremento
```

7 Animação com sprites e *vectors* de Imagens

O programa 5 tem o inconveniente de sempre ser necessário chamar a função *limparTela()*, o que em alguns casos não se justifica, pois queremos apenas limpar a área da imagem e não a tela toda. O programa 6 demonstra como resolver este problema. Em vez de criar vetores/*vectors* de *strings* estamos criando *vectors* de imagens (classe *Imagem*, disponível na biblioteca).

Programa 6: Brincando com sprites e animação - trabalhando com *vectors* de Imagens.

```
//sprites (animação com vetores)
//criando vectors da classe Imagem
//programa_006.cpp
#include "biblaureano.h"

const int SPRITES=4;

int main(){
    int i = 0;
    int m = 0;
    desligaCursor(true);

    vector <string> macarena = retornaArquivoSprites("macarena.txt");

    const string olhos[SPRITES] = {"(-_-)","(^_^)", "(--)", "(v_v)"};
    const string ampulheta[SPRITES] = {"-","\\", "|\n","/\n"};
    const string ratinho[SPRITES]={"~(__^>", " ~(__^>", "  ~(__^>", "   ~(__^>"};

    //criando vector de Imagem
    vector <Imagem> imagemOlhos = criaImagens(olhos, 25,10, SPRITES );
    vector <Imagem> imagemAmpulheta = criaImagens(ampulheta, 35,10, SPRITES );

    //passando vetor
    vector <Imagem> imagemRatinho = criaImagens(ratinho, 5,22, SPRITES );

    //passando vector
    vector <Imagem> imagemMacarena = criaImagens(macarena,5,8);

    //passando vector - retorno da função retornaArquivoSprites
    vector <Imagem> imagemGatinho = criaImagens(retornaArquivoSprites("gatinho.txt"), 45,6);

    //passando o nome do arquivo
    vector <Imagem> imagemPassarinho = criaImagens("passarinho.txt", 60,5);

    while(true) {
        imagemOlhos[i].imprime();
        imagemAmpulheta[i].imprime();
        imagemRatinho[i].imprime();
        imagemPassarinho[i].imprime();
        imagemGatinho[i].imprime();
        imagemMacarena[m].imprime();
        espera(50);
        imagemOlhos[i].limpa();
        imagemAmpulheta[i].limpa();
```

```
45          imagemRatinho[i].limpa();
46          imagemPassarinho[i].limpa();
47          imagemGatinho[i].limpa();
48          imagemMacarena[m].limpa();
49
50          if(kbhit()) break;
51          if( ++i == SPRITES ) i = 0;
52          if( ++m == macarena.size() ) m =0;
53      }
54      cout << "Game Over" << endl;
55      return 0;
56  }
```

7.1 Entendendo o programa

A grande diferença do programa 6 em relação ao programa 5 reside na chamada da função *criaImagens()* (disponível na biblioteca) e a criação de um *vector* de *Imagem*. A função pode receber um vetor de *strings*, um *vector* de *strings* ou o nome do arquivo que contém as imagens e as coordenadas iniciais de cada imagem.

```
1   ...
2   vector <string> macarena = retornaArquivoSprites("macarena.txt");
3
4   const string olhos[SPRITES] = {"(-_-)","(^_^)", "(--)", "(v_v)"};
5   const string ampulheta[SPRITES] = {"-","\\", "|\n","/\n"};
6   const string ratinho[SPRITES]={"~(__^>", " ~(__^>", "  ~(__^>", "   ~(__^>"};
7
8   //criando vector de Imagem
9   vector <Imagem> imagemOlhos = criaImagens(olhos, 25,10, SPRITES );
10  vector <Imagem> imagemAmpulheta = criaImagens(ampulheta, 35,10, SPRITES );
11
12  //passando vetor
13  vector <Imagem> imagemRatinho = criaImagens(ratinho, 5,22, SPRITES );
14
15  //passando vector
16  vector <Imagem> imagemMacarena = criaImagens(macarena,5,8);
17
18  //passando vector - retorno da função retornaArquivoSprites
19  vector <Imagem> imagemGatinho = criaImagens(retornaArquivoSprites("gatinho.txt"), 45,6);
20
21  //passando o nome do arquivo
22  vector <Imagem> imagemPassarinho = criaImagens("passarinho.txt", 60,5);
23  ...
```

Lembrando que a Imagem contém várias funções, entre eles a capacidade de impressão (função *imprime()*) e limpeza da imagem na tela (função *limpa()*), basta realizar a chamada da função acrescentando o . (ponto) e nome da função depois do nome do *vector* e a posição do mesmo. A vantagem desta abordagem é poder imprimir/apagar a imagem na tela de forma individual, sem interferir nos demais elementos que compõem a tela como um todo.

```
1       ...
2       imagemOlhos[i].imprime();
3       imagemAmpulheta[i].imprime();
4       imagemRatinho[i].imprime();
5       imagemPassarinho[i].imprime();
6       imagemGatinho[i].imprime();
7       imagemMacarena[m].imprime();
8       espera(50);
9       imagemOlhos[i].limpa();
10      imagemAmpulheta[i].limpa();
11      imagemRatinho[i].limpa();
12      imagemPassarinho[i].limpa();
13      imagemGatinho[i].limpa();
14      imagemMacarena[m].limpa();
15      ...
```

8 Atividade

1. Alterar o jogo da velha para que seja possível jogar contra o computador e contra outro oponente humano.
2. Implemente o jogo de *Mastermind*, no Brasil também conhecido como Senha. Um jogo de Mastermind tem pinos de seis cores diferentes, aleatórias, exceto preto e branco. Os pinos pretos e brancos são menores. Há quatro buracos grandes em cada fileira, em 10 fileiras, uma abaixo da outra. E ao lado delas, um quadrado menor, com quatro buracos menores, dois em cima de dois. Uma fileira, que seria a décima primeira, tem um defletor que esconde seus buracos. O desafiador faz uma combinação com quatro pinos coloridos, sem repetir as cores de cada pino, e as põe na décima primeira fileira e levanta o defletor, escondendo a senha. Então, o desafiado tenta adivinhar a senha, pondo quatro pinos que ele acha que são a senha na primeira fileira, e o desafiador põe os pinos pretos e brancos no quadrado menor ao lado. A regra dos pinos pretos e brancos são essas: o branco significa que há uma cor certa mas no lugar errado, o preto significa que há uma cor certa no lugar certo, e nenhum pino significa que uma das cores não é contida na senha. O desafiado vai tentando adivinhar, se guiando pelos pinos pretos e brancos. Se o desafiado não acertar até a 10ª fileira, o desafiador fecha o defletor e revela a senha, mas se adivinhar, o desafiador põe quatro pinos pretos e revela a senha.

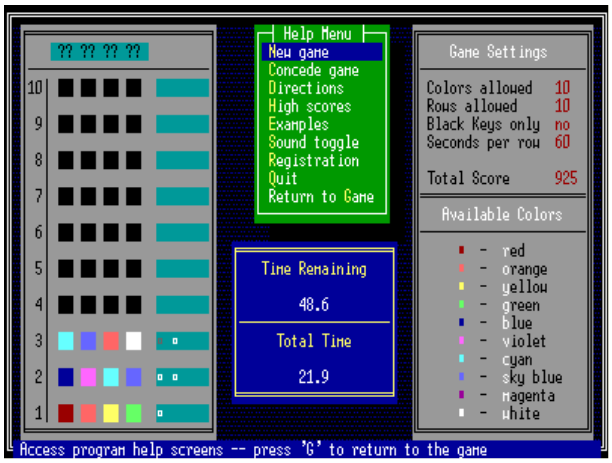


Figura 4: Exemplo do jogo - retirado de *http://www.textmodegames.com/download/codebreaker.html*.