Nada se obtém sem esforço; e tudo se pode conseguir com ele.

Ralph Waldo Emerson

1 Trabalhando com referências - relembrar é viver!!!

Uma referência fornece um outro nome para uma variável. Qualquer operação de alteração numa referência também ocorrerá na variável referida.Você pode pensar em uma referência como um apelido para um nome de outra variável.

O programa 1 declara variável de referência *meuPlacarTambem* para a variável *meuPlacar*.

Programa 1: *Apelidos* para variáveis.

```
1  // referências - começando o abacaxi
2  // programa_001.cpp
3  #include "biblaureano.h"
4
5  int main(){
6      int meuPlacar = 100;
7      int& meuPlacarTambem = meuPlacar;
8
9      cout << "meuPlacar = " << meuPlacar << endl;
10     cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
11
12     cout << "Hackeando o jogo... subindo o meu placar..." << endl;
13
14     meuPlacar += 500;
15
16     cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
17
18     cout << "Sacaneando vocês... estou roubando no placar... " << endl;
19     meuPlacarTambem += 1000;
20
21     cout << "meuPlacar = " << meuPlacar << endl;
22
23     cout << "Game over!!!" << endl;
24     return 0;
25 }
```

1.1 Entendendo o programa

Inicialmente, a variável *meuPlacar* é declarada normalmente (como qualquer outra variável que já utilizamos em vários outros programas):

```
1      ...
2      int meuPlacar = 100;
3      ...
```

Na sequência, é criada a variável *meuPlacarTambem* que é uma referência para a variável *meuPlacar*. Repare que existe um & (letra E comercial) antes do nome da variável:

```
1      ...
2      int& meuPlacarTambem = meuPlacar;
3      ...
```

Cuidado: sempre que você declarar uma variável referência, você deve atribuir algum valor a ela (nome da variável que será referenciada. Uma chamada a:

```
1  int& meuPlacarTambem;
```

Causará um erro de compilação, pois a variável não foi inicializada.

Com a referência criada, as duas variáveis terão o mesmo valor na memória:

```
1  ...
2  cout << "meuPlacar = " << meuPlacar << endl;
3  cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
4  ...
```

A partir deste momento, qualquer alteração feita na variável original (*meuPlacar*) também refletirá na variável apelido (*meuPlacarTambem*:

```
1  ...
2  cout << "Hackeando o jogo... subindo o meu placar..." << endl;
3
4  meuPlacar += 500;
5
6  cout << "meuPlacarTambem = " << meuPlacarTambem << endl;
7  ...
```

E o inverso também ocorre, sempre que ocorrer qualquer alteração realizada na variável referência (*meuPlacarTambem*) a variável original (*meuPlacar*) também será alterada:

```
1  ...
2  cout << "Sacaneando vocês... estou roubando no placar... " << endl;
3
4  meuPlacarTambem += 1000;
5
6  cout << "meuPlacar = " << meuPlacar << endl;
7  ...
```

Lembre-se: pense sempre na variável referência como um *apelido* para uma variável normal.

2 Referências para funções - alterando os parâmetros de entrada

Agora que você já entendeu como funcionam as referências, você deve estar se perguntando: *Quando irei utilizá-los?* ou *Qual a utilidade?*.

Bem, as referências são úteis quando você está passando as variáveis para as funções, pois quando você passa uma variável (parâmetro de entrada), a função recebe uma *cópia* da variável. Isto significa que a variável original que você passou não pode ser modificada.

Claro, isso pode ser exatamente o que você deseja, pois a função mantém a variável de argumento seguro e imutável. Mas outras vezes você pode querer alterar uma variável argumento dentro da função e você pode fazer isso usando referências. O programa 2 demonstra uma possibilidade de uso para referências de variáveis nos argumentos/parâmetros de entrada de uma função:

Programa 2: Parâmetros como referências.

```
// trocando variáveis
// programa_002.cpp
#include "biblaureano.h"

void naoFuncionaTroca( int a, int b);
void aquiFuncionaTroca( int& a, int& b);

int main(){
   int umValor, outroValor;

   umValor = 30072010;
   outroValor = 27121975;

   cout << "Valores antes da troca.." << endl;
   cout << "umValor = " << umValor << endl;
   cout << "outroValor = " << outroValor << endl;

   cout << "Trocando as variáveis..." << endl;
   naoFuncionaTroca( umValor, outroValor );

   cout << "umValor = " << umValor << endl;
   cout << "outroValor = " << outroValor << endl;

   cout << "Trocando as variáveis... agora vai!!!" << endl;
   aquiFuncionaTroca( umValor, outroValor );

   cout << "umValor = " << umValor << endl;
   cout << "outroValor = " << outroValor << endl;

   cout << "Game over!!!" << endl;
   return 0;
}

void aquiFuncionaTroca( int& a, int& b){
   int temporario = a;
   a = b;
   b = temporario;
   return;
}

void naoFuncionaTroca( int a, int b){
   int temporario = a;
   a = b;
   b = temporario;
   return;
}
```

2.1 Entendendo o programa

O objetivo do programa é simples, realizar a troca (*swap*,permuta) dos valores entre duas variáveis. Inicialmente são declarado os protótipos das funções que realizam a troca, repare que os argumentos de entrada da função *aquiFuncionaTroca* são declarados como referência.

```
void naoFuncionaTroca( int a, int b);
void aquiFuncionaTroca( int& a, int& b);
```

Após a declaração das variáveis, é chamada a função *naoFuncionaTroca* e mostrado o resultado da troca:

```
    ...
    cout << "Trocando as variáveis..." << endl;
    naoFuncionaTroca( umValor, outroValor );

    cout << "umValor = " << umValor << endl;
    cout << "outroValor = " << outroValor << endl;
    ...
```

Neste caso, o seguinte código será executado:

```
void naoFuncionaTroca( int a, int b){
    int temporario = a;
    a = b;
    b = temporario;
    return;
}
```

Na sequência, o programa tenta novamente realizar a troca das variáveis. Desta vez é chamada a função *aquiFuncionaTroca* e novamente é impresso o resultado da permuta:

```
    ...
    cout << "Trocando as variáveis... agora vai!!!" << endl;
    aquiFuncionaTroca( umValor, outroValor );

    cout << "umValor = " << umValor << endl;
    cout << "outroValor = " << outroValor << endl;
    ...
```

Neste caso, o seguinte código será executado:

```
void aquiFuncionaTroca( int& a, int& b){
    int temporario = a;
    a = b;
    b = temporario;
    return;
}
```

2.2 O que ocorreu ?

Quando passamos argumentos para um função sem utilizar referência, em termos de programação dizemos que estamos passando uma variável por *valor*. Neste caso é criada uma *cópia* da variável em memória e qualquer alteração será realizada na cópia e não na variável original.

Quando passamos argumentos para uma função utilizando uma referência, em termos de programação dizemos que estamos passando uma variável por *referência*. Neste caso não é criada nenhuma cópia da variável em memória e qualquer alteração será realizada na variável original.

Estes recursos estão disponíveis em qualquer linguagem de programação moderna.

3 Vamos entender um pouco como a memória funciona

O operador & é um operador unário que devolve endereço na memória de seu operando. Observe o programa 3:

Programa 3: Verificando os endereços de memória utilizados.

```
// trocando variáveis
```

```
2  // programa_003.cpp
3  #include "biblaureano.h"
4
5  void mostraValores( int a, int b);
6  void mostraValoresDeNovo( int& a, int& b);
7
8  int main(){
9     int umValor, outroValor;
10
11    cout << "Endereço de umValor:" << &umValor << endl;
12    cout << "Endereço de outroValor:" << &outroValor << endl;
13
14    umValor = readInt("Entre com um valor:");
15    outroValor = readInt("Entre com outro valor:");
16
17    //chamando a função e passando variáveis por valor
18    mostraValores(umValor, outroValor);
19
20    //chamando a função e passando variáveis por referência
21    mostraValoresDeNovo(umValor, outroValor);
22
23    return 0;
24 }
25
26 void mostraValores( int a, int b){
27    cout << "A:" << a << endl;
28    cout << "Endereço de A:" << &a << endl;
29    cout << "B:" << b << endl;
30    cout << "Endereço de B:" << &b << endl;
31    return;
32 }
33
34 void mostraValoresDeNovo( int& a, int& b){
35    cout << "A:" << a << endl;
36    cout << "Endereço de A:" << &a << endl;
37    cout << "B:" << b << endl;
38    cout << "Endereço de B:" << &b << endl;
39    return;
40 }
```

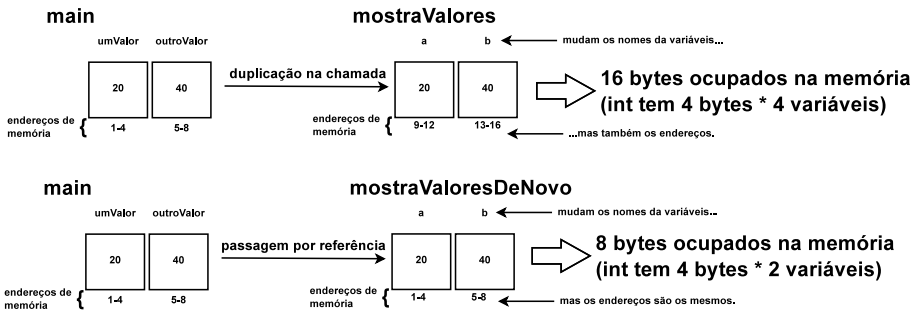

Figura 1: O que acontece na memória.

Como pode ser observado na figura 1, o programa 3 define duas funções que fazem praticamente a mesma coisa, a única diferença é que uma função está definida para trabalhar com as referências e outra não:

```
1  void mostraValores( int a, int b);
2  void mostraValoresDeNovo( int& a, int& b);
```

Logo na sequência, utilizamos o operador & para descobrir o endereço de memória das variáveis *umValor* e *outroValor*:

```
1     ...
```

```
2     int umValor, outroValor;
3
4     cout << "Endereço de umValor:" << &umValor << endl;
5     cout << "Endereço de outroValor:" << &outroValor << endl;
6     ...
```

Após a leitura dos dados, ocorre uma chamada a função *mostraValores* (que não está setado para receber os dados por referência, ou seja, recebe os dados por valor):

```
1     ...
2     //chamando a função e passando variáveis por valor
3     mostraValores(umValor, outroValor);
4     ...
```

O conteúdo da variável *umValor* é *copiado* para a variável *a* e o conteúdo da variável *outroValor* é *copiado* para a variável *b*. A função *mostraValores* também mostra os endereços de memória das variáveis *a* e *b*. Como os valores foram copiados, houve uma *duplicação* dos valores na memória e logo *a* e *b* ocupam áreas de memória diferentes de *umValor* e *outroValor*:

```
1 void mostraValores( int a, int b){
2     cout << "A:" << a << endl;
3     cout << "Endereço de A:" << &a << endl;
4     cout << "B:" << b << endl;
5     cout << "Endereço de B:" << &b << endl;
6     return;
7 }
```

Mas com a chamada da função *mostraValoresDeNovo* (que está setado para receber os dados por referência), não existe a duplicação dos dados, pois as variáveis *a* e *b* *compartilham* o mesmo endereço de memória das variáveis *umValor* e *outroValor*:

```
1     ...
2     //chamando a função e passando variáveis por referência
3     mostraValoresDeNovo(umValor, outroValor);
4     ...
```

Como as variáveis *compartilham* a mesma memória, os endereços de memória de *a* e *b* serão os mesmos de *umValor* e *outroValor*:

```
1 void mostraValoresDeNovo( int& a, int& b){
2     cout << "A:" << a << endl;
3     cout << "Endereço de A:" << &a << endl;
4     cout << "B:" << b << endl;
5     cout << "Endereço de B:" << &a << endl;
6     return;
7 }
```

4 Referências em argumentos de funções - aumento de eficiência

Ao passar uma variável por valor, você cria uma sobrecarga no seu programa pois o valor da variável é copiada - e portanto duplicada na memória - para a variável de argumento de entrada. Quando estamos falando de variáveis simples, construído em tipos, como um *int* ou *float*, a sobrecarga é desprezível.

Mas um objeto grande, como aquela que representa todo um mundo 3D, pode ser oneroso o processo de cópia para o seu jogo ao se passar por valor. Mas passar o objeto por referência é eficiente pois não ocorre a cópia da variável em memória. E para evitar problemas como alterações indesejadas, basta declarar o parâmetro de entrada como uma referência constante (que não pode ser modificada). O programa 4 implementa este conceito:

Programa 4: Parâmetros constantes como referências.

```
// uma outra versão para o programa de inventário
// programa_004.cpp
#include "biblaureano.h"

void mostraInventarioI(const vector<string>& vetor);
void mostraInventarioV(const vector<string>& vetor);

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

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

   cout << "Endereço de vetor:" << &inventario << endl;
   mostraInventarioI(inventario);
   mostraInventarioV(inventario);

   cout << "Game over!!!" << endl;
   return 0;
}

void mostraInventarioI(const vector<string>& vetor){
   cout << "Função utilizando iterador.";
   cout << "Seu inventário tem os seguintes itens:" << endl;
   cout << "Endereço de vetor:" << &vetor << endl;
   for( vector<string>::const_iterator meuIterador = vetor.begin();
        meuIterador != vetor.end();
        ++meuIterador)
   {
      cout << "\t" << *meuIterador << endl;
   }
   return;
}

void mostraInventarioV(const vector<string>& vetor){
   cout << "Função utilizando notação de vetor.";
   cout << "Seu inventário tem os seguintes itens:" << endl;
   cout << "Endereço de vetor:" << &vetor << endl;
   for( int i =0; i<vetor.size(); ++i)
   {
      cout << "\t" << vetor[i] << endl;
   }
   return;
}
```

4.1 Entendendo o programa - é tão simples

A única novidade neste programa está na declaração do protótipo da função, pois além de incluirmos a referência estamos dizendo que o valor será constante:

```
void mostraInventarioI(const vector<string>& vetor);
void mostraInventarioV(const vector<string>& vetor);
```

5 A decisão - como informar argumento para uma função

A grande decisão da sua vida deve ser tomada!!! Passar variáveis para uma função como valor, referência ou como referência constante ? Como orientação podemos utilizar as seguintes regras:

Por valor : passe uma variável por valor quando esta for de um tipo básico da linguagem, como *int* ou *char*. Variáveis deste tipo são tão pequenas que a passagem por referência não resulta em qualquer ganho de eficiência. Você também deve passar por valor quando quiser que o computador faça uma cópia da variável. Você pode desejar utilizar uma cópia, se você pretende alterar um parâmetro em uma função, mas você não deseja que a variável de argumento original seja afetada.

Por referência constante : passe uma referência constante quando você quer passar um valor de forma eficiente mas que você não deseja que a função altere os valores.

Por referência : passe uma referência somente quando você quiser alterar o valor da variável de argumento. No entanto, você deve tentar evitar a mudança de variáveis argumento sempre que possível.

6 Funções que retornam referências - outra forma de poupar memória

Assim como passar argumentos por valor causa sobrecarga no uso da memória, funções que tornam valores também causam sobrecarga. Naturalmente que para tipos básicos ou variáveis de pequeno tamanho esta sobrecarga é desprezível. Mas existem situações que pode ser interessante retornar uma referência. O programa 5 simula uma situação de criação de imagens, que por sua vez são compostos por pontos (*pixels*) e no momento da busca das imagens, a imagem é retornada como uma referência:

Programa 5: Funções retornando referências.

```
1  // retorna referências
2  // programa_005.cpp
3  #include "biblaureano.h"
4
5  //definindo a cor do ponto
6  typedef struct{
7     int cor;
8     int x,y;
9  } PIXEL;
10
11 typedef struct{
12    //uma imagem é composta de vários pontos
13    vector<PIXEL> pixel;
14 } IMAGEM;
15
16 void criaImagem( vector<IMAGEM>& imagens );
17
18 //atenção na declaração da função
19 IMAGEM& retornaImagem( vector<IMAGEM>& imagens, int i);
20
21 void imprimeImagem( IMAGEM & imagem);
22
23 #define QTD_IMAGENS 5
24 #define QTD_PONTOS 50
25
26 int main(){
27    desligaCursor(true);
28
29    //declaração do vetor
30    vector<IMAGEM> imagens;
31
32    //passa o vetor por referência
33    criaImagem( imagens );
34
35    //imprimindo as imagens na tela
36    for( int i = 0; i<QTD_IMAGENS; ++i)   {
37       //obtém o retorno da retornaImagem
38       //a variável que recebe também deve ser declarada como referência
39       IMAGEM& imagem = retornaImagem( imagens, i);
40       //cout << "Endereço:" << &Imagens[i] << endl;
```

```
41          //cout << "Endereço:" << &Imagem << endl;
42          //passa o retorno para a função de impressão
43          imprimeImagem( imagem );
44          espera(200);
45          limparTela();
46
47          //por ser uma referência, eu posso alterar a cor da imagem
48          for( int k = 0; k<QTD_PONTOS; ++k)
49          {
50              imagem.pixel[k].cor = WHITE;
51          }
52      }
53
54      cout << "Nesta nova impressão das imagens, a cor será sempre branca" << endl;
55      espera(300);
56
57      //uma outra forma de fazer a mesma coisa
58      for( int i = 0; i<QTD_IMAGENS; ++i){
59          //chama a função de impressão e passa o retorno
60          //da função retornaImagem
61          limparTela();
62          imprimeImagem( retornaImagem( imagens, i));
63          espera(100);
64      }
65
66      cout << "Game over!!!" << endl;
67      return 0;
68  }
69
70  void criaImagem( vector<IMAGEM>& imagens ){
71      //simulação de criação de imagens e pixels;
72      for( int i = 0; i<QTD_IMAGENS; ++i)    {
73          IMAGEM imagem;
74          for( int k = 0; k<QTD_PONTOS; ++k)        {
75              //monta a imagem pixel por pixel
76              PIXEL ponto;
77              ponto.cor = randomico(1,7);
78              ponto.x = randomico(1,20);
79              ponto.y = randomico(1,20);
80              //coloca o pixel dentro da imagem
81              imagem.pixel.push_back(ponto);
82          }
83          //colocar a imagem no vetor
84          imagens.push_back(imagem);
85      }
86      return;
87  }
88
89  IMAGEM& retornaImagem( vector<IMAGEM>& imagens, int i){
90      return imagens[i];
91  }
92
93  void imprimeImagem( IMAGEM & imagem){
94      for( int i = 0; i<QTD_PONTOS; ++i){
95          gotoXY( imagem.pixel[i].x, imagem.pixel[i].y);
96          mudaCor((COR)imagem.pixel[i].cor,(COR)imagem.pixel[i].cor);
97          cout << " " << endl;
98      }
99      limpaEfeito();
100 }
```

Não é possível retornar referências de variáveis locais a função. Quando uma função termina sua execução, todas suas variáveis locais são excluídas da memória, logo a referência torna-se inválida. Veja um exemplo:

```
1 string & retornaFrase(){
2    local frase = "Esta frase jamais será retornada, pois a variável é local e
3                   não é possível referenciá-la fora da função!";
4    return frase;
5 }
```

6.1 Entendendo o programa

Neste programa são duas novidades, uma no momento da declaração do protótipo, quando colocamos o caractere & antes do nome da função:

```
1 //atenção na declaração da função
2 IMAGEM& retornaImagem( vector<IMAGEM>& imagens, int i);
```

A segunda novidade diz respeito no uso da referência retornada. Caso você deseje alterar o conteúdo na memória, a variável que recebe a referência deve ser declarada como tal.

```
1       ...
2       //obtém o retorno da retornaImagem
3       //a variável que recebe também deve ser declarada como referência
4       IMAGEM& imagem = retornaImagem( imagens, i);
5       //passa o retorno para a função de impressão
6       imprimeImagem( imagem );
7       ...
```

Caso você não faça isto, a próxima parte do programa não funcionaria adequadamente, pois você não estaria alterando a imagem na memória (neste caso a imagem contida dentro do nosso vetor dinâmico).

```
1       ...
2       //por ser uma referência, eu posso alterar a cor da imagem
3       for( int k = 0; k<QTD_PONTOS; ++k){
4          imagem.pixel[k].cor = WHITE;
5       }
6       ...
```

Embora não seja uma novidade, é bom ressaltar que é possível passar o retorno de uma função como parâmetro de entrada para outra função:

```
1       ...
2       //chama a função de impressão e passa o retorno
3       //da função retornaImagem
4       limparTela();
5       imprimeImagem( retornaImagem( imagens, i));
6       ...
```

7 Jogo da espaçonave - uma aplicação com referências

Programa 6: ASG - *Another Spaceship game.*

```
#include "biblaureano.h"

void movimentaInimigos(vector <Imagem> & inimigos);
void movimentaTiros(vector <Imagem> & tiros, vector<Imagem> &inimigos, int &pontos);
bool colisaoInimigos(const vector <Imagem> inimigos, Imagem inimigo);
void criaInimigos(vector <Imagem> & inimigos, int qtdInimigos);
void criaTirosInimigos(vector <Imagem> & tirosInimigos, const vector <Imagem> & inimigos, int & vida, const Imagem & nave
    );

const int X=120,
          Y=30,
          VELOCIDADE_NAVE=8,
          VELOCIDADE_TIRO=3,
          MAX_TIROS=6,
          MAX_TIROS_INIMIGO=12,
          VIDA=5;

int main(){

   int qtdInimigos = readInt("Quantidade de naves:");

   mudaTamanhoTerminal(X,Y);
   desligaCursor(true);
   noecho(true);


   TEMPO inicioNave = tempoInicio();
   TEMPO inicioTiro = tempoInicio();
   TEMPO inicioTiroInimigo = tempoInicio();

   int pontos=0;
   int vida=VIDA;
   vector <Imagem> tirosNave;
   vector <Imagem> tirosInimigos;
   vector <Imagem> inimigos;
   criaInimigos(inimigos, qtdInimigos);

   Imagem tiro("*");
   tiro.mudaCor(RED);
   Imagem nave("    | \n--=O=--\n",randomico(1,X),Y-3 );
   nave.setLimites(1,Y-2,X-nave.getLargura()+1,Y-2);
   nave.mudaCor(BLACK);
   nave.imprime();

   while(vida>0){
      gotoXY(1,2);
      mudaCor(RED,WHITE);
      cout << "Pontuação:" << setw(3) << pontos << " Nave restantes:" << setw(3)<<inimigos.size();
      cout << " Vida:" << setw(3) << vida;
      limpaEfeito();

      if( tempoPassado(inicioTiro)>VELOCIDADE_TIRO){
         inicioTiro = tempoInicio();
         movimentaTiros(tirosNave, inimigos, pontos);
      }

      if( tempoPassado(inicioTiroInimigo) > VELOCIDADE_TIRO){
         inicioTiroInimigo = tempoInicio();
         criaTirosInimigos(tirosInimigos, inimigos, vida, nave);
      }

      if( tempoPassado(inicioNave)>VELOCIDADE_NAVE){
         inicioNave = tempoInicio();
         movimentaInimigos(inimigos);
      }

      if( kbhit() ){
         nave.limpa();
         char tecla = getch();
         switch(tecla){
            case 'd':
            case 'D':
               nave.incrementaX();
```

```
73                  break;
74              case 'a':
75              case 'A':
76                  nave.decrementaX();
77                  break;
78              case 'f':
79              case 'F':
80                  //coloca os tiros no vetor
81                  if( tirosNave.size() < MAX_TIROS){
82                      tirosNave.push_back(tiro);
83                      tirosNave[ tirosNave.size()-1 ].setY( nave.getY() );
84                      tirosNave[ tirosNave.size()-1 ].setX( nave.getX() );
85                      tirosNave.push_back(tiro);
86                      tirosNave[ tirosNave.size()-1 ].setY( nave.getY() );
87                      tirosNave[ tirosNave.size()-1 ].setX( nave.getX()+nave.getLargura()-1 );
88                  }
89          }
90          nave.imprime();
91      }
92
93  }
94  return 0;
95 }
96
97 void criaTirosInimigos(vector <Imagem> & tirosInimigos, const vector <Imagem> & inimigos, int & vida, const Imagem & nave
        ){
98
99      //movimenta os tiros
100     vector<Imagem>::iterator t;
101     t = tirosInimigos.begin();
102     while( t != tirosInimigos.end() ){
103         t->limpa(); //equivalente há (*t).limpa();
104         t->incrementaY();
105         t->imprime();
106         if( t->colisao(nave)){
107             --vida; //colocar efeito de explosão futuramente
108             t->limpa();
109             t = tirosInimigos.erase(t);
110             break; //dar tempo da nave fugir do próximo tiro (se houver)
111         }
112         if( t->getY() == nave.getY()+1){ //não acertou e alcançou limite
113             t->limpa();
114             t = tirosInimigos.erase(t);
115         }
116         else{
117             ++t;
118         }
119     }
120
121     //criar novos tiros
122     if( tirosInimigos.size() < MAX_TIROS_INIMIGO){
123         for( int i=0; i<inimigos.size() && tirosInimigos.size() < MAX_TIROS_INIMIGO; ++i){
124             if( randomico(1,20)%5 == 0){
125                 tirosInimigos.push_back( Imagem("+", inimigos[i].getX(), inimigos[i].getY()+1));
126                 tirosInimigos[ tirosInimigos.size()-1 ].mudaCor(YELLOW);
127                 tirosInimigos[ tirosInimigos.size()-1].setLimites(1,1,X,Y);
128             }
129         }
130     }
131     return;
132 }
133
134
135 void movimentaInimigos(vector <Imagem> & inimigos){
136     for( int i=0; i<inimigos.size(); ++i){
137         inimigos[i].limpa();
138         Imagem inimigo = inimigos[i];
139         if( randomico()%2==0){
140             inimigo.incrementaY();
141         }
142         else{
143             inimigo.decrementaY();
144         }
```

```
145         if( randomico()%2==0){
146             inimigo.incrementaX();
147         }
148         else{
149             inimigo.decrementaX();
150         }
151         //se a nova posicao não colidir com os demais inimigos
152         if( !colisaoInimigos( inimigos, inimigo)){
153             inimigos[i] = inimigo;
154         }
155         inimigos[i].imprime();
156     }
157     return;
158 }
159 
160 //solução com iterator
161 void movimentaTiros(vector <Imagem> & tiros, vector<Imagem> & inimigos, int & pontos){
162 
163     vector<Imagem>::iterator t, i;
164 
165     t = tiros.begin();
166     while( t != tiros.end() ){
167         t->limpa();
168         t->decrementaY();
169         t->imprime();
170         //verificar se houve colisão com alguma nave inimiga
171         i = inimigos.begin();
172         bool acertou=false;
173         while( i!= inimigos.end()){
174             if( i->colisao( *t ) ){ //tirou acertou inimigo
175                 i->limpa(); //colocar efeito de explosão futuramente
176                 inimigos.erase(i);
177                 acertou=true;
178                 pontos+=10;
179                 break; //testar próximo tiro, este já era
180             }
181             ++i;
182         }
183         //eliminar o tiro se acertou inimigou ou se alcançou limite
184         //da tela
185         if( t->getY() == 3 || acertou){
186             t->limpa();
187             t = tiros.erase(t); //vai para o próximo tiro
188         }
189         else{
190             ++t;
191         }
192     }
193 }
194 
195 void criaInimigos(vector <Imagem> & inimigos, int qtdInimigos){
196     int i=0;
197     while(i<qtdInimigos){
198         string _e = "<-" + numeroToString(i+1)+"->";
199         Imagem inimigo(_e,randomico(2,X),randomico(3,Y-5) );
200         inimigo.mudaCor( WHITE );
201         inimigo.setLimites(2,3,X-3,Y-5);
202         if( !colisaoInimigos( inimigos, inimigo)){
203             inimigos.push_back(inimigo);
204             ++i;
205         }
206     }
207 }
208 
209 bool colisaoInimigos(const vector <Imagem> inimigos, Imagem inimigo){
210     for( int i=0;i<inimigos.size();++i){
211         if( inimigo.colisao( inimigos[i] )){
212             return true;
213         }
214     }
215     return false;
216 }
```

8 Aquário mágico - outro exemplo com referências

Programa 7: Aquário mágico.

```
1  //sprites (animação com vetores)
2  //criando vectors da classe Imagem
3  //programa_007.cpp
4  #include "biblaureano.h"
5
6  vector <Imagem> cria( string nomeArquivo, int xTerminal, int yTerminal, COR cor );
7  void movimentaPeixeX( int & peixe, vector <Imagem> & imagem, int xTerminal);
8  void movimentaPeixeY( int peixe, vector <Imagem> & imagem, int yTerminal);
9  void movimentaPeixeEixo( int & peixe, vector <Imagem> & imagem);
10
11 void movimentaGolfinhoX( int & golfinho, vector <Imagem> & imagem, int xTerminal);
12 void movimentaGolfinhoY( int golfinho, vector <Imagem> & imagem, int yTerminal);
13
14 int main(){
15
16    int xTerminal = readInt("Tamanho de x:");
17    int yTerminal = readInt("Tamanho de y:");
18
19    //tratamento do castelo
20    vector <Imagem> imagemCastelo = criaImagens("castelo.txt");
21
22    vector <vector <Imagem> > peixes;
23    int qtdPeixes = readInt("Quantos peixes do tipo 1?");
24    for(int i=0;i<qtdPeixes; ++i){
25       peixes.push_back(cria("peixe01.txt", xTerminal, yTerminal, RED));
26    }
27    qtdPeixes = readInt("Quantos peixes do tipo 2?");
28    for(int i=0;i<qtdPeixes; ++i){
29       peixes.push_back(cria("peixe02.txt", xTerminal, yTerminal, PURPLE));
30    }
31    qtdPeixes = readInt("Quantos peixes do tipo 3?");
32    for(int i=0;i<qtdPeixes; ++i){
33       peixes.push_back(cria("peixe03.txt", xTerminal, yTerminal, YELLOW));
34    }
35
36    vector <vector <Imagem> > golfinhos;
37    int qtdGolfinhos = readInt("Quantos golfinhos?");
38    for(int i=0;i<qtdGolfinhos;++i){
39       golfinhos.push_back(cria("golfinho.txt", xTerminal, yTerminal, GREEN));
40    }
41
42    mudaTamanhoTerminal( xTerminal, yTerminal );
43    desligaCursor(true);
44    limparTela();
45
46    for( int i = 0; i<imagemCastelo.size(); ++i){
47       //pega os pontos que compõem a imagem
48       vector <Ponto> pontos = imagemCastelo[i].getPontos();
49       for( int j=0; j<pontos.size(); ++j)
50       {
51          //altera randomica a cor de cada ponto
52          COR cor = (COR)randomico(0,QTY_COR);
53          pontos[j].setCor( cor );
54       }
55       //devolve os pontos alterados
56       imagemCastelo[i].setaPontos(pontos);
57
58       // coloca o castelo sempre no canto inferior direito
59       imagemCastelo[i].setX( xTerminal - imagemCastelo[i].getLargura() );
60       imagemCastelo[i].setY( yTerminal - imagemCastelo[i].getAltura() );
61    }
62
63
64
65    //controla a posicao inicial de cada peixe
66    vector <int> posPeixes;
67    for(int i = 0; i<peixes.size();++i){
```

```
68          if( randomico()%2==0){
69              posPeixes.push_back(0);
70          }
71          else{
72              posPeixes.push_back(1);
73          }
74      }
75      vector <int> posGolfinhos;
76      for( int i =0; i<golfinhos.size(); ++i){
77          if( randomico()%2==0){
78              posGolfinhos.push_back(0);
79          }
80          else{
81              posGolfinhos.push_back(2);
82          }
83      }
84
85      int castelo=0;
86      while(true) {
87          imagemCastelo[castelo].imprime();
88
89          for( int i =0; i<peixes.size(); ++i){
90            peixes[i][posPeixes[i]].imprime();
91          }
92
93          for( int i=0; i<golfinhos.size();++i){
94              golfinhos[i][posGolfinhos[i]].imprime();
95          }
96
97          espera(20);
98
99          imagemCastelo[castelo].limpa();
100
101         for( int i=0; i<golfinhos.size();++i){
102             golfinhos[i][posGolfinhos[i]].limpa();
103         }
104
105         for( int i =0; i<peixes.size(); ++i){
106           peixes[i][posPeixes[i]].limpa();
107         }
108
109
110         //movimentos dos peixes
111         //x
112         for( int i =0; i<peixes.size(); ++i){
113           movimentaPeixeX( posPeixes[i], peixes[i], xTerminal);
114         }
115
116         //movimento o y apenas se numero com final 0 ou 5 foi sorteado
117         if( randomico()%5 == 0){
118             for( int i =0; i<peixes.size(); ++i){
119                 movimentaPeixeY( posPeixes[i], peixes[i], yTerminal);
120             }
121         }
122         //altera o sentido dos peixes
123         if( randomico()%10==0){
124             for(int i=0;i<posPeixes.size();++i){
125                 if( randomico()%2==0){
126                     movimentaPeixeEixo(posPeixes[i], peixes[i]);
127                 }
128             }
129         }
130
131         //movimento dos golfinhos
132         for( int i=0;  i<golfinhos.size(); ++i){
133             movimentaGolfinhoX( posGolfinhos[i], golfinhos[i], xTerminal);
134         }
135         //movimento o y apenas se numero com final 0 ou 5 foi sorteado
136         if( randomico()%5==0){
137             for(int i=0; i<golfinhos.size(); ++i){
138                 movimentaGolfinhoY(posGolfinhos[i], golfinhos[i], yTerminal);
139             }
140         }
```

```
141
142         if(kbhit()) break;
143         if( ++castelo == imagemCastelo.size() ) castelo = 0;
144
145     }
146     cout << "Game Over" << endl;
147     return 0;
148 }
149
150 void movimentaPeixeX( int & peixe, vector <Imagem> & imagem, int xTerminal){
151     if( peixe == 0 ){
152             imagem[peixe].incrementaX();
153             if( imagem[peixe].getX() == (xTerminal-imagem[peixe].getLargura())){
154                 peixe = 1;
155                 imagem[peixe].setX( imagem[peixe-1].getX());
156             }
157     }
158     else{
159         imagem[peixe].decrementaX();
160         if( imagem[peixe].getX() == 1 ){
161             peixe = 0;
162             imagem[peixe].setX(1);
163         }
164     }
165     return;
166 }
167
168 void movimentaPeixeEixo( int & peixe, vector <Imagem> & imagem){
169     if( peixe == 0){
170         imagem[1].setX( imagem[0].getX() );
171         imagem[1].setY( imagem[0].getY() );
172         peixe = 1;
173     }
174     else{
175         imagem[0].setX( imagem[1].getX() );
176         imagem[0].setY( imagem[1].getY() );
177         peixe = 0;
178     }
179     return;
180 }
181
182 void movimentaPeixeY( int peixe, vector <Imagem> & imagem, int yTerminal){
183     int sentido = randomico();
184     if( sentido%2 ==0){
185         if( imagem[peixe].getY() >= 2){
186             imagem[peixe].decrementaY();
187         }
188     }
189     else{
190         if( imagem[peixe].getY() <= (yTerminal - imagem[peixe].getAltura())){
191             imagem[peixe].incrementaY();
192         }
193     }
194
195     return;
196 }
197
198
199 vector <Imagem> cria( string nomeArquivo, int xTerminal, int yTerminal, COR cor ){
200     //tratamento do peixe
201     vector <Imagem> imagem = criaImagens(nomeArquivo);
202     int x=randomico(1,xTerminal-imagem[0].getLargura());
203     int y=randomico(1,yTerminal-imagem[0].getAltura());
204     for( int i=0; i<imagem.size(); ++i)
205     {
206         imagem[i].mudaCor(cor);
207         imagem[i].setLimites(1,1,xTerminal-imagem[i].getLargura(), yTerminal);
208         imagem[i].setX(x);
209         imagem[i].setY(y);
210     }
211     return imagem;
212 }
213
```

```
214 void movimentaGolfinhoX( int & golfinho, vector <Imagem> & imagem, int xTerminal){
215     if( golfinho == 0 || golfinho == 1){
216         imagem[golfinho].incrementaX();
217         imagem[0].setX( imagem[golfinho].getX());
218         imagem[1].setX( imagem[golfinho].getX());
219         if( imagem[golfinho].getX() == (xTerminal-imagem[golfinho].getLargura())){
220             golfinho = 2;
221             imagem[2].setX( imagem[0].getX());
222             imagem[3].setX( imagem[1].getX());
223         }
224         if( golfinho == 0){
225             golfinho = 1;
226         }
227         else if( golfinho == 1){
228             golfinho = 0;
229         }
230 
231     }
232     else{
233         imagem[golfinho].decrementaX();
234         imagem[2].setX( imagem[golfinho].getX());
235         imagem[3].setX( imagem[golfinho].getX());
236         if( imagem[golfinho].getX() == 1 ){
237             golfinho = 0;
238             imagem[0].setX( imagem[2].getX());
239             imagem[1].setX( imagem[3].getX());
240         }
241         if( golfinho == 2){
242             golfinho = 3;
243         }
244         else if (golfinho == 3){
245             golfinho = 2;
246         }
247     }
248     return;
249 }
250 
251 void movimentaGolfinhoY( int golfinho, vector <Imagem> & imagem, int yTerminal){
252     int sentido = randomico();
253     if( sentido%2 ==0){
254         if( imagem[golfinho].getY() >= 2){
255             imagem[golfinho].decrementaY();
256         }
257     }
258     else{
259         if( imagem[golfinho].getY() <= (yTerminal - imagem[golfinho].getAltura())){
260             imagem[golfinho].incrementaY();
261         }
262     }
263     if( golfinho == 0 || golfinho == 1)
264     {
265         imagem[0].setY( imagem[golfinho].getY());
266         imagem[1].setY( imagem[golfinho].getY());
267     }
268     else{
269         imagem[2].setY( imagem[golfinho].getY());
270         imagem[3].setY( imagem[golfinho].getY());
271     }
272 
273     return;
274 }
```