Não há nada novo sob o sol, mas há
muitas coisas velhas que não conhecemos.

Ambrose Bierce

1 Mais um pouco de estruturas de dados - dados heterogêneos (registros) ou estruturas e classes em C++

Em C++ podemos utilizar uma classe ou estruturas para representar um registro. O conceito de registro visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que guardam estreita relação lógica. Registros correspondem a conjuntos de posições de memória conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições. O registro é um caso mais geral de *variável composta* na qual os elementos do conjunto não precisam ser, necessariamente, homogêneos ou do mesmo tipo. O registro é constituído por componentes. Cada tipo de dado armazenado em um registro é chamado de *campo*.

A figura 1 apresenta dois exemplos de registros: uma espaçonave (composto de nome do capitão, indicador de escudo, quantidade de energia e combustível) e um jogador (composto de nome, cargo, idade, recorde e indicativo se o jogador é ativo).

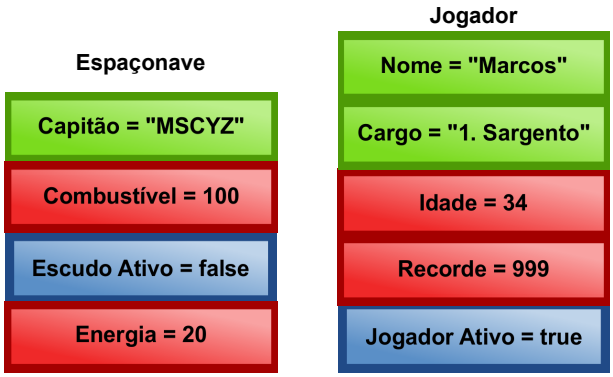

Figura 1: Representação de uma espaçonave.

Na variável composta homogênea (vetor), a individualização de um elemento é feita através de seus índices, já num registro cada componente é individualizado pela explicitação de seu identificador.

2 Classes e estruturas

Como comentado anteriormente, podemos representar registros em C++ pela utilização de uma classe ou de uma estrutura. Classes e estruturas tem campos para individualização, mas a utilização de classes possibilita incluir os métodos (habilidades) enquanto quem um registro não tem esta capacidade.

O programa 1 apresenta a representação do registro ESPACONAVE (conforme figura 1).

Programa 1: Registros em C++.

```cpp
//meu primeiro registro
//programa_001.cpp
#include "biblaureano.h"

//criando uma classe
class ESPACONAVE{
    public: // somos obrigados a especificar o escopo
    int energia;
    int combustivel;
    string nomeCapitao;
    bool escudoAtivo;
};

//criando uma estrutura
struct OUTRA_ESPACONAVE{
    int energia;
    int combustivel;
    string nomeCapitao;
    bool escudoAtivo;
};

int main(){
   ESPACONAVE minhaNave;
   minhaNave.energia = 20;
   minhaNave.combustivel = 100;
   minhaNave.nomeCapitao = "MSCYSZ";
   minhaNave.escudoAtivo = false;

   struct OUTRA_ESPACONAVE outraNave;
   outraNave.energia = 20;
   outraNave.combustivel = 100;
   outraNave.nomeCapitao = "MSCYSZ";
   outraNave.escudoAtivo = false;
   cout << "Game over!!!" << endl;
   return 0;
}
```

Uma classe é um modelo que representa um objeto no mundo real. Na linguagem C++ uma classe passa a se comportar como se fosse um tipo de variável.

3 Estruturas

A vantagem de se ter uma estrutura é que ela passa a ser um tipo definido, podendo-se definir de maneira simplificada uma ou mais variáveis. Cada variável desta estrutura é chamado de campo da estrutura. Com a estrutura definida pode-se fazer atribuição de variáveis do mesmo tipo de maneira simplificada.

Sintaxe de uso:

```cpp
struct NOME_ESTRUTURA{
    tipo_de_variável nome_variável_01
    tipo_de_variável nome_variável_02
    ...
};

struct NOME_ESTRUTURA NomeRegistro;
```

Para se fazer o acesso de um único campo deve-se utilizar o nome da estrutura seguida de um ponto e do nome do campo desejado da estrutura. A partir daí se aplica às regras de uma variável em C++. O programa 2 exemplifica a utilização de estruturas.

Programa 2: Registros em C++ utilizando estruturas (*structs*).

```
//meu primeiro registro
//utilizando estruturas
//programa_002.cpp
#include "biblaureano.h"

//criando uma estrutura
struct JOGADOR{
    string nome, cargo;
    int idade, recorde;
    bool jogadorAtivo;
};

int main(){
   //criando um registro de jogador
   struct JOGADOR jogador;

   jogador.nome = readString("Entre com o nome do jogador:");
   jogador.cargo = readString("Entre com o cargo do jogador:");
   jogador.idade = readInt("Entre com a idade do jogador:");
   jogador.recorde = readInt("Entre com o recorde do jogador:");
   jogador.jogadorAtivo = readBool("O jogador é ativo ?");

   cout << "***** Dados do jogador *****" << endl;

   cout << "nome do jogador: " << jogador.nome << endl;
   cout << "cargo do jogador:" << jogador.cargo << endl;
   cout << "idade do jogador:" << jogador.idade << endl;
   cout << "recorde do jogador:" << jogador.recorde << endl;
   cout << "Ativo ?:" << jogador.jogadorAtivo << endl;
   cout << "Game over!!!" << endl;
   return 0;
}
```

3.1 Entendendo o programa

Na primeira parte do programa ocorre a definição do registro. É definido o nome do registro (*JOGADOR*) e os campos que o compõem. Repare que as definições dos campos sempre ocorrem entre { e } (chaves) e os campos são declarados da mesma forma que as variáveis.

```
struct JOGADOR{
    string nome, cargo;
    int idade, recorde;
    bool jogadorAtivo;
};
```

Na sequência, é declarada uma variável para conter o registro.

```
   struct JOGADOR jogador;
```

E depois o acesso a cada campo do novo registro é dado pelo nome da **variável** que foi declarada como registro, seguindo de um ponto (.) e o nome do campo da definição.

```
   ...
   jogador.nome = readString("Entre com o nome do jogador:");
   jogador.cargo = readString("Entre com o cargo do jogador:");
   jogador.idade = readInt("Entre com a idade do jogador:");
   jogador.recorde = readInt("Entre com o recorde do jogador:");
```

```
6     jogador.jogadorAtivo = readBool("O jogador é ativo ?");
7     ...
```

3.2 Estruturas e o *typedef*

Pode-se também utilizar o *typedef* com uma estrutura, gerando assim um sinônimo para a estrutura. Quando é usado o *typedef* na definição de uma estrutura, não é preciso mais utilizar a palavra *struct* para definir mais variáveis do mesmo tipo (programa 3).

Programa 3: Registros em C++ utilizando estruturas (*structs*).

```
1  //meu primeiro registro
2  //utilizando estruturas e typedef
3  //programa_003.cpp
4  #include "biblaureano.h"
5
6  //criando uma estrutura (definição de registro)
7  typedef struct{
8      string nome, cargo;
9      int idade, recorde;
10     bool jogadorAtivo;
11 } JOGADOR;
12
13 int main(){
14    //criando um registro de jogador
15    JOGADOR jogador;
16
17    jogador.nome = readString("Entre com o nome do jogador:");
18    jogador.cargo = readString("Entre com o cargo do jogador:");
19    jogador.idade = readInt("Entre com a idade do jogador:");
20    jogador.recorde = readInt("Entre com o recorde do jogador:");
21    jogador.jogadorAtivo = readBool("O jogador é ativo ?");
22
23    cout << "***** Dados do jogador *****" << endl;
24
25    cout << "nome do jogador: " << jogador.nome << endl;
26    cout << "cargo do jogador:" << jogador.cargo << endl;
27    cout << "idade do jogador:" << jogador.idade << endl;
28    cout << "recorde do jogador:" << jogador.recorde << endl;
29    cout << "Ativo ?:" << jogador.jogadorAtivo << endl;
30    cout << "Game over!!!" << endl;
31    return 0;
32 }
```

Como pode ser observado no programa 3, houve mudanças na declaração do registro e da variável do registro, como pode ser destacado abaixo:

```
1     //criando uma estrutura
2     typedef struct {
3         string nome, cargo;
4         int idade, recorde;
5         bool jogadorAtivo;
6     } JOGADOR;
7
8     //criando um registro de jogador
9     JOGADOR jogador;
```

3.3 Exercícios

1. Você está montando um simulador de viagens de trens. Para tal é necessário controlar as passagens, embarques e desembarques de um trem. Defina um registro para realizar estes controles. O registro deve conter:

- data da viagem;
- código da viagem;
- origem e destino;
- número do assento;
- hora de partida;
- tempo de viagem;
- indicativo se o passageiro é fumante ou não;

2. Faça um programa, que utilizando a estrutura do exercício anterior, leia os dados do teclado.

4 Dados Homogêneos e Heterogêneos - Combinando registros e vetores - o exemplo dos sapos assassinos

Normalmente em um jogo é necessário combater vários inimigos simultaneamente. Imagine que no seu jogo você tem a missão de destruir vários *sapos assassinos deformados*. O registro desta espécie rara de sapo está representado na figura 2.

Figura 2: Representação do sapo assassino deformado.

Em C++, a representação deste sapo é:

```
1  typedef struct {
2     int nivel;
3     int forca;
4     int defesa;
5     int ataque;
6  } SAPO;
```

Mas como comentado anteriormente, você deve enfrentar vários *sapos assassinos deformados*, 5 sapos por exemplo. A utilidade dos registros torna-se evidente quando utilizados juntamente com vetores. Pode-se definir então um *vetor de registros* de sapos. A manipulação desse vetor segue ao mesmo tempo as regras para acesso de vetores e registros. Para se fazer acesso a um campo da estrutura deve-se indicar o índice do vetor seguido do ponto e o nome do campo da estrutura. A figura 3 ilustra um vetor que contém os 5 sapos.

Em C++ podemos declarar um vetor de sapos como se segue:

```
1  SAPO saposAssassinosDeformados[5];
```

E o acesso a cada campo do vetor é dado pelo nome da variável que contém o vetor de registros, a posição (índice) do vetor, seguindo de ponto (.) e o nome do campo:

Figura 3: Representação em vetor de sapos assassinos deformados.

```
1    ...
2    //acessando a primeira posição do vetor
3    saposAssassinosDeformados[0].nivel = 1;
4    saposAssassinosDeformados[0].forca = 10;
5    saposAssassinosDeformados[0].defesa = 5;
6    saposAssassinosDeformados[0].ataque = 3;
7
8    //acessando a terceira posição do vetor
9    saposAssassinosDeformados[2].nivel = 3;
10   saposAssassinosDeformados[2].forca = 20;
11   saposAssassinosDeformados[2].defesa = 15;
12   saposAssassinosDeformados[2].ataque = 9;
13   ...
```

4.1 Um outro exemplo de vetores de registros

Observe o programa 4. Neste programa a estrutura do jogador (figura 1) é representado.

Programa 4: Registros em C++ utilizando estruturas (*structs*) e vetores.

```
1  //vários registros
2  //utilizando estruturas, typedef e vetores
3  //programa_004.cpp
4  #include "biblaureano.h"
5
6  //criando uma estrutura (definição do registro)
7  typedef struct{
8      string nome, cargo;
9      int idade, recorde;
10     bool jogadorAtivo;
11 } JOGADOR;
12
13 int main(){
14    const int CAPACIDADE=5;
15    //criando um registro de jogador (variável)
16    JOGADOR jogadores[CAPACIDADE];
17
18    for(int jogador=0;jogador<CAPACIDADE; ++jogador){
19       cout << "Dados do jogador " << jogador + 1 << "." << endl;
20       jogadores[jogador].nome = readString("Entre com o nome do jogador:");
21       jogadores[jogador].cargo = readString("Entre com o cargo do jogador:");
22       jogadores[jogador].idade = readInt("Entre com a idade do jogador:");
23       jogadores[jogador].recorde = readInt("Entre com o recorde do jogador:");
24       jogadores[jogador].jogadorAtivo = readBool("O jogador é ativo ?");
25    }
26
27    for(int jogador=0;jogador<CAPACIDADE; ++jogador){
28       cout << "***** Dados do jogador " << jogador + 1 << " *****" << endl;
29       cout << "Nome do jogador: " << jogadores[jogador].nome << endl;
```

```
30        cout << "Cargo do jogador:" << jogadores[jogador].cargo << endl;;
31        cout << "Idade do jogador:" << jogadores[jogador].idade << endl;;
32        cout << "Recorde do jogador:" << jogadores[jogador].recorde << endl;;
33        cout << "Ativo ?:" << jogadores[jogador].jogadorAtivo << endl;;
34    }
35    cout << "Game over!!!" << endl;
36    return 0;
37 }
```

4.2 Entendendo o programa

A declaração da estrutura (registro) permanece idêntico. A única diferença está na declaração da variável que conterá os registros, afinal, agora é um vetor de registros.

```
1    ...
2    const int CAPACIDADE=5;
3    //criando um registro de jogador (variável)
4    JOGADOR jogadores[CAPACIDADE];
5    ...
```

O acesso também é diferenciado, pois agora é necessário indicar a posição do vetor que contém o registro que se deseja acessar.

```
1    ...
2    for(int jogador=0;jogador<CAPACIDADE; ++jogador) {
3       cout << "Dados do jogador " << jogador + 1 << "." << endl;
4       jogadores[jogador].nome = readString("Entre com o nome do jogador:");
5       jogadores[jogador].cargo = readString("Entre com o cargo do jogador:");
6       jogadores[jogador].idade = readInt("Entre com a idade do jogador:");
7       jogadores[jogador].recorde = readInt("Entre com o recorde do jogador:");
8       jogadores[jogador].jogadorAtivo = readBool("O jogador é ativo ?");
9    }
10   ...
```

4.3 Exercício

1. Altere o exercício anterior para pedir dados de vários passageiros.

4.4 Combinando registros com vetores internos - retomando o exemplo dos sapos assassinos

Também é possível utilizar vetores dentro de registros (*vetor interno ao registro*). Deve-se apenas ter o cuidado ao manipular os índices do vetor de registros e do vetor de cada registro. Vamos supor que o nosso *sapo assassino deformado* possui uma habilidade para criar e controlar *sapos de olhos vermelhos esbugalhados*. A representação deste sapo pode ser vista na figura 4.

Em C++, a representação deste sapo é:

```
1    typedef struct {
2       int nivel;
3       int alcanceLingua;
4       int ataque;
5    } SAPO_OLHO_VERMELHO;
```

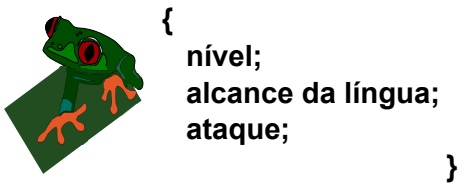

Figura 4: Representação de um sapo de olho vermelho esbugalhado.

Bom, como agora o nosso *sapo assassino deformado* consegue controlar *sapos de olhos vermelhos esbugalhados* devemos representar este novo sapo. A representação pode ser vista na figura 5, neste caso, vamos supor que o *sapo assassino deformado* possa controlar 3 *sapos de olhos vermelhos esbugalhados*.

Figura 5: Nova representação do sapo assassino deformado.

A representação em C++ para este sapo é:

```
1  typedef struct {
2     int nivel;
3     int forca;
4     int defesa;
5     int ataque;
6     //declaramos um vetor dentro do registro
7     SAPO_OLHO_VERMELHO sapinhos[3];
8  } SAPO;
```

Em C++ a definição da variável que controlará o vetor de sapos continua da mesma forma:

```
1  SAPOS saposAssassinosDeformados[5];
```

O que muda é a representação deste novo vetor (figura 6).

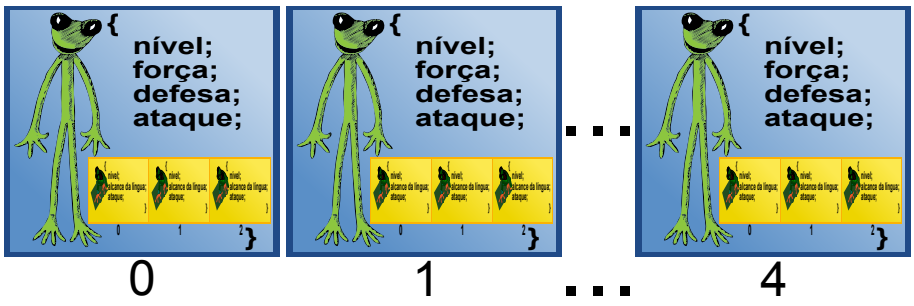

Figura 6: Representação em vetor de sapos assassinos deformados.

E o acesso a cada campo do vetor que continua sendo pelo nome da variável que contém o vetor de registros, a posição (índice) do vetor, seguindo de ponto (.) e o nome do campo, mas agora também temos o vetor interno do registro e este vetor de registros segue a mesma regra.

```
...
//acessando a primeira posição do vetor
saposAssassinosDeformados[0].nivel = 1;
saposAssassinosDeformados[0].forca = 10;
saposAssassinosDeformados[0].defesa = 5;
saposAssassinosDeformados[0].ataque = 3;
//acessando o primeiro sapinho do primeiro sapo
saposAssassinosDeformados[0].sapinhos[0].nivel = 1;
saposAssassinosDeformados[0].sapinhos[0].alcanceLingua = 5;
saposAssassinosDeformados[0].sapinhos[0].ataque = 3;

//acessando o segundo sapinho do primeiro sapo
saposAssassinosDeformados[0].sapinhos[1].nivel = 1;
saposAssassinosDeformados[0].sapinhos[1].alcanceLingua = 5;
saposAssassinosDeformados[0].sapinhos[1].ataque = 3;

//acessando a quinta posição do vetor
saposAssassinosDeformados[4].nivel = 5;
saposAssassinosDeformados[4].forca = 30;
saposAssassinosDeformados[4].defesa = 20;
saposAssassinosDeformados[4].ataque = 30;
//acessando o primeiro sapinho do quinto sapo
saposAssassinosDeformados[4].sapinhos[0].nivel = 4;
saposAssassinosDeformados[4].sapinhos[0].alcanceLingua = 15;
saposAssassinosDeformados[4].sapinhos[0].ataque = 10;

//acessando o segundo sapinho do quinto sapo
saposAssassinosDeformados[4].sapinhos[1].nivel = 1;
saposAssassinosDeformados[4].sapinhos[1].alcanceLingua = 5;
saposAssassinosDeformados[4].sapinhos[1].ataque = 3;
...
```

Desta forma, nosso jogo pode controlar vários *sapos assassinos deformados* que por sua vez também podem controlar outros sapos de olhos vermelhos.

Utilizar vetores ou até mesmo registros de vetores dentro da definição de registros requer alguns cuidados. No caso devemos primeiro definir o registro mais interno para depois utilizar no mais externo, como se segue:

```
//primeiro definir o registro interno
typedef struct {
   int nivel;
   int alcanceLingua;
   int ataque;
} SAPO_OLHO_VERMELHO;

//para depois definir o registro que utilizará na definição o
//registro definido anteriormente.
 typedef struct {
   int nivel;
   int forca;
   int defesa;
   int ataque;
   //declaramos um vetor dentro do registro
   SAPO_OLHO_VERMELHO sapinhos[3];
} SAPO;
```

Lembre-se da velha pergunta: Quem veio antes ? O ovo ou a galinha ? No nosso caso, não podemos utilizar algo que não existe.
O outro cuidado se refere no controle das posições de cada vetor de registros como já comentado.

4.5 Combinando registros com vetores internos - um outro exemplo

O programa 5 ilustra a utilização de um vetor de registros, onde cada registro contém um campo do tipo vetor.

Programa 5: Uso combinado de vetores e registros com vetores internos.

```
//vários registros
//utilizando estruturas, typedef e vetores
//programa_005.cpp
#include "biblaureano.h"

#define MAX_WEAPONS 5 //uma outra forma de declarar constante
#define MAX_PLAYERS 5 //uma outra forma de declarar constante
//criando uma estrutura (definição do registro)

typedef struct{
    string nomeArma;
    int qtdMunicao;
} ARMA;

typedef struct{
    string nome;
    ARMA armas[MAX_WEAPONS]; //vetor de registros
    string cargo;
} JOGADOR;

int main(){
   //criando um registro de jogador (variável)
   JOGADOR jogadores[MAX_PLAYERS];

   for(int jogador=0;jogador<MAX_PLAYERS; ++jogador){
      //leitura dos dados do jogador
      cout << "**** Entre com os dados do jogador "
           << jogador+1
           << "****"
           << endl;

      jogadores[jogador].nome = readString("Entre com nome do jogador:");
      jogadores[jogador].cargo = readString("Entre com o cargo do jogador:");

      //leitura das armas do jogador
      cout << "\t Entre com as armas" << endl;
      for( int armas = 0; armas < MAX_WEAPONS; ++armas){
         jogadores[jogador].armas[armas].nomeArma = readString("\tEntre com nome da arma:");
         jogadores[jogador].armas[armas].qtdMunicao =
                                          readInt("\tEntre com a quantidade de munição:");
      }
   }

   //impressão dos resultados
   cout << "**** Dados dos jogadores ****" << endl << endl;

   for(int jogador=0;jogador<MAX_PLAYERS; ++jogador){
      cout << "nome do jogador:" << jogadores[jogador].nome << endl;
      cout << "cargo do jogador:" << jogadores[jogador].cargo;

      for( int armas = 0; armas < MAX_WEAPONS; ++armas){
         cout << "\tNome da arma:"
              << jogadores[jogador].armas[armas].nomeArma
              << endl;

         cout << "\tQuantidade de munição:"
              << jogadores[jogador].armas[armas].qtdMunicao
              << endl;
      }
   }

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

4.6 Entendendo o programa

Inicialmente declaramos duas constantes para utilizar no restante do programa. A diretiva *#define* permite definir variáveis para o pré-compilador, fazendo com que ele atribua um valor a uma variável. Sendo importante ressaltar é que esta constante é uma variável do pré-compilador e não do programa. Cada vez que o pré-compilador encontrar esta variável, a mesma é substituída pelo conteúdo definido anteriormente, não levando em consideração o contexto de compilação. Ressaltando que a definição de uma variável de pré-compilação é pura substituição de caracteres.

```
...
#define MAX_WEAPONS 5 //uma outra forma de declarar constante
#define MAX_PLAYERS 5 //uma outra forma de declarar constante
...
```

Na sequência definimos as estruturas de registros utilizadas, repare que agora são declaradas duas estruturas e que a primeira estrutura é utilizada para declarar um *vetor de registros* na segunda estrutura.

```
...
typedef struct{
    string nomeArma;
    int qtdMunicao;
} ARMA;

typedef struct{
    string nome;
    ARMA armas[MAX_WEAPONS]; //vetor de registros
    string cargo;
} JOGADOR;
...
```

Finalmente, o maior cuidado que deve ser tomado é o controle do vetor de jogadores (*loop* mais externo) e controlar o vetor interno de cada registro (*loop* mais interno).

```
...
//lembre-se de cuidar de controlar o índice do vetor
jogadores[jogador].nome = readString("Entre com nome do jogador:");
jogadores[jogador].cargo = readString("Entre com o cargo do jogador:");

//leitura das armas do jogador
cout << "\t Entre com as armas" << endl;
for( int armas = 0; armas < MAX_WEAPONS; ++armas){
   //aqui, além de controlar o índice do jogador, devemos controlar o índice das armas
   jogadores[jogador].armas[armas].nomeArma = readString("\t Entre com nome da arma:");
   jogadores[jogador].armas[armas].qtdMunicao =
                                          readInt("\t Entre com a quantidade de munição:");
...
}
```

É possível declarar vetores de registros dentro de vetores de registros e assim sucessivamente.

5 Estruturas com vetores dinâmicos - revendo os exemplos dos sapos

Da mesma que é possível criar vetores estáticos de registros, também é possível criar vetores dinâmicos (*vectors*) de registros. Com a vantagem que é possível acrescentar elementos facilmente, realizar pesquisas, comparações, etc utilizando os métodos de *vector* já vistos anteriormente.

Para utilizar registros com vetores dinâmicos basta declarar o *vector* informando o nome da estrutura:

```
vector <nome estrutura> nome_variavel_vector;
```

O programa 6 apresenta como uma seria a implementação do exemplo dos sapos assassinos controlando os seus sapinhos.

Programa 6: Estruturas com vetores dinâmicos - o exemplos dos sapos.

```
//vários registros
//utilizando estruturas, typedef e vetores dinâmicos
//revendo os sapos assassinos
//programa_006.cpp
#include "biblaureano.h"

Imagem modificaCorPontos( Imagem aColorir, Imagem referencia);

//primeiro definir o registro interno
typedef struct{
   int nivel;
   int alcanceLingua;
   int ataque;
   Imagem desenho;
} SAPO_OLHO_VERMELHO;

//para depois definir o registro que utilizará na definição o
//registro definido anteriormente.
typedef struct{
   int nivel;
   int forca;
   int defesa;
   int ataque;
   //declaramos um vetor dentro do registro
   Imagem desenho;
   vector <SAPO_OLHO_VERMELHO> sapinhos;
} SAPO;

int main(){
   vector <SAPO> sapos;

   //carrega imagem do sapo colorido
   vector <Imagem> sapoImagem = retornaImagens("sapo.txt");
   Imagem sapoColorido = modificaCorPontos( sapoImagem[0], sapoImagem[1]);
   vector <Imagem> sapinhoImagem = retornaImagens( "sapinho.txt");
   Imagem sapinhoColorido = modificaCorPontos(sapinhoImagem[0], sapinhoImagem[1]);

   int qtdSapos = 3;
   int y = 1, x=1;

   for( int i=0; i< qtdSapos;++i ){
      //cria o novo sapo
      SAPO novoSapo;
      novoSapo.nivel = randomico(1,3);
      novoSapo.forca = randomico(1,5);
      novoSapo.defesa = randomico(1,7);
      //acerta coordenadas na tela do novo sapo
      sapoColorido.setX(x);
      sapoColorido.setY(y);
      novoSapo.desenho = sapoColorido;

      int qtdSapinhos = randomico(1,5);
      int xSapinho = x+sapoColorido.getLargura()+2;
```

```
54
55          for( int j=0; j<qtdSapinhos;++j){
56              //cria o novo sapinho
57              SAPO_OLHO_VERMELHO novoSapinho;
58              novoSapinho.alcanceLingua = randomico(1,7);
59              novoSapinho.ataque = randomico(1,5);
60              novoSapinho.nivel = randomico(1,3);
61              sapinhoColorido.setY(y);
62              sapinhoColorido.setX(xSapinho);
63
64              novoSapinho.desenho = sapinhoColorido;
65              novoSapo.sapinhos.push_back(novoSapinho);
66
67              //posição do próximo sapinho
68              xSapinho += sapinhoColorido.getLargura()+2;
69          }
70          sapos.push_back(novoSapo);
71
72          //posição do próximo sapão
73          y+= sapoColorido.getAltura() + 1;
74          x+= sapoColorido.getLargura() + 2;
75
76      }
77      for( int i=0; i<sapos.size();++i){
78          sapos[i].desenho.imprime();
79          for( int j=0; j<sapos[i].sapinhos.size(); ++j){
80              sapos[i].sapinhos[j].desenho.imprime();
81          }
82      }
83      espera(500); //5 segundos
84      cout << "Game Over!!!" << endl;
85      return;
86  }
87
88  Imagem modificaCorPontos( Imagem aColorir, Imagem referencia){
89      vector <Ponto> pColorir = aColorir.getPontos();
90      vector <Ponto> pReferencia = referencia.getPontos();
91      if( pColorir.size() == pReferencia.size() ){//garante que é a mesma imagem
92          for( int i=0; i< pColorir.size(); ++i){
93              if( pReferencia[i].getChar() == "1"){ //BLACK
94                  pColorir[i].setCor(BLACK);
95              }
96              else if ( pReferencia[i].getChar() == "2"){ //BLUE
97                  pColorir[i].setCor(BLUE);
98              }
99              else if ( pReferencia[i].getChar() == "3"){ //GREEN
100                 pColorir[i].setCor(GREEN);
101             }
102             else if ( pReferencia[i].getChar() == "4"){ //CYAN
103                 pColorir[i].setCor(CYAN);
104             }
105             else if ( pReferencia[i].getChar() == "5"){ //RED
106                 pColorir[i].setCor(RED);
107             }
108             else if ( pReferencia[i].getChar() == "6"){//PURPLE
109                 pColorir[i].setCor(PURPLE);
110             }
111             else if ( pReferencia[i].getChar() == "7"){ //YELLOW
112                 pColorir[i].setCor(YELLOW);
113             }
114             else if ( pReferencia[i].getChar() == "8"){ //WHITE
115                 pColorir[i].setCor(WHITE);
116             }
117             pColorir[i].colore();
118         }
119
120         aColorir.setaPontos( pColorir );
121
122     }
123     return aColorir;
124 }
```

5.1 Entendendo o programa

Inicialmente é definido a estrutura *SAPO_OLHO_VERMELHO*, esta estrutura será utilizada na estrutura principal *SAPO*. Repare que na estrutura *SAPO* estamos criando o campo *sapinhos* como um *vector* da estrutura *SAPO_OLHO_VERMELHO*.

```
...
//primeiro definir o registro interno
typedef struct{
    int nivel;
    int alcanceLingua;
    int ataque;
    Imagem desenho;
} SAPO_OLHO_VERMELHO;

//para depois definir o registro que utilizará na definição o
//registro definido anteriormente.
typedef struct{
    int nivel;
    int forca;
    int defesa;
    int ataque;
    //declaramos um vetor dentro do registro
    Imagem desenho;
    vector <SAPO_OLHO_VERMELHO> sapinhos;
} SAPO;
...
```

No programa principal *main()*, é criada a variável *sapos* como um *vector* da estrutura *SAPO*.

```
...
int main(){
    vector <SAPO> sapos;
...
```

Na sequência, dois vectors (*sapoImagem* e *sapinhoImagem*) são criados contendo as imagens que formarão a figura. Repare que também existe uma chamada a função *modificaCorPontos()*. Esta função irá receber as imagens carregadas nas variáveis *sapoImagem* e *sapinhoImagem* e retornará as imagens já com os pontos coloridos (variáveis *sapoColorido* e *sapinhoColorido*).

```
    ...
    //carrega imagem do sapo colorido
    vector <Imagem> sapoImagem = retornaImagens("sapo.txt");
    Imagem sapoColorido = modificaCorPontos( sapoImagem[0], sapoImagem[1]);
    vector <Imagem> sapinhoImagem = retornaImagens( "sapinho.txt");
    Imagem sapinhoColorido = modificaCorPontos(sapinhoImagem[0], sapinhoImagem[1]);
    ...
```

Apenas para fins de ilustração, o arquivo *sapinho.txt* (figura 7) contém duas imagens: a primeira representa a imagem como será vista na tela e na segunda imagem os pontos estão modificados por números representando as cores que serão utilizadas na primeira imagem.

Retornando ao programa principal (*main()*), é criada a variável *novoSapo*, baseada na estrutura *SAPO*. Seus campos são preenchidos aleatoriamente e ao final esta variável é incluída no *vector sapos* (utilizando-se o método *push_back()* do *vector*):

```
        ...

        //cria o novo sapo
        SAPO novoSapo;
        novoSapo.nivel = randomico(1,3);
        novoSapo.forca = randomico(1,5);
        novoSapo.defesa = randomico(1,7);
        //acerta coordenadas na tela do novo sapo
```

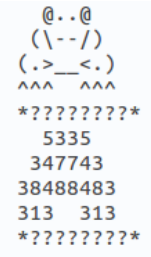

Figura 7: Conteúdo do arquivo *sapinho.txt*.

```
9       sapoColorido.setX(x);
10      sapoColorido.setY(y);
11      novoSapo.desenho = sapoColorido;
12
13      int qtdSapinhos = randomico(1,5);
14      int xSapinho = x+sapoColorido.getLargura()+2;
15
16      ...
17
18      ...
19      sapos.push_back(novoSapo);
20      ...
```

Mas para cada sapo também são criados vários sapinhos. Logo, dentro da estrutura de repetição, temos a definição da variável *novoSapinho*, baseada na estrutura *SAPO_OLHO_VERMELHO*. Os vários campos desta variável são preenchidas aleatoriamente. Ao final, a variável *novoSapinho* é colocado no campo *sapinhos* da variável *novoSapo*. Lembre-se que o campo *sapinhos* foi definido na estrutura *SAPO* como um *vector* da estrutura *SAPO_OLHO_VERMELHO*, portanto utilizamos o método *push_back()* do campo *vector sapinhos*:

```
1       ...
2       for( int j=0; j<qtdSapinhos;++j){
3          //cria o novo sapinho
4          SAPO_OLHO_VERMELHO novoSapinho;
5          novoSapinho.alcanceLingua = randomico(1,7);
6          novoSapinho.ataque = randomico(1,5);
7          novoSapinho.nivel = randomico(1,3);
8          sapinhoColorido.setY(y);
9          sapinhoColorido.setX(xSapinho);
10
11         novoSapinho.desenho = sapinhoColorido;
12         novoSapo.sapinhos.push_back(novoSapinho);
13
14         //posição do próximo sapinho
15         xSapinho += sapinhoColorido.getLargura()+2;
16      }
17      ...
```

Ao final, apenas realizamos a impressão de todos os sapos na tela. Utilizando a notação de vetor, executamos o método *imprime()* do campo *desenho* definidas nas estruturas *SAPO* e *SAPO_OLHO_VERMELHO*. Lembre-se que os campos *desenho* são do tipo *Imagem*) . Preste atenção na separação entre os diversos campos e no controle da posição de cada vetor:

```
1   ...
2   for( int i=0; i<sapos.size();++i){
3      sapos[i].desenho.imprime();
4      for( int j=0; j<sapos[i].sapinhos.size(); ++j){
5         sapos[i].sapinhos[j].desenho.imprime();
6      }
7   }
8   ...
```

E finalmente, chegamos na função *modificaCorPontos()*. Esta função recebe dois argumentos (parâmetros) do tipo *Imagem*. O primeiro argumento (variável *aColorir*) contém a imagem que será vista na tela e o segundo argumento (variável *referencia*) contém as cores de cada ponto.

Cada imagem é composta por vários pontos. Na classe *Imagem* está definido o método *getPontos()*. Este método retorna um *vector* dos pontos que compõem toda a imagem. Para cada ponto é verificado o seu contéudo (método *getChar()*) e configurado a nova cor do ponto (método *setCor()*). Ao final, informamos para que as cores seja aplicadas nos pontos (método *colore()*) e os pontos coloridos são alterados dentro da imagem (método *setaPontos()*):

```
Imagem modificaCorPontos( Imagem aColorir, Imagem referencia){
    vector <Ponto> pColorir = aColorir.getPontos();
    vector <Ponto> pReferencia = referencia.getPontos();
    if( pColorir.size() == pReferencia.size() ){//garante que é a mesma imagem
        for( int i=0; i< pColorir.size(); ++i){
            if( pReferencia[i].getChar() == "1"){ //BLACK
                pColorir[i].setCor(BLACK);
            }
            else if ( pReferencia[i].getChar() == "2"){ //BLUE
                pColorir[i].setCor(BLUE);
            }
            else if ( pReferencia[i].getChar() == "3"){ //GREEN
                pColorir[i].setCor(GREEN);
            }
            else if ( pReferencia[i].getChar() == "4"){ //CYAN
                pColorir[i].setCor(CYAN);
            }
            else if ( pReferencia[i].getChar() == "5"){ //RED
                pColorir[i].setCor(RED);
            }
            else if ( pReferencia[i].getChar() == "6"){//PURPLE
                pColorir[i].setCor(PURPLE);
            }
            else if ( pReferencia[i].getChar() == "7"){ //YELLOW
                pColorir[i].setCor(YELLOW);
            }
            else if ( pReferencia[i].getChar() == "8"){ //WHITE
                pColorir[i].setCor(WHITE);
            }
            pColorir[i].colore();
        }

        aColorir.setaPontos( pColorir );

    }
    return aColorir;
}
```

O resultado final do programa pode ser conferido na figura 8.

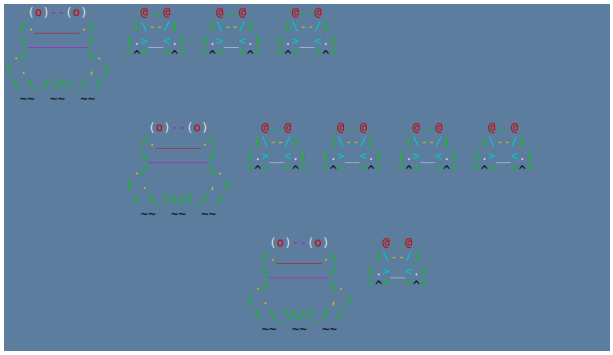

Figura 8: Resultado do programa 6 - exemplos dos sapos com vetores dinâmicos.

6 Operações entre estruturas

Algumas operações entre estruturas podem ser realizadas, vamos considerar a estrutura *PLACAR* :

```
typedef struct{
    int pontos;
    float bonus;
} PLACAR;
```

Podemos criar variáveis da estrutura e já realizar atribuições de valores aos membros (campos) da mesma. Os valores a serem atribuídos a seus membros devem ser colocados na ordem que foram definidos, separados por vírgulas e limitado por chaves:

```
    PLACAR a = { 5, 10.20 }, //é possível atribuir valores diretos
                   b = { 7, 21.22 };
```

Uma variável estrutura pode ser atribuída a outra variável do mesmo tipo:

```
    ...
    PLACAR c;
    c = a;
    cout << "c.pontos " << c.pontos << "\t"
         << "c.bonus " << c.bonus << endl;
    ...
```

Finalmente, realizamos a soma das variáveis *a* e *b* em total e mostramos o seu conteúdo ao final:

```
    ...
    //somar campo a campo
    PLACAR total;
    total.pontos = a.pontos + b.pontos;
    total.bonus = a.bonus + b.bonus;
    ...
```

Você poderia pensar em adicionar os valores de *a* e *b* na variável *total* por meio de uma simples instrução como:

```
total = a + b;
```

Mas operações simples como a soma não estão definidas para tipos criados com *struct*.

O exemplo utilizado está listado no programa 7:

Programa 7: Operações entre estruturas.

```
//operações entre estruturas
//programa_007.cpp
#include "biblaureano.h"

typedef struct
{
    int pontos;
    float bonus;
} PLACAR;

int main(){
```

```
12     PLACAR a = { 5, 10.20 }, //é possível atribuir valores diretos
13            b = { 7, 21.22 };
14
15     PLACAR c;
16     c = a;
17
18     cout << "c.pontos " << c.pontos << "\t"
19          << "c.bonus " << c.bonus << endl;
20
21     //somar campo a campo
22     PLACAR total;
23     total.pontos = a.pontos + b.pontos;
24     total.bonus = a.bonus + b.bonus;
25
26     cout << "total.pontos " << total.pontos << "\t"
27          << "total.bonus " << total.bonus << endl;
28     cout << "Game over!!!" << endl;
29     return 0;
30 }
```

7 Passando estruturas para funções

As estruturas podem ser passadas como argumento de funções da mesma forma que variáveis simples. No programa 8 temos a função *mostraPlacar* que recebe como parâmetro a estrutura *PLACAR*. A forma de tratar essa variável de estrutura é a mesma como uma variável do tipo *int*, *float* ou *char*:

Programa 8: Passando estruturas para funções.

```
1  //passando estruturas para funções
2  //programa_008.cpp
3  #include "biblaureano.h"
4
5  typedef struct{
6      int pontos;
7      float bonus;
8  } PLACAR;
9
10 void mostraPlacar( PLACAR placar );
11
12 int main(){
13    PLACAR a = { 5, 10.20 }, //é possível atribuir valores diretos
14           b = { 7, 21.22 };
15
16     PLACAR c;
17     c = a;
18
19     mostraPlacar(c);
20
21     //somar campo a campo
22     PLACAR total;
23     total.pontos = a.pontos + b.pontos;
24     total.bonus = a.bonus + b.bonus;
25
26     mostraPlacar(total);
27     cout << "Game over!!!" << endl;
28     return 0;
29 }
30
31 void mostraPlacar( PLACAR placar ){
32    cout << "pontos " << placar.pontos << "\t"
33         << "bonus " << placar.bonus << endl;
34    return;
35 }
```

7.1 Entendendo o programa

Antes de declarar o protótipo da função, devemos ter a estrutura declarada:

```
typedef struct{
    int pontos;
    float bonus;
} PLACAR;

void mostraPlacar( PLACAR placar );
```

A chamada a função é idêntica a varias outras que já vimos em aulas anteriores:

```
    ...
    mostraPlacar(c);
    ...
    mostraPlacar(total);
    ...
```

E a função *mostraPlacar* utiliza o parâmetro recebido:

```
void mostraPlacar( PLACAR placar ){
   cout << "pontos " << placar.pontos << "\t"
        << "bonus " << placar.bonus << endl;
   return;
}
```

8 Funções que retornam uma estrutura

Suponhamos que você queira usar uma chamada a uma função para obter os dados sobres os placares. A linguagem C++ permite que funções retornem uma estrutura completa. O programa 9 ilustra o conceito:

Programa 9: Recebendo estruturas de funções.

```
//recebendo estrutura de funções
//programa_009.cpp
#include "biblaureano.h"

typedef struct{
    int pontos;
    float bonus;
} PLACAR;

void mostraPlacar( PLACAR placar );
PLACAR novoPlacar();

int main(){
   PLACAR a,b;

   a = novoPlacar();
   b = novoPlacar();

   mostraPlacar(a);
   mostraPlacar(b);

   //somar campo a campo
   PLACAR total;
   total.pontos = a.pontos + b.pontos;
   total.bonus = a.bonus + b.bonus;

   mostraPlacar(total);
   cout << "Game over!!!" << endl;
   return 0;
```

```
30 }
31
32 void mostraPlacar( PLACAR placar ){
33    cout << "pontos " << placar.pontos << "\t"
34         << "bonus " << placar.bonus << endl;
35    return;
36 }
37
38 PLACAR novoPlacar(){
39    PLACAR novo;
40    novo.pontos = readInt("Entre com a quantidade de pontos:");
41    novo.bonus = readFloat("Entre com o valor do bonus:");
42    return novo;
43 }
```

8.1 Entendendo o programa

O tipo de retorno da função deve ser declarado com o nome da estrutura:

```
1 PLACAR novoPlacar();
```

E depois é só realizar as chamadas da função *novoPlacar* e atribuir o retorno para as variáveis *a* e *b*:

```
1    ...
2    a = novoPlacar();
3    b = novoPlacar();
4    ...
```

- Definir uma estrutura não cria nenhuma variável, somente informa ao compilador as características de um novo tipo de dado. Não há nenhuma reserva de memória.
- Uma vez criada a variável estrutura, seus membros (campos) podem ser acessados por meio do operador ponto (.). O operador ponto conecta o nome de uma variável estrutura a um membro dela.
- Podemos criar vetores dentro de estruturas e vetores de estruturas.
- Uma função pode receber estruturas como parâmetros e retornar estruturas.

8.2 Exercício

1. Em *Cafundos do Judas* existe uma fazenda chamada *PerdeuAsBotas*, cujo dono, é o senhor *BoiTata*. O senhor BoiTata quer implantar, em sua fazenda, um sistema para controlar os seus pastores e suas ovelhas. Ele precisa saber do pastor:
   a) Nome do pastor;

b) Idade do pastor;

c) Estado civil do pastor (Casado, Solteiro, Divorciado, Viúvo);

d) Nome dos cachorros do pastor, afinal, todo pastor que se preze tem pelo menos 1 auxiliares caninos podendo chegar a ter até 10 auxiliares caninos.

   i. Nome do amigo canino;

   ii. Raça do amigo canino;

   iii. Idade do amigo canino;

Um pastor cuida de 1 rebanho de ovelhas. Cada rebanho, pode conter até 150 ovelhas. Cada ovelha é identificada por:

- Código;
- Idade;
- Peso;
- Cor (Branca ou Preta);
- Data da última tosquia (retirada da lã);

Sabendo disto faça a leitura de todos os dados solicitados pelo senhor BoiTata. Lembre-se de validar todos os campos e de registrar a quantidade correta:

a) de pastores que a fazenda possui (considere um máximo de 10 pastores);

b) de amigos caninos de cada pastor;

c) da quantidade de ovelhas do rebanho.