Instituto Superior de Engenharia de Coimbra

Programação Orientada a Objetos
2022/23 | LEICE

Meta 2

Docentes: João Durães
Vasco Pereira

Discentes: Pedro Praça 2020130980
Filipe Oliveira 2018018618

Índice

1.0 Introdução

Este trabalho consiste em conceber, um simulador/jogo de construção e desenvolvimento na linguagem de programação de C++.

O objetivo deste trabalho é aplicar os conhecimentos adquiridos durante as aulas e construir um simulador de uma reserva natural povoada por diversos animais com diferentes comportamentos.

2.0 Leitura de ficheiros

Para facilitar a comparação entre o comando escrito pelo utilizador e os comandos disponíveis optamos pela criação de um vetor de comandos com a estrutura seguinte:

- {<nome>, <argumento 1> <argumento 2> … <argumento N>}

Para isso criamos uma classe chamada Comando com estrutura de um comando, como esta indicado acima.
Utilizamos esta estratégia tanto para o ficheiro de constantes como o ficheiro de comandos, ou seja, este vetor de comandos falado anteriormente inclui as constantes. Consideramos para este efeito as constantes como parte dos comandos.

```
vector<Comandos> comandos = { {"animal", "<especie> <linha> <coluna>" },  //0
                              {"animal", " <especie> "},
                              {"kill", " <linha> <coluna>"},
                              {"killid", "<id>"},
                              {"food", "<tipo> <linha> <coluna>"},
                              {"food", "<tipo>"},
                              {"feed", "<linha> <coluna>  <pontos nutritivos> <pontos de toxicidade>"},
                              {"feedid", "<ID> <pontos nutritivos> <pontos de toxicidade>"},
                              {"nofood", "<linha> <coluna>"},
                              {"nofood", "<ID>"},
                              {"empty", "<linha> <coluna>"}, //10
                              {"see", "<linha> <coluna>"},
                              {"info", "<ID>"},
                              {"n", ""},
                              {"n", "<N>"},
                              {"n", "<N> <P>"},
                              {"anim", ""},
                              {"visanim", ""},
                              {"store", "<nome>"},
                              {"restore", "<nome>"},
                              {"load", "<nome-do-ficheiro>"}, //20
                              {"slide", "<direcao> <linhas/colunas>"},
                              {"SCoelho", "<saudeInicial>" },
                              {"SOvelha", " <saudeInicial> "},
                              {"SLobo", " <saudeInicial> "},
                              {"SCanguru", "<saudeInicial>"},
                              {"VCoelho", "<duracaoDeVida>"},
                              {"VOvelha", "<duracaoDeVida>"},
                              {"VCanguru", "<duracaoDeVida>"},
                              {"VRelva", "<validade>"},
                              {"VLobo", "<instancias>"}, //30
                              {"historico", "<id>"},
                              {"SDragao", "<saudeInicial>"},
                              {"VDragao", "<instancias>"},
                              {"exit", ""} };
```

De modo que quando o programa for inicializado o ficheiro de constantes seja lido automaticamente, lemo-lo na classe iniciar. É no ficheiro "constantes.txt" onde estarão as constantes que se poderão vir a alterar.
As que não estiverem nesse ficheiro, o programa tem implementado as constantes *default* referidas no enunciado, essas variáveis são *static*, e são guardadas posteriormente nas classes filhos.

```
// limpa consola
system("cls");
i.iniciarReserva(linhas, colunas);
i.leFicheiro("constantes.txt");
```

Leitura do ficheiro constantes.txt no início do programa

```
// Constantes do Animais/Alimento
int Coelho::saudePDefeito = 20;
int Coelho::dVidaPDefeito = 30;
int Ovelha::saudePDefeito = 30;
int Ovelha::dVidaPDefeito = 35;
int Lobo::saudePDefeito = 25;
int Lobo::dVidaPDefeito = 40;
int Canguru::saudePDefeito = 20;
int Canguru::dVidaPDefeito = 70;
int Relva::validadePDefeito = 20;
int Dragao::saudePDefeito = 100;
int Dragao::dVidaPDefeito = 100;
```

Constantes dos Animais/Alimentos por default

3.0 Leitura e validação dos comandos

A leitura consiste em percorrer o texto com o *iterator* e separá-lo, este método só pode dividir *strings* por espaços (delimitador *default* * do *istream_iterator*), e copia as strings extraídas para o vetor, usando o algoritmo método copy.

```
copy( first: istream_iterator<string>( &: iss), last: istream_iterator<string>(), result: back_inserter( &: parametros));
```

De seguida percorremos o vetor de comandos (contemplado no ponto 2), e confirmamos se existe algum igual ao que o utilizador introduziu.

Se sim, é chamado o método “executaComando” que recebe os parâmetros e a posição do comando no vetor, onde de seguida vai ser validado todos os seus argumentos.

Este nosso método de leitura é semelhante quer para os ficheiros como para o teclado, o que diferencia é que no caso dos ficheiros temos de o abrir, enquanto no teclado recebemos uma *string* escrita pelo utilizador.

A nossa estratégia para as validações é a mesma para os dois, verifica se tem o número de parâmetros correto, se o que recebemos é um número (com auxílio da função *isNumber*), e se o nº de linhas e colunas está dentro do limite definido.

```
switch (opcao) {
    case 0: {//"animal", "<especie> <linha> <coluna>" ou //"animal", " <especie> "
        if(((parametros.size() != 4 && parametros.size() != 2) ) || (parametros[1] != "c" && parametros[1] != "o" && parametros[1] != "l" && parametros[1] != "g" && parametros[1] != "m" ))
            redirecionaInfos("Erro na introducao do comando!");

        else if(parametros.size() == 2 && !veioDoFicheiro) {
            redirecionaInfos(reserva.criaAnimal(parametros[1]));
            return "oi";

        } else if(parametros.size() == 2 && veioDoFicheiro) {
            redirecionaInfos2(reserva.criaAnimal(parametros[1]));
            return "oi";
        }

        else if (parametros.size() == 4){
            if((stoi(parametros[2]) > reserva.getNL()-1) || (stoi(parametros[2]) < 0) || (stoi(parametros[3]) > reserva.getNC()-1) || (stoi(parametros[3]) < 0) || !isNumber(parametros[2]) ||
                redirecionaInfos("Erro na introducao do comando!");
            else {
                if(!veioDoFicheiro) {
                    redirecionaInfos(reserva.criaAnimal(parametros[1], stoi(parametros[2]), stoi(parametros[3])));
                    return "oi";
                }
                else {
                    redirecionaInfos2( reserva.criaAnimal(parametros[1], stoi(parametros[2]), stoi(parametros[3])));
                    return "oi";
                }
            }
        }
        break;
    }
```

4.0 Estrutura do Trabalho

Main

É onde se faz a inicialização da classe Iniciar, que por sua vez corre o programa.

```
int main() {

    Iniciar i;

    i.iniciar();

    return 0;
}
```

Classes

Iniciar

Contêm a leitura do tamanho da reserva e respetiva validação com a opção de terminar o programa. Chama a função que cria a reserva e devolve na interface para posteriormente ser criado o terminal com as respetivas Windows.

Interface

A interface está encarregue da leitura de comandos e de constantes através de ficheiros, bem como através do teclado.

Após a leitura é feita as devidas verificações do nome dos comandos e constantes, bem como dos seus parâmetros. Informações mais detalhadas sobre este assunto: aqui.

Esta classe é também responsável pela representação visual da reserva toda, bem como a da sua área visível. Foi feita através da classe Terminal e das suas Windows (material fornecido). Os métodos que permitem esta abordagem são os seguintes:

```
// Terminal & Window
void terminal();
void redirecionaInfos(string txt);
```

O 1º método é onde se cria as *Windows* da área visivel e onde permite introduzir comandos. A janela da área visivel move-se através do *operator* fornecido:

```
Window& Window::operator>>(std::string& str) {
    ::noecho();
    ::cbreak();
    ::keypad(window, TRUE);
    int c = ::wgetch(window);
    switch(c) {
        case KEY_UP:
            str = "KEY_UP";
            ::nocbreak();
            ::echo();
            return *this;
        case KEY_DOWN:
            str = "KEY_DOWN";
            ::nocbreak();
            ::echo();
            return *this;
        case KEY_RIGHT:
            str = "KEY_RIGHT";
            ::nocbreak();
            ::echo();
            return *this;
        case KEY_LEFT:
            str = "KEY_LEFT";
            ::nocbreak();
            ::echo();
            return *this;
        case KEY_RESIZE:
            str = "KEY_RESIZE";
            ::nocbreak();
            ::echo();
            return *this;
```

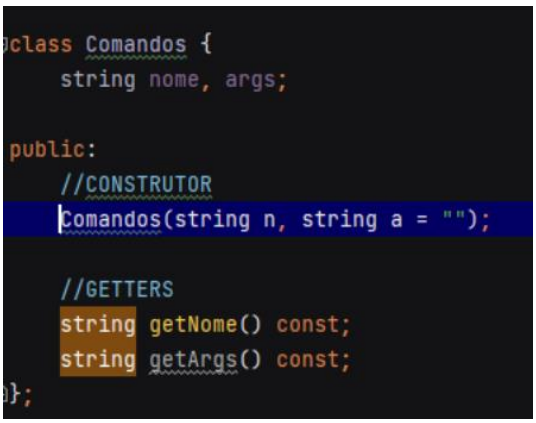
É através das setas do teclado que se movimenta esta janela, e foi esta a estratégia que decidimos usar no trabalho prático, assim não estão implementados o comando *slide* pois já temos essa funcionalidade implementada usufruindo da classe Terminal fornecida.

O 2º método "redirecionaInfos" permite mostrar todas as informações relativas aos animais, alimentos e mensagens de erro no terminal e nas suas *Windows.*

Comandos

Recebe no construtor o nome do comando e os seus respetivos argumentos.

Serve para guardar os nomes dos comandos, bem como os seus argumentos. Têm o propósito de facilitar a comparação já explicada neste ponto.



Reserva

Nesta classe existe a variável *IDglobal* que através da sua manipulação através do operador ++ é enviado para os construtores dos animais e alimentos. O número máximo de Linhas e Colunas foi decidido guardar na reserva pois toda a manipulação da reserva é feita na mesma.

Os vetores (de ponteiros) animais e alimentos é onde são guardados os dados dos animais e alimentos existentes durante a execução do programa. O "arrayReserva" é uma lista ligada dinâmica de *strings* onde é criada a própria reserva e onde é mostrado através de caracteres a posição dos animais e alimentos. O construtor da reserva recebe os valores NL e NC sendo atribuídos através de uma lista de inicialização.

Existem dois comandos para criar alimentos e para criar animais. Usamos *overloading* de funções para conseguirmos criar um Animal/Alimento quando é indicado apenas a espécie/tipo respetivamente, e nesse caso são geradas coordenadas *random* através do método *random* presente nesta classe. Quando são enviados os quatro argumentos (nome_comando, espécie/tipo, linhas, colunas) é criado o animal/alimento com as coordenadas indicadas.

```
void criaAnimal(const string& especie, int lin, int col);
void criaAnimal(const string& especie);

void criaAlimento(const string& tipo, int lin, int col);
void criaAlimento(const string& tipo);
```

Implementamos a movimentação dos animais na reserva, assim como todas as funcionalidades da mesma referidas no enunciado. A maioria destas funcionalidades estão presentes no método Avança().

```
//Jogo em si (movimentos, percecoes, etc...)

string avanca();
```

Store

Na implementação dos comandos *Store* e *Restore* foi criado um vetor dinâmico do tipo *Store* onde é guardada toda a informação relevante para a criação de uma Reserva através do *Restore*.
No momento que é introduzido o comando Store, na interface criamos uma nova reserva, de seguida, na classe store essa reserva vai ficar toda a informação da reserva atual.

```
class Store{

    string nome;
    Reserva reserva;
    vector<Animal*> animais;
    vector<Alimento*> alimentos;
    string **arrayReserva;

public:

    Store(Reserva& m, string n="save") : nome(n), reserva(m) {
```

```
if(f == 0) {
    stores.emplace_back(new Store(*reserva, parametros[1]));
    redirecionaInfos("Store efetuado com sucesso!");
    return "oi";
```

Animal

A classe Animal tem definida as suas variáveis e os consequentes *Getters* e *Setters* necessários para controlar as variáveis. Existem dois construtores nesta classe, ambos contêm as coordenadas, saúde, e o seu ID.

O construtor sem a duração de vida é exclusivamente usado para o Lobo pois este não necessita de receber esta variável através do construtor (não tem duração de vida).

Introduzimos o conceito de Herança através das funções virtuais implementadas de forma que se consiga aceder aos animais derivados. Desta maneira tornámos esta classe numa classe Abstrata, fazendo dela uma classe Base.

O objetivo da aplicação dos conceitos de Herança e de classe Abstrata é deixarmos de criar um animal, e passarmos a criar um objeto derivado da classe Base (classe Animal) com as suas próprias características.

```
// Métodos Virtuais

virtual string info() const;
virtual string infoExtra() const;
virtual int getPeso() const { return 0; };
virtual void setPeso(int peso) {};
virtual int getFome() const { return 0; };
virtual void setFome() {};
virtual void setFome(int val) {};
virtual string getEspecie() const = 0;
```

Métodos Virtuais

Alimento

A classe Alimento tem definida as suas variáveis e os consequentes *Getters* e *Setters* necessários para controlar as variáveis. Existem dois construtores nesta classe, ambos contêm as coordenadas, e o seu ID.

O construtor com a validade é exclusivamente usado pela Relva pois esta é a única classe que necessita de receber esta variável através do construtor.

Introduzimos o conceito de Herança através das funções virtuais implementadas de forma que se consiga aceder aos alimentos derivados. Desta maneira tornámos esta classe numa classe Abstrata, fazendo dela uma classe Base.

O objetivo da aplicação dos conceitos de Herança e de classe Abstrata é deixarmos de criar um alimento, e passarmos a criar um objeto derivado da classe Base (classe Alimento) com as suas próprias características.

```
class Alimento {

    int ID;
    int linha, coluna;
    int validade;
    string empty;

public:

    Alimento(int linha, int coluna, int ID);
    Alimento(int linha, int coluna, int ID, int validade);

    //Getters & Setters - classe base

    int getId() const;
    void setId(int id);
    int getLinha() const;
    int getColuna() const;
    int getValidade() const;
    void setValidade(int validade);


    // Métodos Virtuais

    virtual string info() const;
    virtual string infoExtra() const;
    virtual void consumirAlimento() {} ;
    virtual void apodrecer() {}; // o alimento fica podre ao longo do tempo,
    virtual void esgotaDuracao() {};
    virtual string getCheiro() const = 0;
    virtual void setCheiro(string str) {};
    virtual string getTipo() const = 0;
    virtual int getValorNutritivo() const { return 0;};
    virtual int getToxicidade() const { return 0;};
    virtual void setValorNutritivo(int valorNutritivo) {};
```

Historico

Foi implementado um array dinâmico (sem uso de bibliotecas STL) do tipo Historico onde é guardado o historial de alimentação de cada animal. Para ter acesso a esta funcionalidade criámos um comando adicional denominado de "historico", que apresenta o histórico de alimentação do animal através do seu id, com a seguinte estrutura:

- {"historico", <"id">}

```
Historico * array;
int size, tam;
```

Histórico

```
string mostraHistorico() const;
void adicionaInfo(string comida, int vNutritivo, int pToxicidade);
```

Métodos de acesso ao Histórico

5.0 Animal e Alimento Mistério

Animal-mistério

Dragao

- Representado visualmente por um “m”.
- Saúde inicial = “SDragao” = 100 pontos.
- Consegue aperceber-se do que o rodeia a 8 posições de distância à sua volta na reserva.
- Desloca-se duas posições em cada instante, aleatoriamente (direção), a não ser que veja algo interessante.
- Tem um peso inicial de 20 Kg quando nasce, torna-se adulto quando tiver 50 kg. Ganha 2kg a cada instante.
- A cada instante aumenta a sua fome em 1. Se tiver mais do que 20 de fome passa a perder 1 ponto de saúde a cada instante e passa a deslocar-se a 3 posições de cada vez. Se tiver mais do que 35 de fome perde 2 pontos de saúde a cada instante e desloca-se a 6 posições de cada vez.
- Não morre espontaneamente. Morre se a sua saúde chegar a 0.
- Se se aperceber de um alimento que cheire a “carne” ou “batata frita” no caso do dragão bebé nas suas redondezas, dirige-se na sua direção.
- Se se aperceber de um animal nas suas redondezas dirige-se na sua direção com velocidade 2 posições (3 se tiver mais do que 20 de fome). Em caso de haver vários, persegue o que for mais pesado.
- Se se encontrar na mesma posição, ou numa posição adjacente a um animal que pese menos que ele, mata-o.
- Se se encontrar na mesma posição que um alimento que cheire a “carne”, ou “batata frita” no caso do dragão bebé consome esse alimento.
- Se cheirar a “batata” também vai de encontro com esse alimento e se estiver na mesma posição torna essa batata em batata frita (“cuspiu fogo”).
- O dragão põe um ovo com uma probabilidade de 20%. Do ovo nasce um Dragão bebé se o ovo não tiver sido comido anteriormente. Este só pode ser comido pelos Lobos. O ovo aparece numa posição que não esteja a mais de 3 posições de distância (linha e coluna).
- O Dragão bebé alimenta-se unicamente das batatas fritas que o Dragão Adulto vai deixando para ele.
- Quando morre faz aparecer numa posição ao seu lado um Alimento do tipo Corpo, cujo valor nutritivo é 25 e o nível de toxicidade é 0.

Alimento-mistério

Batata

- Representado por um “a”.
- Duração é de 30 instantes.
- Valor nutritivo 5 e a cada instante diminui 1 ponto até 0.
- Toxicidade 0.
- A toxicidade aumenta 1 ponto a cada instante até ao máximo de 10 se o valor nutritivo estiver a 0.
- Tem cheiro a “batata” e “verdura”.
- Se o Dragão passar por cima da batata, esta é transformada em Batata frita.

Batata Frita

Ao contrário da batata e do Ovo de Dragão a Batata Frita não é uma derivada da classe Alimento, está simplesmente inserida na batata como uma booleana, ou seja, se se tornar Batata Frita fica com as características referidas abaixo.

- Representado por um “a”.
- Duração é de 30 instantes.
- Valor nutritivo 20 e a cada instante diminui 3 pontos.
- A toxicidade aumenta 2 ponto a cada instante até ao máximo de 20 se o valor nutritivo estiver a 0.
- Tem cheiro de "batataFrita".

Ovo de Dragão

- Representado por um “e”.
- A duração é de 20 instantes. Após os mesmos nasce o Dragão Bebé.
- Valor nutritivo 15 e a cada instante diminui 1 ponto até chegar a 0.
- Toxicidade 0.
- A toxicidade começa a aumentar 1 ponto quando passar dos 15 instantes até aos 20 instantes de vida, e ao mesmo tempo também o valor nutritivo desce 1 ponto.
- Tem cheiro a "ovo".

Requisitos e Funcionalidades

Todos os requisitos elencados no enunciado foram implementados.

Contudo, o comando Store e Restore apresentam algumas falhas. Dá para guardar um momento do jogo e voltar a este, quer seja após um simples movimento, ou algum animal ou alimento a desaparecer.

No entanto após fazer restore, se avançarmos no jogo (comando “n”) o animal movimenta-se normalmente, mas vai deixar o seu carater imprimido no ecrã (no ecrã observa-se dois caracteres do mesmo animal, sendo que um deles não corresponde a nenhum animal, é apenas um carater impresso no ecrã).

Além desta falha, o histórico de animais não acompanha os restores feitos. Ou seja, um animal comeu um alimento, fazemos store a, de seguida come outro alimento e fazemos restore a, vai aparecer no ecrã que comeu dois alimentos e devia aparecer apenas um.