

Relatório do trabalho prático de POO

Primeira Meta

Autores

Renato Santos 2020134927

Sérgio Alves 2020134949

Coimbra, 23 novembro 2021

Índice

- I. Geração do Mapa - 2
 - I. Mapa Estético - 2
 - II. Mapa de Informação - 3
 - i. Classe Cell - 3
- II. Leitura, validação e implementação de comandos - 4
 - I. Desconstrução - 4
 - II. Validação - 5
 - III. Execução - 5
 - i. Comando exec - 5
 - ii. Comando cons - 6
 - iii. Comando cont - 7
 - iv. Comando list - 8
 - v. Comando config – 9
- III. Termos Técnicos - 9

Geração do Mapa

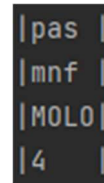
O grupo fez a implementação de dois mapas dinâmicos:

- Mapa Estético;
- Mapa de Informação.

Mapa Estético

O mapa estético é um vetor bidimensional de ponteiros de strings, que contém toda a informação (formatada) a ser impressa na consola. Essa informação é adquirida por contacto com o mapa de informação. Cada célula do mapa, é composta por um ponteiro que aponta para um array de quatro strings. Cada string representa uma linha de informação, estando ordenadas da seguinte forma:

1. Ambiente;
2. Edifício ;
3. Trabalhadores;
4. Número de trabalhadores.



pas
mnf
MOLO
4

```
void createMap(vector<vector<string*>> &map, vector<vector<Cell>> &cells) //cria mapa inicial
{
    for (int i = 0; i < cells.size(); i++) {
        vector<string*> temp;
        for (int j = 0; j < cells[0].size(); j++)
        {
            string* line = new string[4];
            vector<char> workers = cells[i][j].getWorkers();
            string str(workers.begin(), workers.end());
            line[0] = cells[i][j].getBiome();
            line[1] = cells[i][j].getBuilding();
            line[2] = str;
            line[3] = to_string(cells[i][j].countWorkers());
            stringController(line);
            temp.push_back(line);
        }
        map.push_back(temp);
    }
}
```

Mapa de Informação

O mapa de informação é o “backbone” de todo o trabalho, sendo o principal distribuidor de informação das funções. O mapa de informação é um vetor bidimensional de objetos do tipo Cell, sendo que estes contêm toda a informação necessária ao funcionamento do programa. Todos os objetos são construídos com um bioma aleatório, no entanto é garantida a existência de pelo menos uma célula do tipo “pasto”.

```

void createCells(vector<vector<Cell>> &cells, int rows, int cols) //cria duplo vetor de objetos (contém informação)
{
    string random[] = {"mnt", "dsr", "flr", "pas", "ptn"};

    for (int i = 0; i < rows; i++) {
        vector<Cell> temp;
        for (int j = 0; j < cols; j++)
            temp.emplace_back(random[rand() % 5]);
        cells.push_back(temp);
    }
    cells[rand() % cells.size()][rand() % cells[0].size()].setBiome("pas");
}

```

Classe Cell

A classe Cell é a responsável pela criação de todas as células do mapa de Informação. Esta contém como variáveis privadas toda a informação relativa a uma célula do mapa. Como funções públicas, tem um conjunto de funções simples que maioritariamente obtêm e alteram os valores das variáveis primárias.

```

class Cell {

    string biome;
    string building;
    vector<char> workers;
    int n_workers ;

public:

    Cell(string biome);
    ~Cell();
    void setBuilding(string type); //altera a construção da célula
    void setBiome(string type); //altera o bioma da célula
    void destroyBuilding(); //remove a construção da célula
    void addWorker(char type); //adiciona um trabalhador à célula
    string getBiome() const; //retorna o bioma
    string getBuilding() const; //retorna a construção de uma célula em específico
    vector<char> getWorkers() const; //retorna os trabalhadores de uma célula em específico
    int countWorkers() const; //retorna o número total de trabalhadores em uma célula específica
};

```

O construtor desta classe, visa-se apenas a iniciar cada objeto com um valor para o bioma e com o número de trabalhadores a 0;

```

Cell::Cell(string biome) : biome(biome), n_workers(0){}

```

Leitura, validação e implementação de comandos

Os comandos são o que permite ao utilizador trocar informação com os mapas. O seu processamento é um processo simples:

- Desconstrução;
- Validação;
- Execução.

Desconstrução

A desconstrução consiste na receção de uma frase e na sua separação em palavras. Para este efeito usamos a função *deconstructor()*. Esta função recebe uma string e um ponteiro de string e tem como finalidade fazer o ponteiro apontar para um array de quatro strings, isto porque, no máximo, um comando recebe 3 argumentos (1 comando + 3 argumentos = 4 strings).

```
void deconstructor(const string& comando, string* comandos) //recebe frase, descontroi-a em palavras,
{
    int i = 0;
    string word;
    istringstream iss(comando);
    while(iss >> word)
    {
        comandos[i] = word;
        if(i == 3)
            break;
        i++;
    }
}
```

Validação

Para a validação do número de argumentos passados, implementámos a função *checkArgument()*, que recebe o número de argumentos de um comando e o ponteiro que aponta para o array formado no *deconstructor*. Esta função devolve *true* se o número de argumentos estiver certo ou *false* em caso contrário.

```
bool checkArguments(int number, string *comandos ){
    if(comandos[number].empty()){
        cout << "Numero de argumentos invalido" << endl;
        cout << "Este comando utiliza " << number << " argumento(s)" << endl;

        system( _Command: "pause");

        return false;
    }
    return true;
}
```

Execução

A execução é uma função de grande dimensão (*execCommands()*), que verifica qual o comando a ser usado e o executa. Todos os comandos são excertos de código de dita cuja função.

Como a primeira meta não necessitava da implementação de todos os comandos, apenas vão ser abordados no relatório os que foram implementados.

Comando exec

O comando exec recebe como argumento o nome de um ficheiro .txt, sendo suposto dito cujo ficheiro ter uma lista de comandos, separados por uma mudança de linha, para serem executados. O código extrai as linhas dos comandos, dando recurso à função *getline()*, que armazena as linhas num array de strings; Cada uma das strings do array é processada como se fosse um comando, passando pelas mesmas funções referenciadas anteriormente, exceto a *execCommands()*, sendo que vai para a função *execFile()* que é especializada em executar os comandos provenientes do comando exec.

```
fstream file;
file.open(comandos[1].c_str(), ios::in);
if (file.is_open()) {
    string reader;
    vector<string> configs;
    while (getline(file, reader)){

        configs.push_back(reader);
    }
    file.close();
    for(int i = 0; i < configs.size(); i++)
    {
        string* aux = new string[4];
        deconstructor(configs[i], aux);

        execFile(cells, aux);
    }
    return 1;
}
cout << "Falha ao abrir o ficheiro" << endl;
return 1;
```

Comando cons

O comando cons é responsável pela construção de edifícios no mapa.

O comando recebe como argumentos o *tipo de edifício*, a *linha* e a *coluna* onde vai ser construído. Ele utiliza os argumentos *linha* e *coluna* para localizar a célula do mapa onde se quer efetuar a alteração, e de seguida usa a função `Cell::setBuilding` para alterar o valor da string *building* do objeto em questão.

```
void checkTypeBuilding(string *comandos, vector<vector<Cell>> &cells){

    int i;
    int rows = stoi( str: comandos[2]);
    int cols = stoi( str: comandos[3]);

    string buildings[6] = {"mnf", "mnc", "elec", "bat", "fun", "edx"};

    for(i=0; i<6; i++){
        if(comandos[1] == buildings[i]){
            if(checkBuildings( &: cells, rows, cols)){
                cells[rows-1][cols-1].setBuilding( type: comandos[1]);
                return;
            }
            else{
                cout << "Nesse local ja se encontra uma construcao" << endl;
                system( _Command: "pause");

                return;
            }
        }
        else{
            cout << "A construcao que inseriu e invalida" << endl;
            system( _Command: "pause");
        }
    }
}
```

Comando cont

O comando cont é responsável pela contratação de trabalhadores.

O comando recebe como argumento o tipo de trabalhador a contratar e coloca-os na primeira célula com bioma do tipo “pasto”. Para a obtenção do bioma utiliza a função *Cell::getBiome()*, e para a adição de um trabalhador ao vetor *trabalhadores* utiliza a função *Cell::addWorker()*.

```
void cont(vector<vector<Cell>> &cells, string type) {
    char type_char;

    if(type != "oper" && type != "len" && type != "miner"){
        return;
    }

    if(type == "oper"){
        type_char = 'O';
    }
    if(type == "len"){
        type_char = 'L';
    }
    if(type == "miner"){
        type_char = 'M';
    }

    for (int i = 0; i < cells.size(); i++) {
        for (int j = 0; j < cells[0].size(); j++) {
            if (cells[i][j].getBiome() == "pas") {
                cells[i][j].addWorker(type_char);
                return;
            }
        }
    }
}
```


Comando list

Comando que dá ao utilizador um conjunto de informações gerais, quando usado sem argumentos; ou um conjunto de informações detalhadas de uma célula, quando usados os argumentos linha e coluna.

Para isso é utilizado overloading de duas funções, uma para o primeiro caso descrito e outra para o segundo caso descrito.

A função relativa ao primeiro caso, não apresenta nenhuma informação relevante ao utilizador, visto que ainda não foram implementados recursos e afins.

```
void list(){  
  
    cout << "lista geral // EM DESENVOLVIMENTO" << endl;  
    system("_Command: \"pause\"");  
}
```

A função que recebe os argumentos, apresenta alguma informação, no entanto pretendemos torná-la bastante mais detalhada até à segunda meta. A única informação relevante que apresenta, é uma lista de todos os trabalhadores presentes na célula.

```
void list(const string& rows, const string& cols, vector<vector<Cell>> &cells){  
  
    int irows = stoi(rows);  
    int icols = stoi(cols);  
  
    cout << "\n--Informacoes--" << endl;  
    cout << "Bioma -> " << cells[irows-1][icols-1].getBiome() << endl;  
  
    if(cells[irows-1][icols-1].getBuilding().empty()){  
        cout << "Construcao -> Vazio " << endl;  
    }  
    else{  
        cout << "Construcao -> " << cells[irows-1][icols-1].getBuilding() << endl;  
    }  
  
    if(cells[irows-1][icols-1].getWorkers().empty()){  
        cout << "Trabalhadores -> Vazio" << endl;  
    }  
    else{  
        cout << "Trabalhadores -> ";  
        for(int i = 0; i < cells[irows-1][icols-1].getWorkers().size(); i++){  
            cout << cells[irows-1][icols-1].getWorkers()[i] << " ";  
        }  
        cout << endl;  
    }  
  
    cout << "Numero total de trabalhadores -> " << cells[irows-1][icols-1].countWorkers() << endl;  
}
```

Comando config

O comando config é um comando que lê de um ficheiro o nome e valor de variadas variáveis relevantes à implementação da meta 2.

Como as variáveis ainda não são relevantes o que este comando faz é ler o ficheiro, extraindo informação linha a linha, e depois separando essa informação em dois vetores, um para o nome da variável e outro para o valor da variável. Isto torna a futura implementação deste comando bastante mais fácil, visto que a informação já está organizada desde a meta 1.

```
fstream file;
file.open(s comandos[1].c_str(), ios::in);

if (file.is_open()) {
    string reader;
    vector<string> configs; //armazena dados da file | Formato: "<nome> <valor>"
    while (file >> reader) {
        configs.push_back(reader);
    }
    file.close();

    vector<string> names;
    vector<int> values;
    for (int i = 0; i < configs.size() / 2; i++) //cria dois vetores, um que contem os nomes das variav
    {
        //e outro que contém os valores das variáveis
        names.push_back(configs[2 * i]);
        values.push_back(stoi(str configs[(2 * i + 1)]));
    }

    cout << "Ficheiro " << comandos[1] << " lido com sucesso" << endl;
    cout << "Mudanca a variaveis por implementar" << endl;

} else {
    cout << "Erro ao verificar file (Requer que ficheiro esteja dentro da pasta cmake-build-debug)"
    << endl;
```

Termos técnicos

- Bioma – Ambiente/ecossistema da célula.