Relatório

Davidson dos Santos Dias
Mateus Fellipe Alves Lopes

6 de abril de 2016

Sumário

1 Introdução

O trabalho tem como objetivo apresentar uma introdução a programação orientada a objetos com conceitos de classes e suas instâncias, a utilização de classes pré-definidas pela linguagem, como, por exemplo, string, vector e ifstream, a utilização de constantes e alocação dinâmica de memória, etc.

A programação orientada a objetos surgiu com o principal objetivo de unir os dados e funções em um único elemento: o objeto. O objeto é um estado de uma classe, sendo uma classe é um tipo definido pelo usuário, semelhante a uma estrutura, com o adicional que funções também podem ser inseridas. Estas funções (métodos) vão agir sobre os dados (atributos) da classe.

2 Desenvolvimento

Especificações das questões propostas utilizando programação orientada a objetos.

2.1 *rot13*

A questão 2.1 insere conceitos de criptografia através da implementação do algoritmo rot13, utilizando a manipulação de arquivos para leitura e gravação em novos arquivos dos dados gerados pelo programa, aplicando funções da biblioteca **fstream**.

Propomos a solução deste problema com a utilização da tabela ASCII para a criptografia dos caracteres contidos nos arquivos a serem manipulados pelas funções **ifstream** e **ofstream**, sendo a função **ifstream** responsável em abrir e ler os arquivos e a **ofstream** pela criação e inserção de dados no arquivo desejado.

2.2 *vector<string>*

Na questão 2.2 fizemos a leitura de um arquivo utilizando a função **ifstream**, criamos **vector** de **strings** e armazenando cada palavra em uma posição do vector, em seguida usamos a função: **std::sort(v.begin(),v.end())** para ordenar seus elementos.

2.3 *CWindow*

Na questão 2.3 criamos uma classe **CWindow** para representar uma janela, utilizando objetos de classes distintas como atributos, **Canvas** e **Color**, a classe CWindow encapsula a classe Canvas que por sua vez contêm instâncias da classe Color.

2.3.1 *Color*

Na classe **Color** encapsulamos cores em formato **RGB**, para isso utilizamos uma *struct* com três inteiros, e uma string para armazenar o nome da cor. Para criarmos objetos dessa classe usamos uma estratégia para facilitar a visualização e evitar grandes quantidades de dados nos parâmetros das funções, convertemos uma *string* com o nome da cor para seu respectivo formato **RGB**, função essa nomeada de **colorir()**. Para simplificar e reduzir o código consideramos apenas oito cores distintas sendo elas: branco (255,255,255), azul (0,0,255), vermelho (255,0,0), verde (0,255,0), amarelo (255,255,0), magenta (255,0,255), ciano (0,255,255) e preto (0,0,0).

O **construtor** por *default* recebe uma *string* com a cor branca que inicializa o objeto invocando a função *colorir()* que converte a *string* para o formato RGB.

2.3.1.1 ***Métodos :*** Desenvolvemos o método **set_Color()** que recebe o nome de uma cor e recorre ao método **colorir()** para preencher a *struct RGB* com os valores característicos da cor. **Cor()** retornar a *string nome_cor* contida na *struct RGB*.

2.3.1.2 *Parâmetros*

- *Color(const string c="branco");*
- *void set_Color(const string c);*
- *string Cor() const;*
- *void colorir(const string c);*

2.3.2 *Canvas*

Classe aninhada que armazena atributos de desenho da janela: cor da fonte, tamanho da fonte, tipo de fonte, cor da pena, cor do pincel.

Construtor não possui um método *default* pois **Canvas** sempre será inicializado no **construtor** de **CWindow** com todos os valores.

2.3.2.1 *Métodos :* set_fonte(), set_pena(), set_pincel(), set_tamanho() e set_tipo_fonte()

são métodos que recebem por parâmetros *constantes* e alteram os atributos do objeto. Os métodos que alteram as instâncias da classe **Color** chamam o método *set_Color()* pois os atributos dela são restritos para a classe **Canvas**.

get_fonte(), **get_pena()**, **get_pincel()**, **get_tamanho()** e **get_tipo_fonte()** são métodos *constantes* que retornam os valores dos atributos do objeto. Os métodos para retornar fonte, pena e pincel invocam o método **Cor()** da classe **Color**.

2.3.2.2 *Parâmetros*

- *Canvas(const int tam_fonte, const string tip_fonte, const string font, const string pen, const string pince);*
- *void set_fonte(const string font);*
- *void set_pena(const string pen);*
- *void set_pincel(const string pince);*
- *void set_tamanho(const int tamanho);*
- *void set_tipo_fonte(const string tipo_font);*
- *string get_fonte()const;*
- *string get_pena()const;*
- *string get_pincel()const;*
- *int get_tamanho()const;*
- *string get_tipo_fonte()const.*

2.3.3 *CWindow*

A classe **CWindow** contém atributos que armazenam características da posição e dimensão da janela e um ponteiro do tipo **Canvas** com atributos de desenho da janela.

O **construtor** do **CWindow** irá alocar o ponteiro da classe **Canvas** passando valores para o **construtor** da classe aninhada.

2.3.3.1 *Métodos :*

Cset_fonte(), **Cset_pena()**, **Cset_pincel()**, **Cset_tamanho()** e **Cset_tipo_fonte()** são métodos que recebem por parâmetros *constantes* e alteram os atributos do objeto. Os métodos que alteram as instâncias da classe **Canvas** chamando seus respectivos métodos, pois os atributos de **Canvas** são restritos para a classe **CWindow**.

Cget_fonte(), **Cget_pena()**, **Cget_pincel()**, **Cget_tamanho()** e **Cget_tipo_fonte()** são métodos constantes que retornam os valores dos atributos do objeto. Os métodos para retornar invocam os métodos respectivos da classe **Canvas**.

move() e **resive()** recebem novas características da posição e dimensão da janela, por parâmetros *constantes*.

show() é um método *constante* que envia para a saída padrão a posição com largura e altura da janela.

LeDeArquivo() recebe por parâmetro um *string* como *constante*, fornecida pelo usuário, que indica o nome do arquivo a ser lido pela função **ifstream**, armazenando os valores na instancia da classe.

GravarArquivo() recebe por parâmetro um *string* como *constante*, fornecida pelo usuário, que indica o nome do arquivo a ser criado ou modificado pela função **ofstream**, armazenando os dados da **CWindow**.

Destrutor apenas desaloca o *ponteiro* da classe **Canvas** que foi alocado no **construtor** através da função **delete**.

2.3.3.2 *Parâmetros*

- *CWindow(const int newx=0,const int newy=0,const int newcx=10,const int newcy=10, const int tam_fonte=12, const string tip_fonte="Arial", const string font="preto", const string pen="preto", const string pince="preto");*

- *void show() const;*
- *void move(const int newx,const int newy);*
- *void resize(const int newcx,const int newcy);*
- *void Cset_fonte(const string font);*
- *void Cset_pena(const string pen);*
- *void Cset_pincel(const string pince);*
- *void Cset_tamanho(const int tamanho);*
- *void Cset_tipo_fonte(const string tipo_font);*
- *string Cget_fonte()const;*
- *string Cget_pena()const;*
- *string Cget_pincel()const;*
- *int Cget_tamanho()const;*
- *string Cget_tipo_fonte()const;*
- *void LeDeArquivo(const std::string nomeArquivo);*
- *void GravarArquivo(const std::string nomeArquivo).*

2.4 *CWindowSingleton*

De forma análoga a questão 2.3, na questão 2.4 alteramos a classe **CWindow** para o padrão *singleton*, sendo possível criar apenas uma instancia para a esta classe. Para isso colocamos o construtor como privado e utilizamos o método ***Instace()** de tipo *static* como o único ponto de acesso ao construtor assim limitando a quantidade de objetos criados e retornado o endereço do objeto criado, para conseguirmos identificar se já existe um objeto da classes usamos o ponteiro também de tipo *static*, ***instaceptr**, para armazenar o endereço da primeira janela criada e retornar este endereço caso seja solicitado a criação de mais uma janela assim garantido que apenas um objeto seja criado.

2.4.1 *Método e variavel static*

- *static CWindowSingleton *instaceptr;*
- *static CWindowSingleton *Instace();*

2.5 *Racional*

Na questão 2.5 utilizamos a definição de classes para realizar operações com números racionais, inicialmente criamos uma classe chamada **Racional** com atributos privados do tipo inteiro para representar o numerador e denominador, e um método também privado para manter os números racionais no formato irredutível, optamos por criá-la privada pois é um método utilizado apenas pela própria classe.

O **Construtor** por default inicializa o numerador e o denominador como 0 e 1 respectivamente, e tratamos o caso de entrado 0 para o denominador, pois isso geraria indeterminação, e usado o método de simplificação antes do número ser armazenado no objeto.

2.5.1 *Métodos*

subtrair(), **multiplicar()**, **dividir()**, todos estes métodos tem o mesmo princípio de funcionalidade, onde vão receber objetos como *constantes* por parâmetro e armazena resultado no formato irredutível em um terceiro objeto (objeto invocador da função).

O método **add()** define novos valores para o objeto a partir de dados fornecidos pelo usuário, por default o denominador recebe 1 para caso ele não seja especificado.

Os métodos de impressão são *constantes*, pois não ocorrerá modificação nos objetos, e são necessários pelo fato dos atributos serem privados a não termos acesso direto a eles.

2.5.2 *Parâmetros*

- *Racional(int n=0,int d=1);*
- *void add(const int n, int d=1);*
- *void subtrair(const Racional a ,const Racional b);*

- *void multiplicar(const Racional a,const Racional b);*
- *void dividir(const Racional a, const Racional b);*
- *void printflutuante()const ;*
- *void print()const;*

2.6 *IntegerSet*

Na questão 2.6 representamos os conjuntos com **arrays** do tipo **bool**, colocando **true** na posição que representa o valor contido no conjunto.
Ex: A[9] =True => 9 pertence ao conjunto.

O array está encapsulado na classe nomeada como **IntegerSet**, juntamente com seus métodos.
O **construtor** irá criar por default um conjunto vazio.

2.6.1 *Métodos*

Uniao() e **Intersecao()**, são métodos que recebem por parâmetro dois objetos como *constantes* e armazena o resultado em um terceiro objeto (objeto invocador da função).

Igual() é um método *constante* que recebe por parâmetro outro objeto também como *constante* que será comparado com o objeto que invocou a função.

InsereElemento() e **RemoveElemento()** inicialmente verificam se o valor fornecido é valido, posteriormente alteram o array de tamanho definido o valor na posição que representa o elemento do conjunto de false para true e true para false respectivamente.

LeDeArquivo() recebe por parâmetro uma string como **constante**, fornecida pelo usuário, que indica o nome do arquivo a ser lido pela função **ifstream**, armazenando todos os inteiros no objeto.

No programa para testar a classe usamos um *vector* para armazenar a quantidade de conjuntos desejado.

2.6.2 *Parâmetros*

- *InterSet();*

- *int InsereElemento(const int k);*
- *int RemoveElemento(const int k);*
- *void Uniao(const InterSet x,const InterSet y);*
- *void Intersecao(const InterSet x,const InterSet y);*
- *void Imprime() const ;*
- *int Igual(const InterSet x) const;*
- *void LeDeArquivo(const string nomeArquivo);*

2.7 *Lista simplesmente encadeada*

Na questão 2.7 criamos uma lista simplesmente encadeada usando os conceitos de classes, para isso encapsulamos uma **struct** de nome **Link** para representar um nó da lista, a struct contém seu próprio construtor e destrutor, que serão invocados na criação e exclusão de nó no momento que se deseja inserir/excluir um elemento da lista.

A classe **Lista_encadeada** tem como atributo o ponteiro da struct Link, que aponta para o início da lista. Por *default* o **construtor** irá apontar o ponteiro *Lista para 0.

2.7.1 *Métodos*

inserir() inicialmente faz uma busca pela lista procurando a posição que o elemento será inserido de forma ordenada, quando não repetido.

O método **remover()** assim como o **inserir()** faz uma busca pela lista a procura do elemento para exclui-lo.

procura() recebe por parâmetro o elemento que se deseja localizar e retorna a posição do elemento se ele estiver contido na lista.

LeDeArquivo() recebe por parâmetro um string como **constante**, fornecida pelo usuário, que indica o nome do arquivo a ser lido pela função **ifstream**, armazenando todos os inteiros na lista. **GravarArquivo()** recebe por parâmetro um string como **constante**, fornecida pelo usuário, que indica o nome do arquivo a ser criado ou modificado pela função **ofstream**, armazenando a lista no arquivo.

O **destrutor** da classe ao ser invocado (final da execução) precisa invocar também o destrutor da *struct* para que cada nó seja excluído separadamente.

2.7.2 *Parâmetros*

- *Lista_encadeada();*
- *void inserir(const int d);*
- *int procura(const int d)const;*
- *void remover(const int d);*
- *void imprimir()const;*
- *void LeDeArquivo(const string nomeArquivo);*
- *void GravarArquivo(const string nomeArquivo);*

2.8 *Matriz*

Resolvemos a questão 2.8 encapsulando uma matriz dinâmica do tipo **double** em uma classe chamada **Matriz**, que contém os atributos **linhas** e **colunas** do tipo **inteiro**, e um método privado **Alocar()** que aloca a memória de acordo com os valores de linha e coluna.

Existem dois **construtores** na classe:

- O primeiro recebe o número de linhas e colunas criando uma matriz com essas dimensões e inicializando os elementos da matriz com 0, e por *default* inicializa linha e coluna com 0, logo não aloca memória para a matriz, pois geraria conflito na modificação do mesmo posteriormente, já que a memória da matriz não poderia ser redefinida após a sua inicialização.

- O segundo recebe por parâmetro uma **string** como **constante**,indicando o nome de um arquivo, e inicializa a matriz a partir dos dados lidos do arquivo.

2.8.1 *Métodos*

somar() recebe dois objetos como **constantes** por parâmetro e armazena o resultado em um terceiro objeto (objeto invocador da função), retorna 0 caso as matrizes não coincidirem as dimensões.

multiplicavector() recebe um *vector* e um objeto do tipo *Matriz* como **constantes** por parâmetro e armazena o resultado em um terceiro objeto (objeto invocador da função).

grava() recebe por parâmetro a *referência* de uma *string* como **constante**, fornecida pelo usuário, que indica o nome do arquivo a ser criado ou modificado pela função **ofstream**, armazenando a matriz em disco.

inserir(),**imprimir()** e **size_coluna()** são métodos que foram criados com o intuito de facilitar na manipulação do objeto no programa.

O **destrutor** da classe ao ser invocado (final da execução) necessita desalocar a Matriz com o auxilio do destrutor da *struct* Link, caso contrário desalocaria apenas o ponteiro.

2.8.2 *Parâmetros*

- *Matriz(const int l=0,const int c=0);*
- *Matriz(const string nomeArquivo);*
- *void grava(const string& filename)const;*
- *int soma(const Matriz x, const Matriz y);*
- *void multiplicavector(const Matriz m, const vector<double> vec);*
- *void inserir();*
- *void imprimir() const;*
- *void size_coluna() const;*

3 Conclusão

Concluímos que o trabalho foi de essencial importância para colocarmos em pratica conceitos apresentados em sala de aula, conceitos de criação de classes e instâncias (objetos) que são a base para a programação orientada a objetos e de utilização de classes pré-definidas. A principal dificuldades que tivemos foi a adaptação com um novo método de programação (Orientado a objetos), já que estávamos familiarizados com a programação estruturada.

4 Bibliografia

1. Estivemos em discussão com a dupla de Jair Gomes e Kevin Jonas do 4º período de Engenharia de Sistemas, sobre os conceitos e usos de classes. Assim também sanamos duvidas com Juliana Miranda do 8º período de Engenharia de Sistemas.

2. *cplusplus.com*

3. *programacaodescomplicada.wordpress.com*/2012/11/09/*aula*-64-*alocacao-dinamica-pt*-6-*alocacao-de-matrizes*

4. *www-usr.inf.ufsm.br/pozzer/disciplinas/cg_5_oo_c++.pdf*

5. *ic.unicamp.br/ everton/aulas/hardware/tabelaASCII.pdf*

6. *hex.com.br/rot*13

7. *www.uff.br/cdme/matrix/matrix-html/matrix_color_cube /matrix_color_cube_br.html*