Relatório III

Davidson dos Santos Dias
Mateus Fellipe Alves Lopes

6 de novembro de 2015

Sumário

1 Introdução

O trabalho tem como objetivo a criação de classes completas, envolvendo diversos tipos de **constutores**, **destrutores**, **atributos dinâmicos** e **sobrecarga de operadores** em C++, além da utilizaçaõ de mecanismos como ***namespaces***, **constantes**, **funções *inline***, **gerenciamento dinâmico de memória**, etc;

As sobrecarga de operadores consiste na redefinição de itens já existentes, permitindo que sejam definidas duas ou mais funções com o mesmo nome, desde que suas listas de argumentos sejam diferentes para que não haja comflito sobre qual função deve ser chamanda. Os *namespace* são agrupamentos lógicos dos métodos, assim é possível a criação de funções de mesmos nomes e parâmetros, definido-os em *namespaces* distintos.

2 Desenvolvimento

Especificações das questões propostas utilizando programação orientada a objetos.

2.1 *CVetor*

Na questão 2.1 criamos um vetor usando os conceitos de classes, para isso encapsulamos um tamanho e um ponteiro para representar o tamanho do vetor e o vetor que será alocado a partir desse tamanho.

Por *default* o **construtor** irá inicializar o tamanho em 0, assim não alocando nenhuma posição para o vetor.Toda a classe foi incluida em um *namespace* **DataStrutures**.

O **construtor de cópia** recebe como parâmetro um objeto também da classe CVetor, copiando todos os atributos e elementos alocados dinamicamente do objeto-parâmetro corretamente.

2.1.1 *Métodos e Operadores*

Operadores sobrecarregados como função membro:

- O operador de atribuição = recebe um objeto como uma **referência constante** copiando os valores dos atributos alocados dinamicamente

de forma correta.

- Os operadores () e [] recebem um inteiro **constante** para indicar qual posição a ser acessada no vetor, retornando por **referência** o dado contido naquela posição.

- Os operadores + e - utilizam dois vetores para somar/subtrair recebebendo um objeto do tipo *CVetor* como **referências constantes**. Os resultados são armazenados em um objeto auxiliar que será retornado por valor no final da operação.

Operadores sobrecarregados como funções ***friend***:

- Os operadores de fluxo de entrada e saida(« e ») são utilizados para inserir/imprimir elementos do vetor a partir do tamanho inicialmente fornecido.

Os métodos **escalar()** e **vetorial_3D()** recebem um objeto da classe CVetor como **referência constante**.O primeiro retorna um inteiro representando o produto escalar entre os dois vetores, e o segundo foi restrito a vetores de 3 dimensões, sendo inviavel a representação com mais dimenções, e retorna um novo vetor por valor.

O **destrutor** da classe ao ser invocado (final da execução) verifica se foi feita alguma alocação do vetor para que não haja problema no uso do *delete[]* para posições não alocadas.

2.1.2 *Parâmetros*

- `CVetor(const CVetor& h);`
- `CVetor(const int n=1);`
- `CVetor& operator = (const CVetor& h);`
- `CVetor operator + (const CVetor& h)const;`
- `CVetor operator - (const CVetor& h)const;`
- `int& operator [] (const int)const;`
- `int& operator () (const int)const;`
- `int escalar(const CVetor&) const;`

- `CVetor vetorial_3D(const CVetor&) const;`
- `friend std::ostream& operator « (std::ostream& op, const CVetor& h);`
- `friend std::istream& operator » (std::istream& op, CVetor& h).`

2.2 *Matriz*

Resolvemos a questão 2.2 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.Toda a classe foi incluida em um *namespace* ***Matematica***.

Existem 2 **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 é o **construtor de cópia** que recebe como parâmetro um objeto também da classe matriz, copiando todos os atributos e elementos alocados dinamicamente do objeto-parâmetro corretamente.

2.2.1 *Métodos e Operadores*

Operadores sobrecarregados como função membro:

- O operador de atribuição = recebe um objeto como uma **referência constante** copiando os valores dos atributos alocados dinamicamente de forma correta.
- Os operadores +, **-** e ***** recebe um objeto como uma **referência constante** por parâmetro e armazena o resultado em um objeto auxiliar que será retornado por valor no final da operação.
- Os operadores +=, -= e *= recebem um objeto como uma **referência constante** por parâmetro e armazena o resultado no próprio objeto que será retornado por referência no final da operação através do ponteiro *this*.

- O operador() recebe dois inteiros **constantes** para indicar qual posição a ser acessada na Matriz(linha, coluna),retornando por **referência** o dado contido naquela posição.

- Os operadores * e *= recebe um objeto do tipo *CVetor* ou um *vector* como **referências constantes** por parâmetro. O resultado do operador * é armazenado em um objeto auxiliar que será retornado por valor no final da operação. E o resultado do operador *= é armazenado no proprio objeto que será retornado por referência no final da operação através do ponteiro *this*.

- Os operadores « e » recebem uma **string** como **referência constante**, indicando o nome de um arquivo, para gravar dados em aquivo e ler os dados de arquivo, respectivamente.

Operadores sobrecarregados como funções ***friend***:

- Os operadores de fluxo de entrada e saída (« e ») por serem *friend* tem acesso aos atributos da classe Matriz, facilitando a inserção e impressão de dados no objeto.

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.

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.

2.2.2 *Parâmetros*

- `Matriz(const Matriz& h);`
- `Matriz(const int l=0,const int c=0);`
- `Matriz& operator=(const Matriz& );`
- `Matriz operator+(const Matriz& )const;`
- `Matriz operator-(const Matriz& )const;`
- `Matriz& operator+=(const Matriz& );`
- `Matriz& operator-=(const Matriz& );`

- `Matriz operator*(const Matriz& )const;`
- `Matriz& operator*=(const Matriz& );`
- `Matriz operator*(const std::vector<double>& )const;`
- `DataStrutures::CVetor operator*(const DataStrutures::CVetor& vet)const;`
- `Matriz& operator*=(const DataStrutures::CVetor& vet);`
- `Matri& operator*=(const std::vector<double>& );`
- `double& operator()(const int i,const int j)const;`
- `Matriz& operator » (const std::string& nomeArquivo);`
- `Matriz& operator « (const std::string& nomeArquivo)const;`
- `int size_coluna() const;`
- `friend std::ostream& operator « (std::ostream& op, const Matriz&);`
- `friend std::istream& operator » (std::istream& op, Matriz&).`

2.3 *Racional*

Na questão 2.3 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.Toda a classe foi incluida em um *namespace* ***Matematica***.

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.

2.3.1 *Métodos e Operadores*

Os operadores +, -, * e / foram sobrecarregados como função membro recebendo um objeto como uma **referência constante** por parâmetro e armazena o resultado em um objeto auxiliar que será retornado por valor no final da operação.

O operador de fluxo de saída («) foi sobrecarregado como função *friend* para realizar a impressão de dados no objeto no formato a/b.

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.

Foram criadas dois conversores:

- ***double*** para **Racional**-> fornecido o número flutuante é feito uma conversão, para números indefinidos foi feito uma aproximação.
- **Racional** para ***double***-> retorna a divisão do numerador e denominador para uma variável ***double***.

2.3.2 *Parâmetros*

- `Racional(const int n=0, int d=1);`
- `Racional(const double d);`
- `void add(const int n, int d=1);`
- `Racional operator - (const Racional& )const;`
- `Racional operator + (const Racional& )const;`
- `Racional operator * (const Racional& )const;`
- `Racional operator / (const Racional& )const;`
- `operator double()const;`
- `friend std::ostream& operator « (std::ostream& op, const Racional&);`

- `void printflutuante()const.`

2.4 *IntegerSet*

Na questão 2.4 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* de *bool* foi ultilizado para facilitar algumas operações, utilizando métodos pré-definidos na classe *array*.
Toda a classe foi incluida em um *namespace* ***Matematica***.

O **construtor** irá criar por default um conjunto vazio.

2.4.1 *Métodos e Operadores*

Operadores sobrecarregados como função membro que recebem por parametro objetos com **referência constante**:

- Os operadores +(União) e **&**(Interseção) armazenam os resultadso em um objeto auxiliar que será retornado por valor no final da operação.
- O operador == retorna um valor do tipo bool indicado **true** caso os dois conjuntos sejam iguais e **false** caso contrário.
- Os operadores += e **-**= inserem e removem elementos do conjunto respectivamente, inicialmente em cada sobrecarga 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.
- O operador » recebe uma **string**, indicando o nome de um arquivo, para ler os dados do respectivos arquivo.

Operadores sobrecarregados como funções ***friend***:

- Os operadores de fluxo de entrada e saída (« e ») foram sobrecarregados como funções *friend* para que tenha acesso aos atributos da classe IntegerSet, facilitando a inserção e impressão de dados no objeto.

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

2.4.2 *Parâmetros*

- `IntegerSet();`
- `IntegerSet operator + (const IntegerSet& )const;`
- `IntegerSet operator & (const IntegerSet& )const;`
- `IntegerSet& operator += (const int );`
- `IntegerSet& operator -= (const int );`
- `IntegerSet& operator » (const std::string& nomeArquivo);`
- `friend std::ostream& operator « (std::ostream& op, const IntegerSet&);`
- `friend std::istream& operator » (std::istream& op, IntegerSet&);`
- `bool operator == (const IntegerSet& x) const.`

2.5 *CWindow*

Na questão 2.5 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.

2.5.1 *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.

2.5.1.1 ***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*.

2.5.1.2 *Parâmetros*

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

2.5.2 *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.

2.5.2.1 ***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**.

2.5.2.2 *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.`

2.5.3 *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. Foram criadas funções testes para testa o construtor de copia e o operador de atribuição.

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

O **construtor de cópia** recebe como parâmetro um objeto também da classe CWindow, copiando todos os atributos e elementos alocados dinamicamente do objeto-parâmetro corretamente.

2.5.3.1 ***Métodos e Operadores:*** O operador de atribuição '=' foi sobrecarregado como função membro recebendo um objeto como uma **referência constante** por parâmetro copiado os valores dos atributos alocados dinamicamente de forma correta.

Os operadores de fluxo de entrada e saída (« e ») foram sobrecarregados como funções *friend* para que tenha acesso aos atributos da classe CWindow, facilitando a inserção e impressão de dados do objeto.

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**.

2.5.3.2 *Parâmetros*

- `Window(const CWindow& h);`
- `CWindow& operator= (const CWindow& h);`
- `friend std::ostream& operator « (std::ostream& op, const CWindow&);`
- `friend std::istream& operator » (std::istream& op, CWindow&);`
- `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).`

2.5.4 *CWindowSingleton*

Na classe **CWindowSingleton** alteramos a classe **CWindow** para o padrão *singleton*, sendo possível criar apenas uma instancia para 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**, com o intuito de 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. Para manter a unicidade da classe *CWindowSingleton* o construtor de cópia e o operador de atribuição foram colocados como privado evintando assim que uma cópia seja criando em qualquer escopo.

Método e variavel static:

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

2.6 *Lista simplesmente encadeada*

Na questão 2.6 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. Toda a classe foi incluida em um *namespace* **DataStrutures**.Além do mais foram criadas funções testes para o construtor de copia e o operador de atribuição.

O **construtor de cópia** recebe como parâmetro um objeto também da classe **Lista_encadeada**, copiando todos os atributos e elementos alocados dinamicamente do objeto-parâmetro corretamente.

2.6.1 *Métodos e Operadores*

Operadores sobrecarregados como função membro que recebem por parametro objetos **constantes**:

- O operador de atribuição = recebe a **referência** de um objeto copiando os valores dos atributos alocados dinamicamente de forma correta.
- Os operadores += e **-**= recebem um inteiro para inserir/remover elementos da lista, armazenando o resultado no próprio objeto que será retornado por referência no final da operação através do ponteiro *this*.
- O operador() recebe um inteiro para indicar qual posição a ser acessada na Lista, retornando por **referência** o dado contido naquela posição.
- Os operadores + e += foram criados com o propósito de concatenar duas listas recebebendo um objeto do tipo *Lista_encadeada* por **referências**. O resultado do operador + é armazenado em um objeto auxiliar que será retornado por valor no final da operação. E o resultado do operador += é armazenado no próprio objeto que será retornado por referência no final da operação através do ponteiro *this*.

Operadores sobrecarregados como funções ***friend***:

- Os operadores de fluxo de entrada e saída (« e ») foram criadas para que haja acesso aos atributos da classe Lista_encadeada, facilitando a inserção e impressão de dados no objeto.

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.

2.6.2 *Parâmetros*

- `Lista_encadeada(const Lista_encadeada& h);`
- `Lista_encadeada& operator = (const Lista_encadeada& );`
- `Lista_encadeada operator + (const Lista_encadeada& ) const;`
- `Lista_encadeada& operator += (const Lista_encadeada& );`
- `Lista_encadeada& operator +=(const int d);`
- `Lista_encadeada& operator -= (const int d);`
- `int operator () (const int );`
- `int size()const;`
- `Lista_encadeada();`
- `void LeDeArquivo(const std::string& nomeArquivo);`
- `void GravarArquivo(const std::string& nomeArquivo);`
- `friend std::ostream& operator « (std::ostream& op, const Lista_encadeada& h);`
- `friend std::istream& operator » (std::istream& op, Lista_encadeada&).`

2.7 *Classe Derivada*

Na questão 2.7 usamos o conceito de herança para fazermos a derivação de **C** em relação a **A**, o fato de **C** ser derivado **A** torna possivel a classe **C** ser construida a partir do construtor de **A**, demonstramos isso a partir da inesistência do construtor de **C** e a utilização do *cout* nos contrutores e destrutores das classes envolvidas (A,B,C).

3 Conclusão

Concluímos que o trabalho foi de essencial importância para colocarmos em pratica conceitos apresentados em sala de aula, conceitos de sobrecarga de operadores como funções membro e funções não-membro, construtor de cópia onde e tratado a de atributos dinamicos. Uma das principais dificuldades foi o tratamento das exceções para as funções de retorno, em alguns casos foi retornado um objeto vazio, o que significa a não execução do método, isso quando é retorno por valor, para os demais casos só será executado o método quando o mesmo for possível.

4 Bibliografia

1. Estivemos em discussão com a dupla de Jair Gomes e Kevin Jonas, e de Cristiano Antunes do 4º período de Engenharia de Sistemas, sobre os conceitos e usos de sobrecarga de operadores.

2. *cplusplus.com*