Relatório IV

Davidson dos Santos Dias
Mateus Fellipe Alves Lopes

6 de janeiro de 2016

Sumário

1 Introdução

O trabalho tem como objetivos a criação de **hierarquias simples de classes** e a sua programação em C++, incluindo a utilização de mecanismos de **herança**, **composição**, **polimorfismo** e **classes abstratas**, além da utilização de outros mecanismos da linguagem C++, como **templates**, a **biblioteca STL** e o **lançamento de exceções**.

A **herança** e **composição** são um dos mecanismos mais caracteristicos da programação orientada a objetos, visto que com o uso correto deste é possivel derivar um classe de um classe(pai) , tornando a classe derivada um tipo da classe herdada, contendo na classe filha todos os metódos publicos e/ou protegidos da classe base (herança publica, privada ou protegida), criando assim inúmeros meios para resolver problemas no meio da programação, viabilizando ainda mais a POO.

2 Desenvolvimento

Especificações das questões propostas utilizando programação orientada a objetos.

2.1 *Sequências*

Na questão 2.1 criamos uma classe **Seq** com um *vector* que a principio deveria ser *static* mas não conseguimos implementar algum método que funcionasse o *static*, testamos pelo método colocando **Seq** como um *template* mas esbarramos no problema em que não conseguimos criar um ponteiro da classe **Seq** porque ela era um *template* e necessitava de ser passado um tipo, tentamos também implementar um ponteiro do *vector* mas a cada alocação o conteúdo do ponteiro era subscrevido. Optamos então por mandar o algoritmo que estava funcionando corretamente sem o *static* junto com o algoritmo com o método utilizando o ponteiro para demonstrarmos o erro que estava acontecendo.

Criamos então um *vector* do tipo *unsigned long int* privado na classe **Seq** e métodos para setar o vetor, inserindo elementos no *vector*, limpar o *vector*, retornar o elemento naquela posição, método para imprimir a sequência e operador de fluxo de saida. Os métodos para *set*, *get*, *clear* e para gerar elementos da sequência foram colocados como protegidos para que na herança

eles pudessem ser utilizados pelas classes filhas. Além de tornar a classe abstrata para que ela não pudesse ser instânciada.

Herdamos 6 classes, **Fibonacci, Lucas, Pell, Triangular, Quadrados e Pentagonal**, da classe **Seq** e cada uma com um tipo diferente para gerar sua sequência e armazenar-la no *vector* da classe base, e como elas foram herdadas da classe **Seq** todas eram manipuladas da mesma maneira somente com a sequências distintas.

A classe **container** foi criada para armazernar qualquer tipo de **Seq** e por ele é possivel imprimir toda a sequência armazenada.

Segue hierarquia das classes e seus métodos:

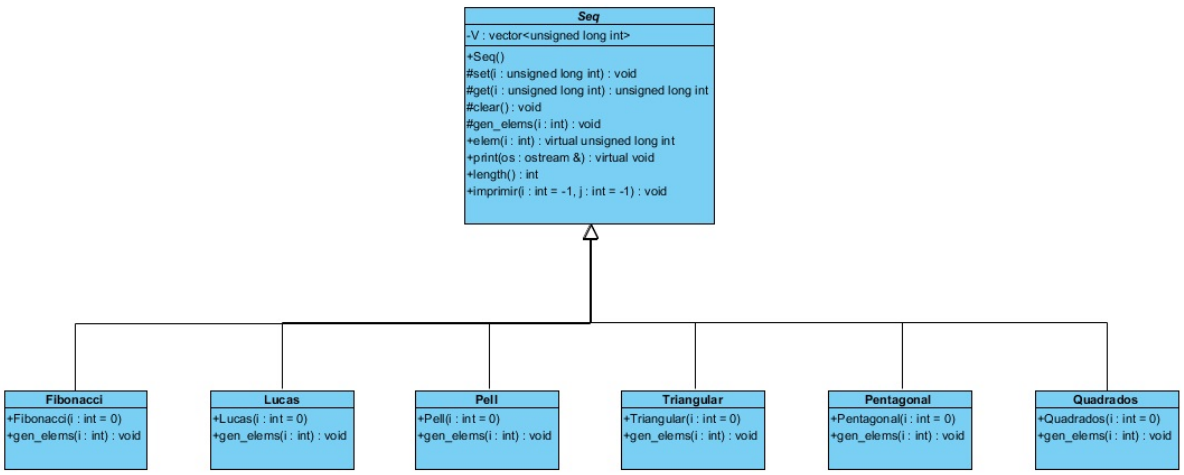

Figura 1: Diagrama

2.1.1 *Parâmetros - Seq*

- virtual unsigned long int elem(int i);
- virtual void print(std::ostream& os);
- virtual int length();
- virtual void imprimi(int i=-1,int j=-1);

2.2 *Matriz*

Na questão 2.2 modificamos a classe **Matriz**, já implementada nos trabalhos anteriores, para que a classe **Matriz** se tornasse genérica, usando o mecanismo de *template*, é acrescentamos também o conceito de *iterator*, criando uma nova classe alinhada em **Matriz**. Para criar uma matriz de um tipo específico, é preciso mudar o parâmetro nas as ocorrência da declaração da Matriz.

Na classe alinhada **iterator**, sobrecarregamos alguns operadores necessários, tais como:

- O operador de atribuição;
- Operator* que retorna objeto que o *iterator* está apontado no instante;
- Operator++ para incrementar o *iterator*, de forma que ira percorrer a matriz linha por linha;
- Operator==, Operator!= e Operator> que retornam true caso os *iterators* sejam iguais, diferentes e maior que, respectivamente.
- E os métodos **begin()** e **end()**, que retornem *iterators* para o início e um elemento após o final da matriz.

A **it_maior()** é uma função*template* que recebe como parâmetro dois *iterators*, e retorna o *iterator* que aponta para o maior valor.

2.3 *Pesquisa em lista telefônica*

Na questão 2.3 implementamos um programa para fazer uma pesquisa em uma dada lista telefônica onde o usuário fornece um nome ou um número de telefone e o programa fornece os outros dados correspondentes. Para criação deste programa foi sugerido o uso do **map** e do **multimap** inclusos na biblioteca <map.h>.

Na primeira opção de pesquisa, por nome, foi utilizada a estrutura do multimap, pois foi considerado o caso de um mesmo assinante ter mais de um endereço ou telefone, para constar um novo endereço ou telefone para o mesmo assinante deve se manter o padrão no arquivo do conjunto de três linhas para cada assinante, longo para o inserir outro endereço deve preencher todos os campos novamente. Assim usamos o nome como chave para uma *struct* com os dados do assinante, nome e endereço.

Na segunda opção de pesquisa, por telefone, foi utilizado a estrutura do *map*, onde usamos o número de telefone como chave de acesso para uma *struct* com os dados do assinante, nome e endereço. O operador de fluxo de saída « foi sobrecarregado para a impressão das duas *structs* distintas presentes no programa.

2.4 *Racional*

Na questão 2.4 utilizamos a classe Racional implementada nos trabalhos anteriores, e acrescentamos outros métodos, tais como: sobrecarga de operadores e *function object*.

O operador de fluxo de entrada » foi sobrecarregado para a efetuar a leitura de números na forma racional. O *function object* foi implementado e usado para fazer a comparação entre os objetos da classe Racional, e posteriormente usado para ordenação dos objetos, o *function object* é constituído por uma *struct* com um método que retorna verdadeiro ou falso para uma comparação entre os objetos tipo Racional. A **maior_menor()** retorna verdadeiro se o primeiro objeto for maior que o segundo, e a **menor_maior()** retorna verdadeiro se o primeiro objeto for menor que o segundo.

Para a gerenciamento do fluxo de entrada e saída usamos iterators:

- O *istream_iterator* é um iterator de entrada, que lê elementos sequencialmente até um estado especial que indica o fim da entrada de elemento ou falha. O estado especial é representado especificadamente por *eos*, é atribuímos um objeto do tipo ifstream a um objeto do tipo *istream_iterator* para fazer a leitura de arquivo. itemize
- O *ostream_iterator* é um iterator de saída, que escreve elementos sequencialmente, sempre que o operador de **(=)** é utilizado um novo elemento e inserido no *stream*, por isso o uso da função **copy()**, que irá copiar todo o *container* e posteriormente imprimi-lo. Armazenamos os objetos Racionais em um *set*, e para isso fizemos uso da *function object*, que também e passada com parâmetro para o *set*, assim possibilitando a ordenação interna através da comparação feita na *function object*.

2.5 Lista de Adjacência

Na questão 2.5 representamos grafos com o conceito de Lista de adjacência, para isso usamos criamos as classes *Graph* e *Edge*.

2.5.1 *Edge*

A classe *Edge* representa as arestas onde encapsula dois inteiros para simbolizar o par de vértices adjacentes, e métodos para setar e retornar algum dos vértices, e por *default* ou vértice invalido os vértices são apontados para 0, pois a posição zero não é utilizada.

2.5.2 *Parâmetros - Edge*

- Edge(const int a=0,const int b=0);
- int get_inicio()const;
- int get_fim()const;
- void set(const int a,const int b);
- virtual Edge();

2.5.3 *Graph*

A classe *Graph* representa o grafo encapsulando um *vector* do tipo *List*, o *List* usado foi o implementado por nós nos trabalhos anteriores, com o uso desta estrutura facilitamos a implementação da classe *Graph*, tais como a inserção ordenada, tratamento de repetições, procura de elementos, entre outros, e também encapsulado inteiros para a contagem de vértices e arestas. Para essa representação de grafo não consideramos a posição zero no *vector*, assim a contagem dos vértices é iniciada pelo primeira posição.

2.5.4 *Parâmetros - Graph*

Insert() e **remove()** são métodos para inserção e remoção de arestas no grafo, onde retorna **true** se inserir ou remover corretamente, e **false** caso contrario, recebendo um objeto do tipo *Edge* como referência constante.

get_vertices() e **get_arestas()** retornam a quantidade de vértices e arestas respectivamente.

coonnectedComponentes() retorna o número de subgrafos existentes no grafo, para isso usamos também um outro método privado **DFS()** que faz uma busca em profundidade na lista de adjacência, como foi dito que o grafo será não-direcionado podemos fazer a contagem pela quantidade de vezes que o método **DFS()** é chamado.

edge() retorna true quando a *Edge* passada por parâmetro estiver presente no grafo e **false** caso contrario.

O operador de fluxo de saída (**«**) foi sobrecarregado de forma que todo o grafo seja impresso por completo.

O construtor recebe um inteiro que indica a quantidade de vértices do grafo e a partir desse valor e feita a alocação de memoria no *vector* e por *default* é alocado apenas um vértice, não considerando o zero. No destrutor é feita a desalocação do vector.

3 Conclusão

Concluímos que o trabalho foi de essencial importância para colocarmos em pratica conceitos apresentados em sala de aula, conceitos de hierarquias de classes. Além de termos que estudar e pesquisar sobre assuntos que não foram abordados em sala, exigindo mais empenho para entendimento do assunto. Uma das principais dificuldades foi na questão 2.1 das sequências onde não conseguimos implementar o *static* na classe base mesmo tentando implementar alguns métodos pesquisados e tendo ajuda dos colegas e do professor. Do mais o trabalho foi de grandíssima importância para o encerramento de toda a matéria da disciplina, nos deixando com uma base firme para que possamos posteriormente aprofundarmos sem grande dificuldade em áreas afins.

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 de polimorfismos, *template* e *iterator*.

2. *cplusplus.com*

3. $http://www.yolinux.com/TUTORIALS/CppStlMultiMap.html$

4. $http://stackoverflow.com/questions/12796580/static-variable-for-each-derived-class$

5. *Livro: Thinking in C++ Volume.1.SE-2000*

6. *Livro: Conceitos de computação com o essencial de C++, Cay Horstman, 3º edição*