UNIVERSIDADE FEDERAL DE OURO PRETO - UFOP
DEPARTAMENTO DE COMPUTAÇÃO - DECOM
PROGRAMAÇÃO ORIENTADA A OBJETOS - BCC221

David Souza do Nascimento - 19.2.4029
Gabriel Ferreira Pereira - 20.1.4015
Lavínia Fonseca Pereira - 19.1.4170
Pablo Martins Coelho - 20.1.4113

TRABALHO PRÁTICO DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Ouro Preto - MG
2021

1. Introdução

O presente relatório, tem por finalidade se tratar da implementação do código voltado para uma corretora de imóveis, essa corretora terá um sistema que gerencia imóveis, essa implementação é baseada na Standard Templates Library (STL).

Escolhemos os métodos utilizados através das funcionalidades necessárias para cada um, o contexto e os algoritmos responsáveis pela manipulação, adequando cada parte do código a sua respectiva necessidade e função.

Com esse trabalho foi possível implementar um código eficaz e entender o proposto a cada parte do código criado, adequando-o em todos os quesitos para que ele realize o que lhe foi proposto. Através dessa implementação, conseguimos aumentar nosso conhecimento em STL e seus adendos, sem contar o aprimoramento de nossos conhecimentos.

2. Desenvolvimento

2.1. Contêiners

No código estão presentes os contêineres vector e map, o contêiner vector é um tipo de contêiner sequencial, baseado em arrays. Os vetores são iguais aos arrays dinâmicos com a capacidade de se redimensionar automaticamente quando um elemento é inserido ou excluído, com seu armazenamento sendo controlado automaticamente pelo contêiner. Os elementos do vetor são colocados em um armazenamento contíguo para que possam ser acessados e percorridos usando iteradores. Nos vetores, os dados são inseridos no final. A inserção no final leva um tempo diferencial, pois às vezes pode haver a necessidade de estender o array. A remoção do último elemento leva apenas um tempo constante porque não ocorre nenhum redimensionamento. Inserir e apagar no início ou no meio é linear no tempo. Já o contêiner map é um contêiner associativo. Um map, às vezes chamado de dicionário, consiste em um par chave/valor. A chave é usada para ordenar a sequência e o valor é associado a essa chave, ou seja, cada elemento possui um valor-chave e um valor mapeado. Dois valores mapeados não podem ter os mesmos valores-chave.

2.2. Imóvel

Na classe imóvel tem-se algumas variáveis que são comuns a todos os imóveis tratados no código, ou seja, todos os imóveis utilizam dessa classe, essas variáveis são o id do imóvel, o valor do mesmo, o nome do proprietário, a localização em que o imóvel se encontra, que é a rua, o bairro e a cidade, tem também o número do imóvel, a quantidade de banheiros e a quantidade de quartos. Essa classe é chamada de classe base, pois ela tem as operações de herança pública, pois todos os imóveis compartilham dessas informações. Nas funções do imóvel, encontram-se os gets e os sets, sendo que os seters têm a mesma finalidade nas funções, que é definir o id, o valor, o proprietário, a rua, o

bairro, a cidade, o número, a quantidade de quartos e banheiros de todos os imóveis presentes no código e as funções de geters têm o objetivo de pegar esses mesmos valores, sendo assim, com get defini e com set retorna o valor.

É possível também notar a presença da função amiga, friend em uma das funções e também a função virtual, a friend pode melhorar a performance da aplicação e é utilizada na sobrecarga de operadores e na criação de iteradores e a virtual é uma função que é redefinida pela classe derivada, a redefinição da função na classe derivada sobrepõe a definição da função na classe base. Essa implementação traz consigo, como já foi dito, as funções e variáveis comuns a todos os imóveis, sendo a classe base, que tem como método a herança pública, onde outras classes podem herdar dessa implementação.

Para poder utilizar o operador “<<” para os objetos da classe Imóveis, foi implementado uma função amiga da classe “ostream” que permite que seja feito overload no operador “<<”. No entanto, essa implementação ainda possui um problema na questão de polimorfismo. Caso o operador “<<” seja utilizado em um ponteiro para um objeto de uma subclasse de Imóvel, ele irá invocar o método da classe base em vez da subclasse. Para resolver este problema, foi utilizado um método virtual *print* que formata o output, e como o método é virtual, as subclasses podem sobrescrever o mesmo para especializar a saída de cada tipo.

2.3. Apartamento

Começando pelo construtor *Apartamento*, ele possui os parâmetros de entrada próprios e também os parâmetros de entrada com relações da classe Imóvel. As variáveis presentes nesse construtor são andar, que serve para determinar qual o andar do apartamento, a variável taxa_condominio, que determina a taxa de condomínio do apartamento a ser paga, tem também as variáveis elevador e sacada que servem para verificar se o apartamento possui elevador ou se ele possui sacada.

No método para realizar a impressão na tela das informações do apartamento se encontra as respectivas informações: nome do proprietário, valor do apartamento, quantidade de quartos, localização do apartamento que é a rua, o bairro e a cidade, por fim é impresso se o apartamento possui ou não possui elevador.

Nos métodos do apartamento se encontra a função getAndar, que é responsável por retornar qual o andar em que se encontra o apartamento, a função setAndar, que é responsável por definir o valor da variável andar, a função getTaxaCondominio é responsável por retornar qual a taxa de condomínio que deve ser paga, a setTaxaCondominio serve para definir a taxa de condomínio, as demais funções, getElevador, setElevador, getSacada e setSacada servem para retornar se o apartamento tem os respectivos adendos e para definir se eles tem ou não. Em geral as funções que iniciam com get fazem o retorno da informação e as funções com início set definem o valor das variáveis.

2.4. Casa

O construtor *casa* tem como parâmetros de entrada próprios além dos parâmetros de entrada da classe imóvel. Suas variáveis são *andares*, que nos informa quantos andares possuem a casa, sala_jantar que nos informa se a casa possui ou não uma sala de jantar.
Além dessas variáveis também temos as que estão presentes na classe imóvel, que nos informa no geral o valor, o proprietário, a rua, entre outros atributos mencionados no tópico imóvel. Ela também possui seus respectivos geters e seters para utilização durante a implementação do código.

Também possui o método para realizar a impressão na tela das informações do imóvel: nome do proprietário, valor do apartamento, quantidade de quartos, localização do apartamento que é a rua, o bairro e a cidade, por fim é impresso o número de andares que a casa possui.

2.5. Chácara

O construtor chacara, como em outros construtores possui além dos parâmetros de entrada da classe imóvel, seus próprios parâmetros que são: salão_de_festa*,* salao_jogos*,* campo_futebol*, churrasqueira* e *piscina* que nos informa se a chácara possui esses respectivos adendos. Essas variáveis são todas do tipo booleanas, pois basta nos informar se o atributo é verdadeiro ou falso.

Seus Geters e Seters são referentes às suas variáveis, mas também possui acesso aos geters e seters da classe imóvel. Possui também o método para realizar a impressão na tela das informações do imóvel: nome do proprietário, valor do apartamento, quantidade de quartos, localização do apartamento que é a rua, o bairro e a cidade, por fim é impresso se a chácara possui piscina.

2.6. Main

O main é constituído por uma coleção polimórfica de ponteiros de imóveis. Essa coleção é inicializada através do método loadFile, que percorre as linhas do arquivo, criando objetos das subclasses de Imóvel e adicionando-os no vetor. Cada linha é processada utilizando a função *processString*, que processa a linha lida do arquivo, removendo os “;” e transforma essa linha em um vetor de strings. Após a inserção, algumas variáveis são estabelecidas para poderem ser utilizadas como parâmetro para as funções de filtro que foram implementadas para extrair informações da coleção de Imóveis. Por último, a função *displayImovel* é o método de saída utilizado, caso o modo de saída correspondido por um inteiro de entrada da função seja 0 a saída será pelo terminal, caso for 1 a saída será por um arquivo texto.

2.7. Filtros

No arquivo main são encontrados alguns filtros, que serão responsáveis por filtrar os dados dos imóveis que serão retornados ao usuário.

Começando pelo *checkProprietario* essa função usa como principal parâmetro o atributo proprietário, em seu interno ela faz a comparação da entrada fornecida pelo usuário com todos os proprietários presentes dentro da classe imóvel. Essa comparação é feita pegando cada imóvel do vetor do tipo imóvel. No final ela retorna true para caso o proprietário seja encontrado e false para caso não seja.

Depois temos a função *filterValor* que tem como atributo principal o valor do imóvel, nesta função como as outras tem o principal intuito ir comparando valor por valor de cada imóvel, até que encontremos um valor que seja menor ou igual ao desejado pelo usuário.
Depois que satisfeita essa condição o usuário tem como retorno as características do imóvel.

Na função *filterTipo* é verificado qual o tipo de imóvel o usuário deseja buscar, se é um apartamento, casa ou chácara. Para isso iremos passar imóvel por imóvel fazendo a verificação, por isso usamos o size para ter como retorno o tamanho do vetor de imóveis. Logo depois teremos 3 comparações, nessas comparações 2 condições devem ser satisfeitas, a primeira é com base na entrada do usuário, verificaremos se o imóvel que está sendo verificado atualmente condiz com o tipo requisitado pelo parâmetro. Se sim iremos verificar a segunda condição que verifica o tipo do imóvel, utilizando da função *dynamic_cast*. Quando ambas as condições estiverem satisfeitas iremos retornar para o usuário o imóvel correspondente. E por último temos uma ordenação, para retornar os imóveis ordenados com base no valor para o usuário. Essa ordenação é feita utilizando o método sort da STL. Para realizar as comparações de ordenação foi passado uma função lambda que define como os objetos devem ser comparados.

A função *filterCidade* irá funcionar como as outras, iremos ter a entrada do usuário nos informando qual cidade deseja procurar, depois iremos fazer a comparação imóvel por imóvel e retornar ao usuário os imóveis presentes naquela cidade, porém esse retorno será ordenado com base nos valores, da mesma forma que o método anterior (com a diferença de que um está sendo ordenado de maneira crescente e o outro decrescente.

Na função *filterProprietario* tem como principal parâmetro o atributo proprietário, essa função diferente da *checkProprietario* retorna para o usuário um iterador para cada tipo de imóvel que o proprietário possua. Nesta função é utilizado um for, que vai do começo do vetor de imóveis até o final. Neste for iremos comparar a informação do proprietário fornecida pelo usuário com o atributo proprietário dos imóveis. Depois é feita a comparação do tipo do imóvel. Satisfeitas as duas condições o iterador do objeto será adicionado em um map, esse container irá nos retornar dois valores, com uma key do tipo string e um valor de tipo vetor de iteradores, associando todos os imóveis do proprietário.

2.8. Melhor contêiner para as seguintes situações:

- Acessar uma posição específica de um contêiner;

  Tanto vetores quanto arrays possuem complexidade de acesso O(1) para acesso, no entanto a segurança trazida pela implementação de vetores faz com que ele seja preferível.

- Adicionar um elemento e manter somente elementos únicos no contêiner;

  Para manter elementos únicos é possível utilizar estruturas do tipo Set ou Map e suas variações. A escolha depende do tipo de dado que vai ser utilizado, se os dados precisam estar ordenados, etc.

- Inserção no final;

  Tanto a List quanto a Deque possuem complexidade de inserção O(1) em qualquer parte, no entanto, como a Deque aloca memória em blocos, ela tende a acessar menos a memória do que a List, além disso, como os dados são alocados em blocos eles possuem uma maior probabilidade de serem enviados para a memória cache.

- Remoção no final;

  A melhor estrutura é a Deque pelo mesmo motivo anterior.

- Retornar um valor baseado em uma chave específica (Não necessariamente inteiros);

  Map, devido a possibilidade de associar quaisquer tipos de dados em um par de key e valor.

- Inserção no início;

  A melhor estrutura é a Deque pelo mesmo motivo que a inserção no final.

- Remoção no início;

  A melhor estrutura é a Deque pelo mesmo motivo que a remoção no final.

- Busca por um elemento;

  Ambos map e set possuem uma complexidade O(logN), sendo preferíveis a estruturas como vector. No entanto, um vetor ordenado utilizando uma função como *lower_bound* pode ter performance similar. No entanto, tanto map quanto set na maioria dos casos é menos eficiente que *unordered_set*, já que ele possui em média complexidade O(1) para busca, contudo caso o pior caso de busca seja frequente ele perde para set e map.

- Contêiner com o comportamento de primeiro a entrar é o último a sair;

  Um contêiner Stack segue a semântica de UEPS (último a entrar, primeiro a sair, ou seja, primeiro a entrar, último a sair). O primeiro elemento enviado por push para a pilha é o último elemento a ser removido da pilha como o menos recente.

- Contêiner com o comportamento de primeiro a entrar é o primeiro a sair.

  Um contêiner Queue segue a semântica de PEPS (primeiro a entrar, primeiro a sair). O primeiro elemento enviado por push, ou seja, inserido na fila, é o primeiro a ser removido como o mais recente da pilha, ou seja, removido da fila.

3. Conclusão

No presente trabalho foi possível observar como as implementações de algoritmos e estruturas de dados da STL são ferramentas poderosas para o desenvolvimento de sistemas em C++. Também foi possível utilizar os conceitos de Orientação a Objetos para reduzir a quantidade de código e aumentar a qualidade do código, utilizando de técnicas como polimorfismo, sobrecarga e downcasting.

Concluímos este relatório certos de ter acrescido informações e práticas que nos acompanharão pelo resto de nossa caminhada profissional, buscando sempre nos aprofundar e aprender coisas novas a respeito, com intuito de nos capacitarmos cada vez mais.

4. Referências

STACKOVERFLOW. **In which scenario do I use a particular STL container?.** Disponível em:https://stackoverflow.com/questions/471432/in-which-scenario-do-i-use-a-particular-stl-container. Acesso em: 26 out. 2021.

EMBEDDEDARTISTRY. **Choosing the Right Container: Sequential Containers.** Disponível em:https://embeddedartistry.com/blog/2017/09/11/choosing-the-right-container-sequential-containers/. Acesso em: 26 out. 2021.

STACKOVERFLOW. **How is std::set slower than std::map?.** Disponível em:https://stackoverflow.com/questions/15971429/how-is-stdset-slower-than-stdmap/15972513. Acesso em: 28 out. 2021.

STACKOVERFLOW. **c++ container very efficient at adding elements to the end.** Disponível em:https://stackoverflow.com/questions/30735896/c-container-very-efficient-at-adding-elements-to-the-end. Acesso em: 28 out. 2021.

STACKOVERFLOW. **Which STL container should I use for a FIFO?** Disponível em: https://stackoverflow.com/questions/1262808/which-stl-container-should-i-use-for-a-fifo/1263122. Acesso em: 1 nov. 2021.

GITHUB.IO. **C++ Standard Template Library Quick Reference**. Disponível em: https://alyssaq.github.io/stl-complexities/. Acesso em: 1 nov. 2021.

STACKOVERFLOW. **Which is the fastest STL container for find?**. Disponível em: https://stackoverflow.com/questions/6985572/which-is-the-fastest-stl-container-for-find. Acesso em: 2 nov. 2021.

STACKOVERFLOW. **Why would anyone use set instead of unordered_set?**. Disponível em:https://stackoverflow.com/questions/1349734/why-would-anyone-use-set-instead-of-unordered-set. Acesso em: 8 nov. 2021.

MICROSOFT. **Contêineres da biblioteca padrão C++.** Disponível em: https://docs.microsoft.com/pt-br/cpp/standard-library/stl-containers?view=msvc-160. Acesso em: 8 nov. 2021.