Universidade Federal de Ouro Preto

Departamento de Computação

Augusto Guilarducci (20.1.4012)

Caio Monteiro (20.1.4110)

Matheus Silva Araújo (20.1.1018)

Paulo Correa (20.1.4036)

Pedro Lucas Damasceno (20.1.4003)

Trabalho Prático I – Programação Orientada a Objetos

Implementação de uma corretora de imóveis, sem interface gráfica, baseada na Standard Templates Library (STL)

Ouro Preto, Minas Gerais

2021

SUMÁRIO

- # Introdução

Este relatório se refere ao primeiro trabalho prático da disciplina Programação Orientada a Objetos, realizado durante o período 21.1, cujo objetivo consiste na implementação de uma corretora de imóveis baseada em Standard Templates Library (STL) sem interface gráfica.

As etapas do desenvolvimento serão explicadas em detalhes adiante e o trabalho pode ser visualizado no repositório do *Github* a partir do link https://github.com/augustofgui/grupo-21.1. Para compilar, basta utilizar o arquivo Makefile presente no diretório por meio do comando ‘make’ executado no terminal. Também é possível compilar através do G++ pelo comando ‘g++ *cpp *h -o imobiliária -Wall’, executado sobre o diretório ‘src’, mas para isso é necessário copiar os headers para a mesma pasta.

- **Leitura do arquivo e coleção polimórfica de objetos**

Estabelecemos que as informações referentes aos imóveis seriam passadas como argumento na linha de comando durante a execução do programa. A partir disso, efetuamos a leitura de cada linha e a divisão de informações com base na implementação em C++ da função *explode*, nativa da linguagem PHP. Após lida e separadas as informações de uma linha, criamos um objeto de acordo com a primeira posição do vetor de *strings*, que nesse caso determina o tipo do imóvel (casa, apartamento ou chácara), inserimos os dados a partir do construtor do objeto e, por fim, inserimos o objeto em um vetor polimórfico de tipo *Imovel**.

- **Fluxo de execução e menu de opções**

Uma vez que um arquivo válido seja passado na execução do programa todos os imóveis presentes atendam aos tipos descritos no enunciado do trabalho prático, um menu de opções é exibido e o usuário pode conferir a execução dos 8 critérios exigidos. O fluxo de execução das opções é determinado por um comando *switch*, que retorna 0 na função principal quando a opção '0 – Sair' é inserida.

```
- Menu de Opções -

0 - Sair
1 - Buscar por proprietário
2 - Buscar por valor máximo
3 - Buscar por número de quartos
4 - Buscar por tipo do imóvel
5 - Buscar por cidade
6 - Criar iterador para um
proprietário
7 - Exibir ou salvar coleção de
imóveis
8 - Imprimir menu de opções

Digite qual opção deseja:
```

- **Funções do sistema**

1. Sobrecarga do operador << (saída)

Conforme visto nas aulas, criamos uma função *virtual* que faz a construção do *stream* de dados e a referenciamos na função friend responsável pela sobrecarga de <<.

```
void Imovel::saida (ostream &out){
    out << setprecision(2);
    out << "Proprietario: " << proprietario << endl
        << "    Valor: " << fixed << valor << endl
        << "    Quartos: " << quartos << endl
        << "    Rua: " << rua << endl
        << "    Bairro: " << bairro << endl
        << "    Cidade: " << cidade << endl;
}

ostream &operator<<(ostream &out, Imovel &imovel){
    imovel.saida(out);
    return out;
}
```

Sobrecarga na função da classe base (Imovel)

```
void Casa::saida (ostream &out){
    Imovel::saida(out);
    out << "    Andares: " << andares << endl;
}

ostream &operator <<(ostream & out, Casa &casa){
    casa.saida(out);
    return out;
}
```

Sobrecarga na função da classe derivada (Casa)

2. Função que retorna *true* se o proprietário possuir algum imóvel ou false se não

Percorremos toda a extensão da coleção de imóveis e utilizamos o *getter* '*getProprietario*' para acessar a informação privada do objeto e compará-la com o proprietário informado.

```
for (int i = 0; i < (int)imoveis_database.size(); i++)
    {
        if (imoveis_database[i]->getProprietario() == nome)
            return true;
    }
    return false;
```

Iteração sobre a coleção de imóveis

3. Função que retorna uma única coleção com imóveis a partir de um valor máximo

Haja vista a necessidade de retornar uma coleção, elaboramos uma função de tipo *std::vector<Imovel*>* e percorremos a coleção passada por parâmetro da mesma forma vista na função 2. Utilizamos o *getter* de '*getValor*' para comparação e inserimos com *push_back* na coleção de retorno se a condição for satisfeita.

```
vector<Imovel *> imoveis_preco_maximo;
    for (int i = 0; i < (int)imoveis_database.size(); i++)
    {
        if (imoveis_database[i]->getValor() <= preco)
            imoveis_preco_maximo.push_back(imoveis_database[i]);
    }

    if(!(int)imoveis_preco_maximo.size())
        cout << "Não foram encontrados imóveis nessa faixa de
preço!\n" << endl;

    return imoveis_preco_maximo;
```

Iteração sobre a coleção de imóveis e inserção na outra se o critério for atendido

4. Função que retorna uma única coleção com imóveis a partir de um número de quartos

Implementação similar à função 3, diferindo apenas quanto ao *getter* utilizado para realizar a comparação.

5. Função que retorna uma coleção de um tipo ordenada por valor

Utilizamos o operador *typeid* para determinar a natureza de um objeto e compará-lo com o requisitado pelo usuário durante a iteração da coleção de imóveis. Uma vez criada a coleção contendo apenas o tipo de imóvel requisitado, o vetor é ordenado com o algoritmo *sort* presente na STL. Para a utilização do algoritmo, implementamos uma função para ser utilizada como comparador de objetos.

```
bool ordem_crescente(const Imovel *imovel1, const Imovel *imovel2){
    return imovel1->getValor() < imovel2->getValor();
}
```

Função comparadora para ordenação do vetor de objetos

6. Função que retorna uma coleção de uma mesma cidade ordenada decrescentemente por valor

Implementação similar à função 5, diferindo apenas quanto ao sinal da função de comparação.

7. Função que retorna um iterador para imóveis de um proprietário

Tivemos dificuldade para compreender e implementar esse critério em específico. Trabalhando com o tipo *iterator*, conseguimos retornar apenas um imóvel da função, o que não satisfaz casos onde o proprietário possui mais de um imóvel. Dialogando com outros alunos, foi comentada a possibilidade da utilização de um vetor de inteiros contendo os índices da coleção de imóveis nos quais hajam imóveis do proprietário referenciado. A partir dessa ideia, conseguimos implementar uma solução funcional para o problema e imprimir no *main* apenas as posições que contenham os imóveis buscados.

```
vector <int> iteradores;
    for (int i = 0; i < (int) imoveis_database.size(); i++){
        if (imoveis_database[i]->getProprietario() ==
nome_proprietario)
            iteradores.push_back(i);
    }

    if(!(int)iteradores.size())
        cout << "Não foram encontrados imóveis desse proprietário
(iterador vazio)!\n" << endl;

    return iteradores;
```

Criação do vetor de iteradores (inteiros)

```
for (int i = 0; i < (int) iteradores.size(); i++){
                    cout << *imoveis_database[iteradores[i]] <<
endl;
                    cout << "- - - - -\n" << endl;
                }
```

Impressão dos índices buscados a partir do vetor de iteradores

8. Função que imprime uma coleção ou salva em memória secundária

Primeiro, perguntamos ao usuário qual função ele gostaria de utilizar (impressão na tela ou registro em arquivo em memória secundária). Caso queira imprimir, utilizamos uma pequena função para imprimir os objetos utilizando o operador << sobrecarregado previamente. Já para registrar em arquivo, utilizamos novamente o operador *typeid* para determinar a natureza do objeto e convertê-lo utilizando *downcasting*. Por fim, escrevemos a impressão de cada tipo de objeto separadamente e as utilizamos conforme comparação por *typeid*.

```
for (int i = 0; i < (int)imoveis_database.size(); i++){
        if (typeid(*imoveis_database[i]).name() == typeid(class
Casa).name()){
            Casa* casa = dynamic_cast<Casa *>
(imoveis_database[i]);
            arquivo_saida << "casa;" <<
            casa->getValor() << ";" <<
            casa->getProprietario() << ";" <<
            casa->getRua() << ";" <<
            casa->getBairro() << ";" <<
            casa->getCidade() << ";" <<
            casa->getNumero() << ";" <<
            casa->getQuartos() << ";" <<
            casa->getBanheiros() << ";" <<
            casa->getAndares() << ";" <<
            casa->getSalaJantar() << ";" << endl;
        }
```

Comparação por typeid para determinar modelo de impressão e downcasting para utilizar os getters característicos das classes derivadas

• Contêiners da STL

Dado os contêineres presentes na STL, deve ser descrito sucintamente qual o melhor contêiner para cada uma das seguintes situações:

- Acessar uma posição específica de um contêiner;
  - Contêiner vector, pois um vetor armazena elementos de um determinado tipo em uma disposição linear e permite acesso aleatório rápido a qualquer elemento. Ele tem comprimento altamente flexível, além de ser armazenado contiguamente.

- Adicionar um elemento e manter somente elementos únicos no contêiner;
  - Contêiner set, pois é apenas um contêiner em ordem crescente de elementos exclusivos onde o valor também é a chave. Assim, esse container seria o preferencial para esse caso.

- Inserção e Remoção (início e final);
  - Contêiner deque (fila de duas extremidades), pois permite rápidas inserções e remoções no início e no final do contêiner. Ele possui as vantagens de acesso aleatório e comprimento flexível, mas não é contíguo.

- Retornar um valor baseado em uma chave específica (não necessariamente inteiros);
  - Contêiner map, pois consiste em um par chave/valor. A chave é usada para ordenar a sequência e o valor é associado essa chave. Por exemplo, um map pode conter chaves que representam cada palavra exclusiva em um texto e valores correspondentes que representam o número de vezes que cada palavra aparece no texto. Logo, por se tratar exclusivamente desse caso, esse container seria o preferencial.

- Busca por um elemento;
  - Contêineres associativos ordenados (map, multimap, set e multiset), pois são suportados por pesquisa heterogênea, o que significa que você não precisa mais passar exatamente o mesmo tipo de objeto que a chave ou o elemento em funções membro. Em vez disso, você pode passar qualquer tipo para o qual é definido um operador sobrecarregado que permite a comparação com o tipo de chave.

- Contêiner com o comportamento de primeiro a entrar é o último a sair;
  - Contêiner stack, pois segue a semântica de UEPS (último a entrar, primeiro a sair). O último elemento enviado por push para a pilha é o primeiro elemento a ser removido da pilha como o mais recente.

- Contêiner com o comportamento de primeiro a entrar é o primeiro a sair.
  - Contêiner queue, pois segue a semântica de PEPS (primeiro a entrar, primeiro a sair). O primeiro elemento enviado por push para a pilha é o primeiro a ser removido como o mais recente da pilha .