***Importante***: Para cada item abaixo deve ser copiado trechos do código que cumprem o requisito e explicado, se não for aparente, o porquê o requisito é cumprido. Sejam bem explícitos. Deve ser indicado também o arquivo da classe em que está o trecho do código. Eu avaliarei o código do Github a partir desse documento para confirmá-lo e também para detectar possíveis erros. **Quem não seguir o que está indicado aqui, não terá o projeto avaliado e perderá a atividade.**

**Criar repositórios desde o início do desenvolvimento. Primeiro commit deve estar com todas as classes declaradas sem implementação.**

**Primeira versão do código completa dia 01 de dezembro.**

**Versão final do código dia 08 de dezembro.**

**No dia da apresentação deve ser mostrado o código rodando para vários casos de teste.**

**Será avaliado o quanto o projeto está elaborado. Trabalhos muito simples com métodos apenas mostrando mensagens, serão penalizados.**

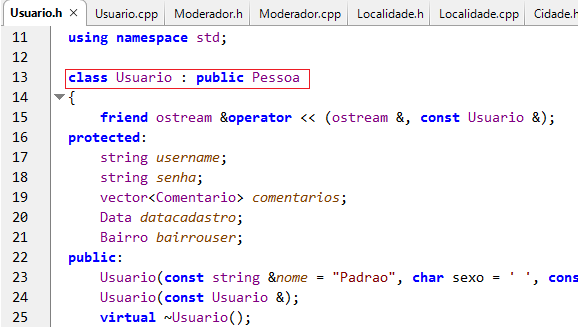
**Requisitos de implementação para C++**

1. Todas as classes concretas devem vir de classes abstratas. Pelo menos três hierarquias de classes. Uma das hierarquias deve ter três níveis. Exemplo: Personagem (abstract) >> Ciborgue (Abstract) >> Robocop; Class Arma (Abstract) >> Beretta93R

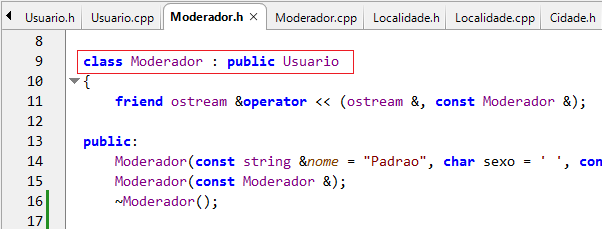
- Como pode ser visto no diagrama UML e no trecho de código a seguir, todas as classes concretas(com exceção da classe “Data” e “Comentario”), sofrem herança de classes abstratas.

- Classe *Pessoa:* dessa classe herdam as classes *Usuário* e *Moderador:*

*- Usuário.h:*

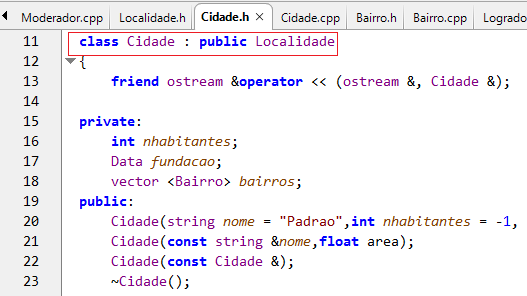


*- Moderador.h*(herda de usuário):

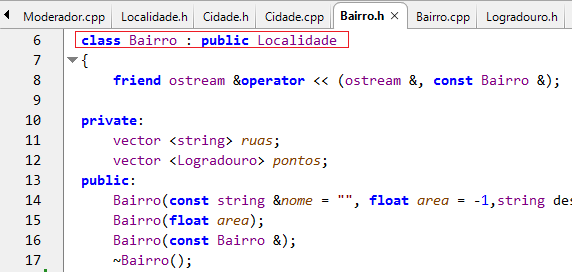


- Classe *Localidade*: dessa classe herdam as classes *Cidade*, *Bairro* e *Logradouro*(da qual a classe *PontoTuristico* herda):

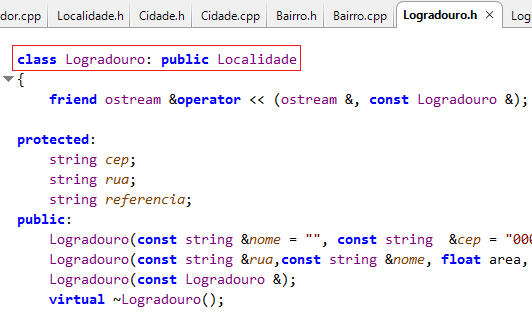
- *Cidade.h*:



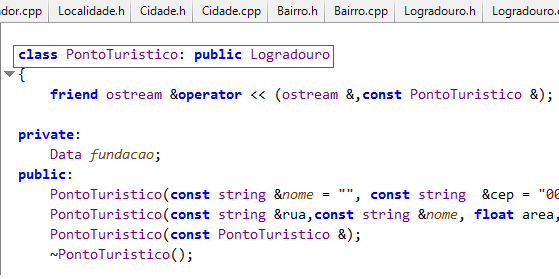
- *Bairro.h:*



*- Logradouro.h*:



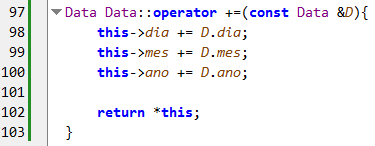
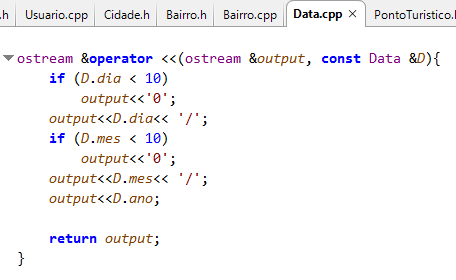
- *PontoTuristico.h*(herda de *Logradouro*):



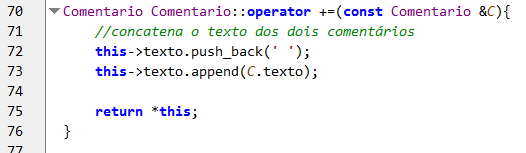
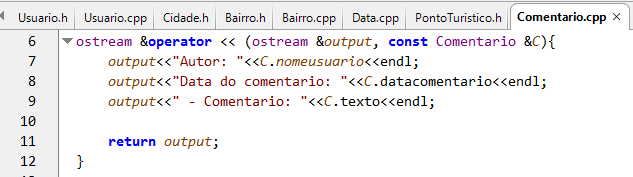
1. Em todas as classes: **construtor de cópia**, **operatores<< e +=,** e **construtor default**. Fazer o máximo de reaproveitamento de código usando **static\_cast**

- Como pode ser visto no item 6, todas as classes possuem construtor de cópia e default. Segue abaixo os códigos das sobrecargas dos operadores “<<” e “+=”(nas classes abstratas não é possivel a implementação do operador “+=”):

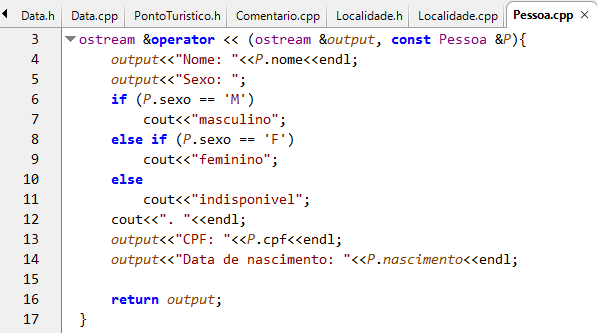
- *Data:*



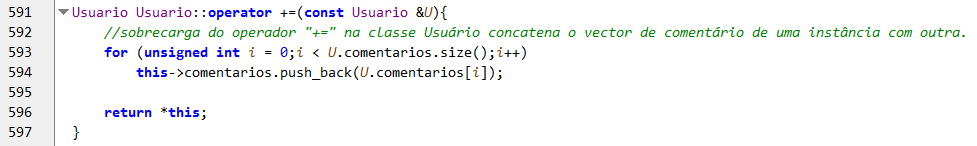
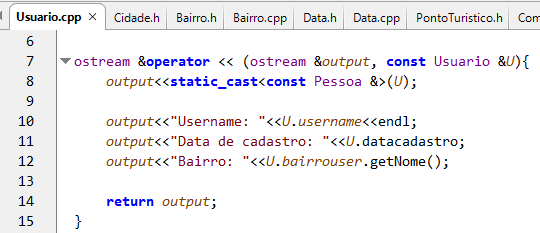
*- Comentario:*



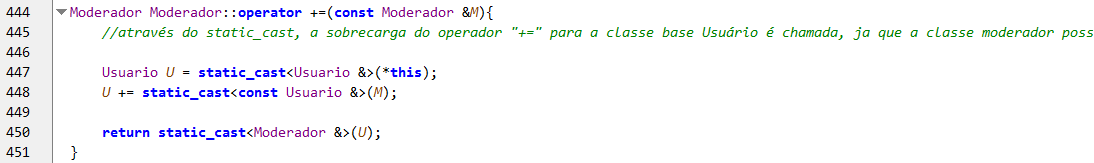
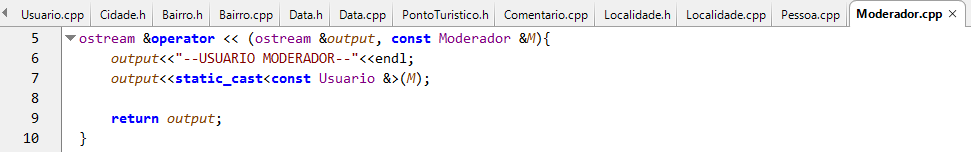
*- Pessoa:*



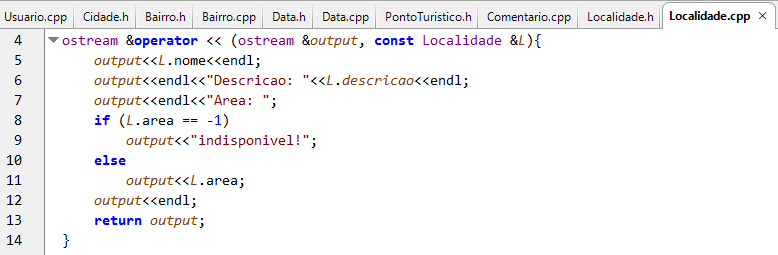
*- Usuario:*



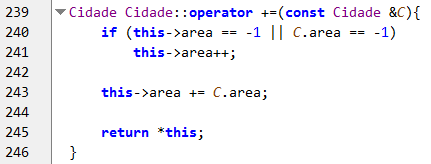
*- Moderador:*



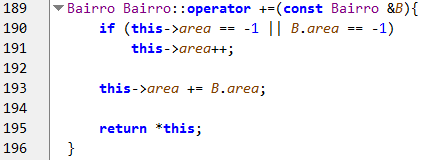
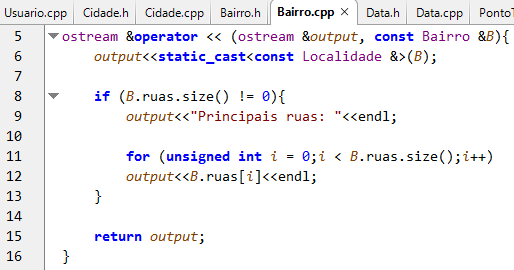
*- Localidade:*



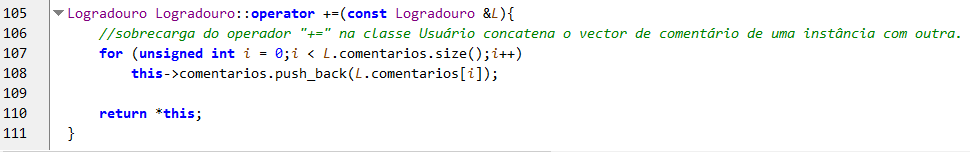
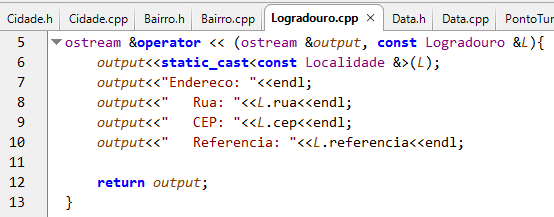
*- Cidade:*



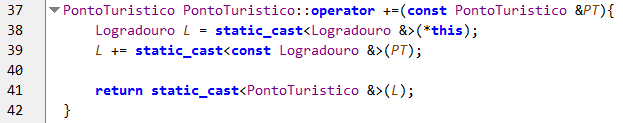
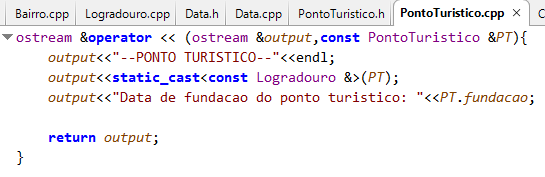
*- Bairro:*



*- Logradouro:*



*- PontoTuristico:*



1. Todas as hierarquias devem ter classes Concretas, e em uma das hierarquias, três classes Concretas relacionadas: Exemplo Servico >> ServicoStream >> (Netflix, HBOStream, AmazonPrime, NowTv). Em uma pesquisa de 10 segundos: <http://www.tomsguide.com/us/pictures-story/620-top-online-streaming-video.html>

- Como pode ser visto no item 1, a hierarquia que começa na classe abstrata *Pessoa* possui 2 classes concretas(*Usuario* e *Moderador*).

A outra hierarquia que começa na classe abstrata *Localidade* possui 4 classes concretas relacionadas: *Cidade*, *Bairro*, *Logradouro* e *PontoTuristico.*

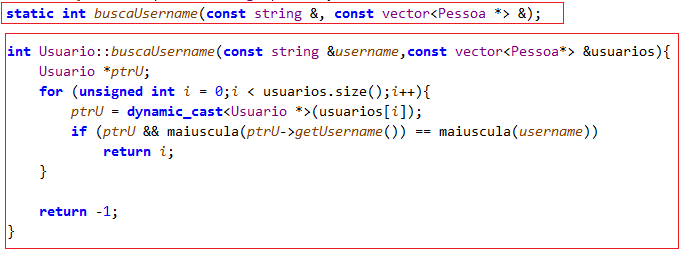
1. Atributos static e const static em todas as hierarquias de classe

- Apenas um atributo const static(*QTDMAXCHAR = 140*, localizado na classe comentário).

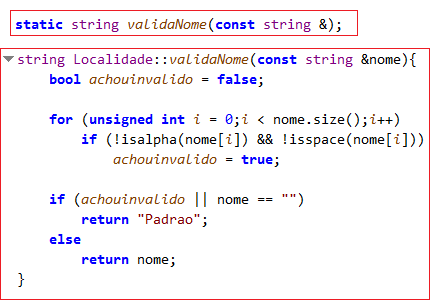
1. Método static em todas as hierarquias de classe

- Nas duas principais hierarquias de classe do projeto, existem alguns métodos static. Seguem um exemplo de cada hierarquia abaixo:

- Hierarquia da classe *Pessoa* : método *buscaUsername()* na classe *Usuario*:

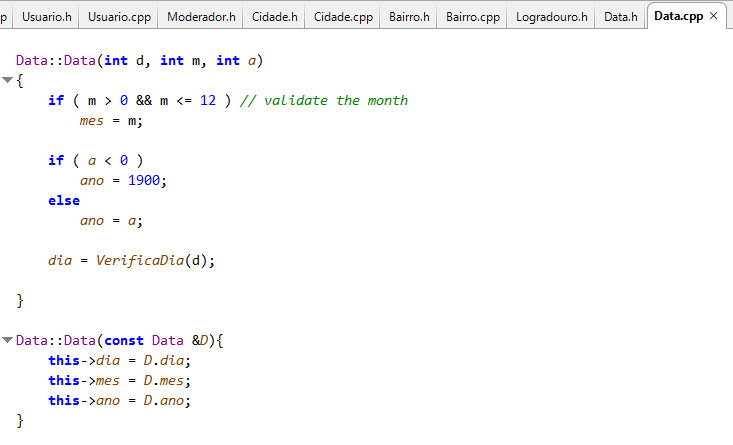


- Hierarquia da classe *Localidade* : método *validaNome()* na classe *Localidade*:

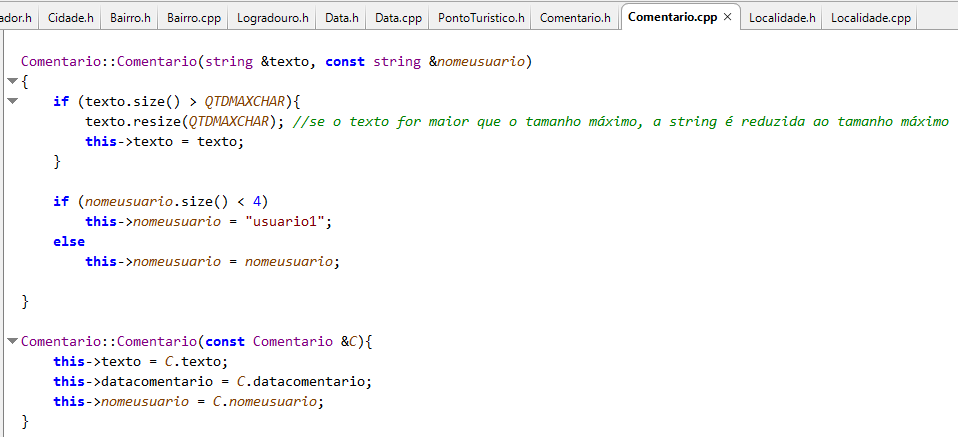
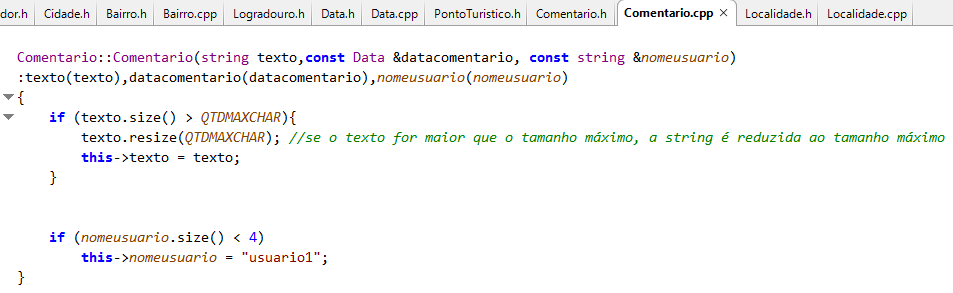


1. Construtores em todas as classes, e três para todas as classes da hierarquia principal. Sempre validar os dados em todas as classes

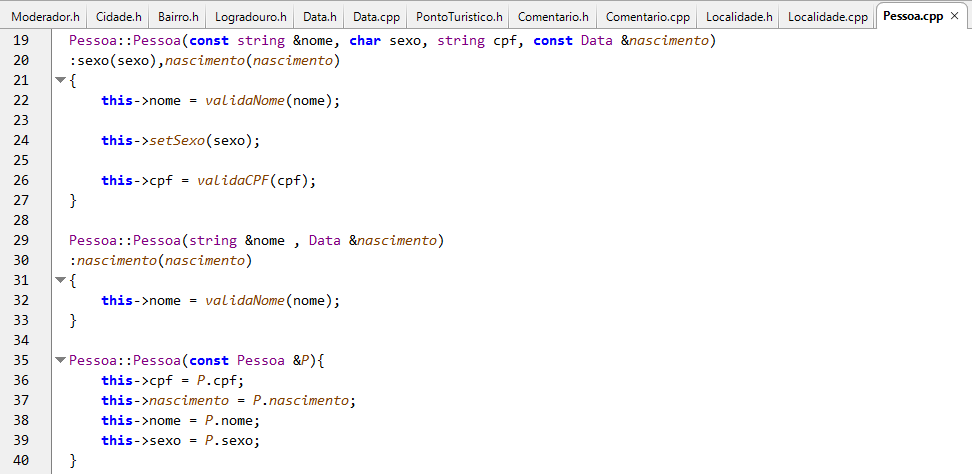
- *Data:*



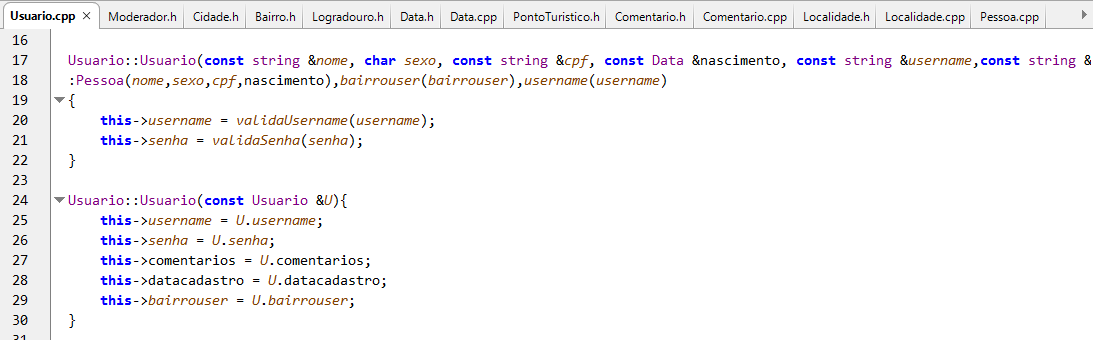
*- Comentario:*



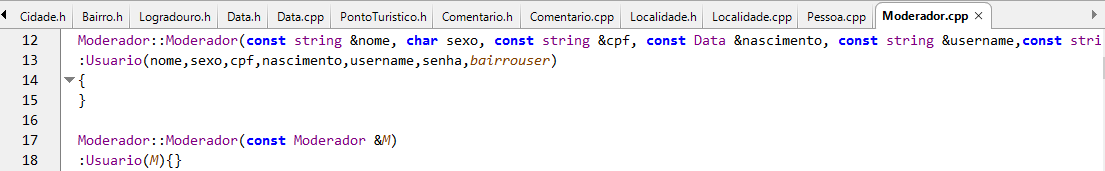
*- Pessoa:*



*- Usuario:*

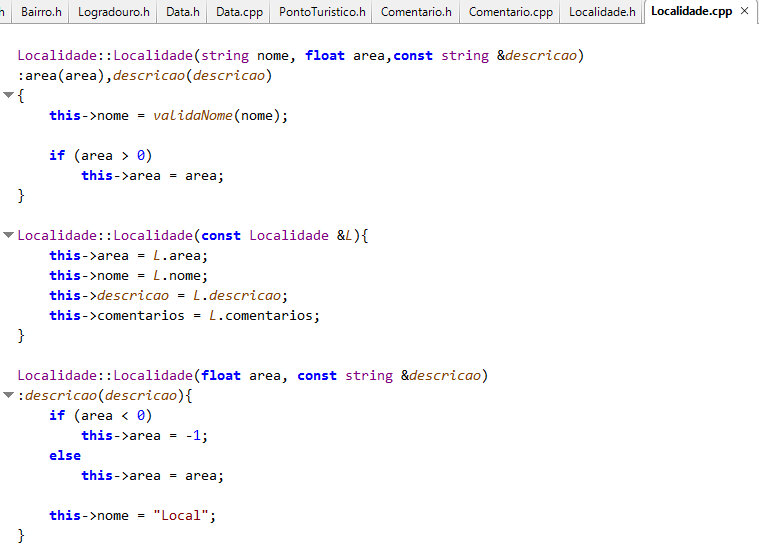


*- Moderador:*

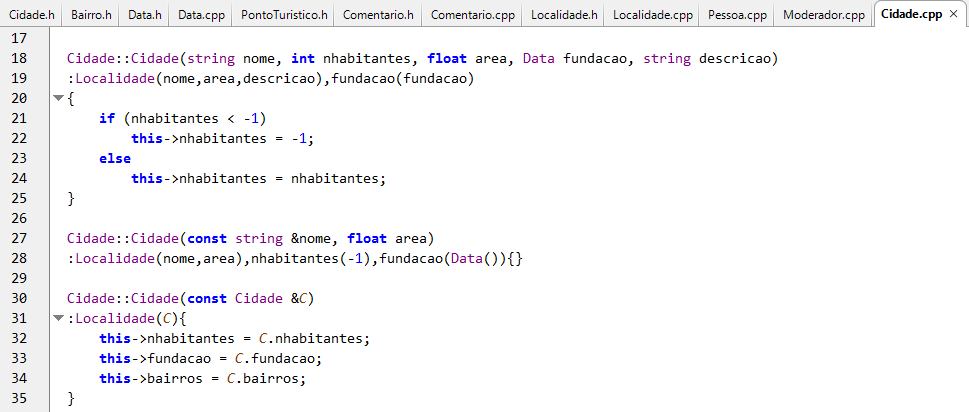


*-* **Hierarquia principal:**

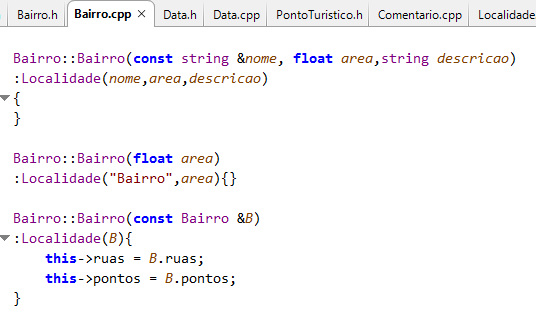
*- Localidade:*



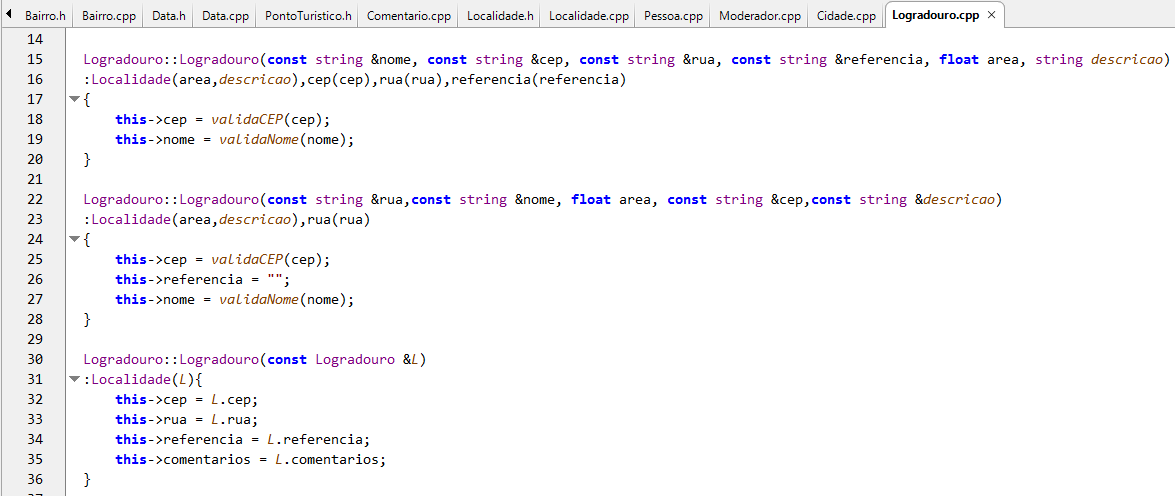
*- Cidade:*



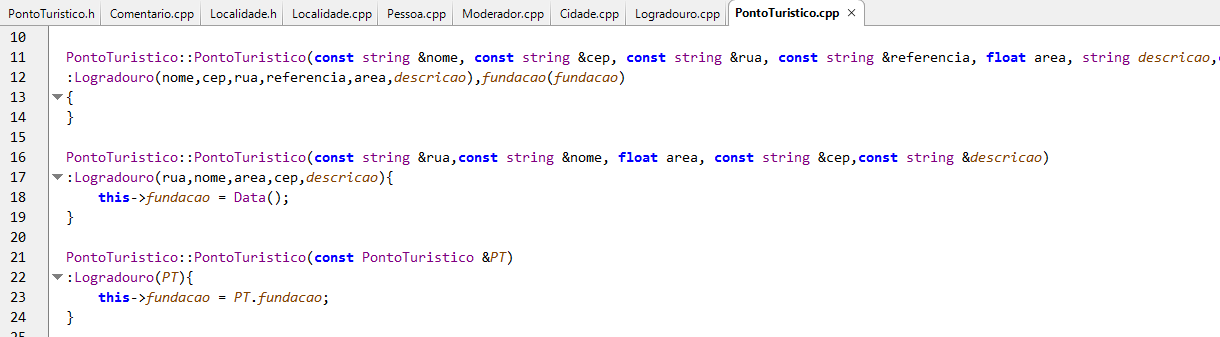
*- Bairro:*



*- Logradouro:*



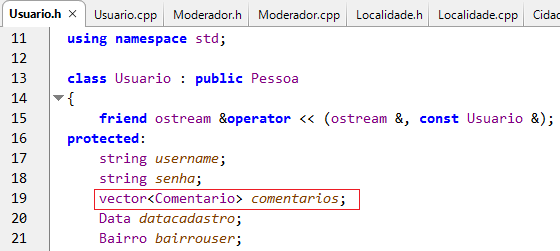
*- PontoTuristico:*



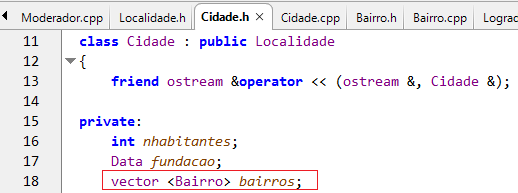
1. Vector em todas em todas as hierarquias de classe

- Ha pelo menos um vector em praticamente todas as classes do projeto. Seguem dois exemplos abaixo, um de cada hierarquia principal do projeto.

- Hierarquia da classe *Pessoa* : Uma instância da classe *Usuario* possui um vector de objetos da classe *Comentario:*



- Hierarquia da classe *Localidade* : Uma instância da classe *Pessoa* possui um vector de objetos da classe *Bairro* :



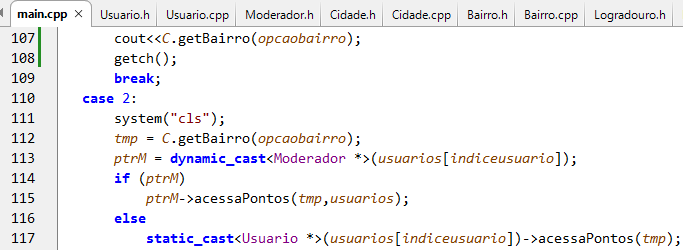
1. ENUM na hierarquia principal

- Requisito não cumprido.

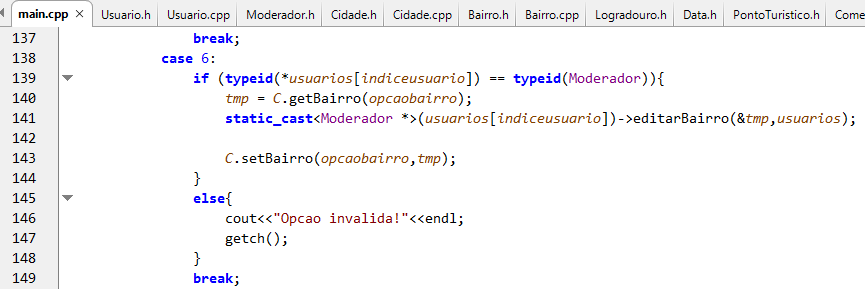
1. Usar o **dynamic cast** e **typeid** no main junto com as classes concretas. Para uma da classe concreta identificada, chamar um método dessa classe e fazer uma ação:

- Seguem abaixo trechos de código de exemplos de uso do **dynamic\_cast** e **typeid** no main:

- **dynamic\_cast**: neste exemplo, o dynamic\_cast é usado para verificar se um dos objetos do vector *usuarios*, cujos elementos são ponteiros para objetos da classe *Pessoa* , é do tipo *Moderador*. Se o cast obter sucesso, é chamada a função *acessaPontos()* pertencente à classe *Moderador.* Caso contrário, é feito um **static\_cast**para um ponteiro da classe *Usuario* e é chamada a função *acessaPontos()* dessa classe:

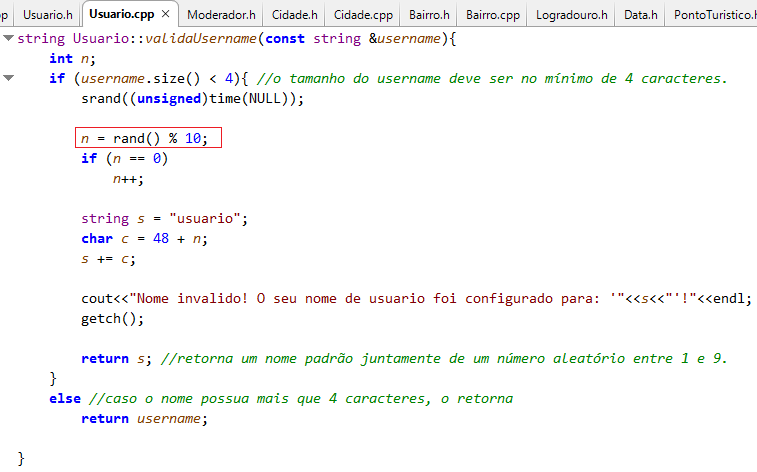


- **typeid:** No exemplo abaixo, o typeid é utilizado para verificar se o objeto dado é um ponteiro para um objeto da classe *Moderador*. Caso seja, ele chama a função *editarBairro()*, e com seus privilégios de moderador pode modificar os atributos de um bairro. Caso contrário, não acontece nada e o programa não permite que o usuário comum acesse essa opção:



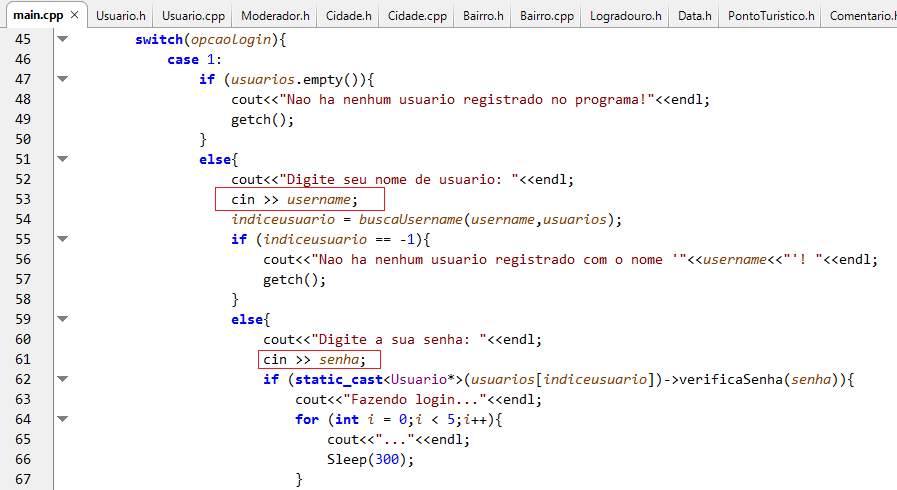
1. Usar o rand. Nota: deve ser usado conforme o contexto do projeto. Se for usado em um método genérico sem relação com a classe e apenas para cumpri-lo, esse requisito será desconsiderado. <http://en.cppreference.com/w/cpp/numeric/random/rand>

- O rand é utilizado na função *validaUsername(),* que valida um nome de usuário dado. Caso o nome seja inválido, o programa retorna uma string “usuario”, seguida no final por um número aleatório de 1 a 9, gerado pelo rand:



1. No main o usuário deve fazer entrada via teclado e interagir com a aplicação

- Isso acontece diversas vezes. Segue um exemplo de codigo abaixo, que corresponde ao evento no programa onde o usuário faz o login, entrando com o seu nome de usuário e senha, e o programa então verifica se o usuário está registrado:



**Requisitos de implementação para Java**

1. Todas as classes concretas devem vir de interfaces ou classes abstratas. Pelo menos três hierarquias de classes. Uma das hierarquias deve ter três níveis. Exemplo: Personagem >> Ciborgue >> Robocop; Class Arma (interface) >> Beretta93R
2. Ao menos três interfaces. A terceira interface deve ser uma interface que liga duas hierarquias como no exemplo da interface **corredor** (Figura 1).
3. Usar a interface **Comparable** e sobrescrever o método **compareTo** em pelo menos uma hierarquia
4. Sobrescrever **equals** para de Object
5. Todas as hierarquias devem ter classes Concretas, e em uma das hierarquias, três classes Concretas relacionadas: Exemplo Servico >> ServicoStream >> (Netflix, HBOStream, AmazonPrime, NowTv). Em uma pesquisa de 10 segundos: http://www.tomsguide.com/us/pictures-story/620-top-online-streaming-video.html
6. Sempre usar o super para o máximo de reaproveitamento de código em todas as classes
7. Atributos static e const static em todas as hierarquias de classe
8. Método static em todas as hierarquias de classe
9. Três construtores para todas as classes, sendo um deles o construtor de cópia. Sempre validar os dados em todas as classes
10. ArrayList na hierarquia principal
11. ENUM na hierarquia principal
12. Usar o **instanceof** no main junto com as classes concretas. Para uma da classe concreta identificada, chamar um método dessa classe e fazer uma ação;
13. Dividir o projeto em pacotes
14. Sobrescrever para todas as classes o método toString
15. Usar a classe Random do pacote java.util (java.util.Random). Nota: deve ser usado conforme o contexto do projeto. Se for usado em um método genérico sem relação com a classe e apenas para cumpri-lo, esse requisito será desconsiderado.
16. No main o usuário deve fazer entrada via teclado e interagir com a aplicação. Opcional de bônus: pode ser usada a classe JOptionPane do pacote javax.swing. Vejam: showInputDialog e showMessageDialog.



Figura 1 – Interface Corredor conectando duas hierarquias de classe