UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CARLOS DANIEL ALBERTINO VIEIRA

DAVID CRISTIAN MOTTA PROPATO

SISTEMA ELEITORAL BRASILEIRO

Vitória

2022

CARLOS DANIEL ALBERTINO VIEIRA
DAVID CRISTIAN MOTTA PROPATO

SISTEMA ELEITORAL BRASILEIRO

Relatório apresentado à disciplina de Programação Orientada a Objetos como requisito parcial para obtenção de nota.
Professor João Paulo Andrade Almeida.
Matrículas: 2020100867 e 2020101300

Vitória
2022

Sumário

1 INTRODUÇÃO

O sistema eleitoral brasileiro apresenta dois modelos de eleições na escolha dos representantes políticos: majoritário e proporcional. Cargos como presidente, governador, senador e prefeito são escolhidos sob o regime da eleição majoritária, na qual os candidatos mais votados são eleitos, enquanto os demais cargos são decididos pelo modelo da eleição proporcional, onde a eleição dos candidatos depende também do número de votos de seus partidos.

Todavia, o funcionamento do sistema eleitoral brasileiro continua uma incógnita para muitos cidadãos, que em geral desconhecem sua operação. Essa desinformação serve de obstáculo no entendimento tanto do meio como se tece o cenário político atual, como da cidadania de modo geral.

Em vista disso, desmistificar o funcionamento das eleições se torna uma tarefa essencial na efetivação da cidadania. Nessa ótica, ferramentas que disponibilizam informações capazes de diminuir o nível de desentendimento sobre as eleições ganham relevância, o que abre espaço para contribuições provenientes da programação orientada a objetos no fomento de sistemas de análise e exposição de dados sobre tal temática.

1.1 OBJETIVOS

Implementar um sistema em C++ capaz de processar dados obtidos da Justiça Eleitoral referentes à votação de vereadores, de forma a levantar informações e gerar relatórios sobre as eleições de 2020 em um município, a fim de aplicar e desenvolver os conhecimentos adquiridos em aula.

2 IMPLEMENTAÇÃO

A fim de padronizar a escrita dos dados em diferentes máquinas segundo a convenção brasileira exigida, utilizou-se o locale pt Br, o qual foi levemente modificado. A alteração se deu por meio da criação de uma subclasse da faceta numpuct<char>, a fim de substituir o formato de agrupamento padrão de milhares do locale pt Br, que é um ponto (ex: 12.345,67), para nenhum caráter (12345,67). Vale ressaltar, ainda, que por se tratar de uma medida específica e não tão demandante, decidiu-se não definir a subclasse previamente comentada em um arquivo separado e externo ao arquivo cliente. Além disso, foi inicializado como default no início do programa a precisão de duas casas decimais para todos os números do tipo ponto flutuante, ou seja, as porcentagens.

Foram organizados os arquivos .h e .cpp separadamente, e seguindo o padrão da implementação em Java, foi mantido a classe de Candidato e RelatoriosCandidatos juntos, da mesma forma para as classes Partido e RelatoriosPartidos.

Figura 1 - Árvore dos arquivos e diretórios feitos no trabalho.

2.1 CLASSES PRINCIPAIS

O programa contou com a criação de classes específicas para a realização das tarefas propostas, nas quais foram empregadas tipos nativos e de referência que apresentam funções úteis no trato da informação armazenada. Assim, ao todo foram criadas 7 classes: Leitura, Data, Candidato, Partido, RelatoriosCandidato, RelatoriosPartido e Excecoes.

2.1.1 Leitura

Essa classe é responsável pela leitura dos arquivos de candidatos e vereadores.

As funções LerCandidatos() e LerPartidos() leem todas as linhas do arquivo de entrada. Para cada uma dessas, criam ponteiros dinamicamente alocados para Candidato e Partido através das funções LerCandidato() e LerPartido(), armazenando esses em uma lista (de ponteiros para Candidatos e de Partidos).

Caso ocorra algum erro que gere uma exceção na leitura dos partidos, o programa será encerrado (caso não seja, um candidato que seria daquele partido ficaria sem nenhum, resultando em erro). Na mesma lógica, se houver erro na leitura de algum candidato, a linha com suas informações será ignorada e o programa continuará, pois pode haver partidos sem nenhum candidato mas não candidatos sem nenhum partido.

2.1.2 Data

Como não se encontrou na linguagem um tipo nativo ou importado de dado que oferecesse um método satisfatório no tratamento das datas contidas no arquivo de entrada, foi implementada uma classe personalizada para suprir esse papel. Esse tipo customizado permitiu que fossem atendidas as demandas específicas do projeto, preenchendo lacunas à medida que estas se mostravam desafiadoras.

Acerca dessa classe, é possível destacar alguns métodos que significaram bons avanços na manipulação de informação relacionada a datas. Dentre essas estão a definição de operadores (atribuição (=) e subtração (-)) que reduziram a

complexidade na atribuição e comparação de datas, bem como ainda funções que permitiram contabilizar a idade dos candidatos.

Por fim, de modo geral, essa classe proveu uma forma interessante de trabalhar com todo tipo de informação e ação relacionadas ao tratamento de datas.

2.1.3 Candidato

Foi criada com o intuito de reunir as informações relacionadas a um candidato (disponíveis no arquivo de leitura) e também funções específicas relacionadas à impressão na saída padrão e comparação de objetos desse tipo.

O ponto de destaque para os campos dessa classe está na escolha dos tipos de alguns desses. Durante o desenvolvimento do trabalho, notou-se que não eram utilizados alguns campos da classe, sendo que outros poderiam ser substituídos por valores mais simples, como booleanos. Assim, como uma das requisições para o trabalho é considerar apenas os candidatos válidos, candidatos com destinos de votos diferentes não foram armazenados. Além disso, a variável de situação também foi modificada para armazenar valor booleano *True* para Eleitos e *False* para os demais.

Ademais, também foi adicionado uma variável da classe Partido, a fim de relacionar cada candidato diretamente com seu partido e facilitar a construção de funções necessárias para a geração de alguns relatórios.

Sobre as funções implementadas, foram criados, ainda, getters e setters de acordo com a necessidade do projeto. Desse modo, esses não foram definidos para todas variáveis, a exemplo do único setter construído para o partido do candidato, visto que esse campo, diferentemente dos demais, apenas poderia ser preenchido após lido o arquivo de entrada dos partidos.

2.1.4 Partido

Foi criada com o intuito de reunir as informações relacionadas a um partido (disponíveis no arquivo de leitura) e também funções específicas relacionadas aos objetos desse tipo.

Para as variáveis, o destaque foi na criação de uma variável do tipo list<Candidato*> para armazenar a lista de ponteiros para candidatos de um partido. E também, da mesma forma que para os candidatos havia variáveis não utilizadas, nos dados de entrada dos partidos é passado os nomes de cada, contudo esses não são utilizados na execução do código, com isso, se torna mais otimizado não armazená-los.

Sobre as funções implementadas, foram criados getters e setters de acordo com a necessidade no projeto. Desse modo, esses não foram definidos para todas variáveis, a exemplo do único setter construído para a adição de um elemento na lista de candidatos, visto que essa variável, diferentemente das demais, apenas poderia ser preenchida após a leitura do arquivo de entrada dos candidatos. Além disso, também foram implementados funções que poderiam ser úteis na contabilização de votos e candidatos.

2.1.5 RelatoriosCandidatos

Foi criada com o intuito de analisar e processar as informações relacionadas a uma lista de ponteiros para Candidato, que é armazenada no próprio objeto.

Essa classe se destaca através da função AssociarPartidosCandidatos(), que, ao receber listas de candidatos e partidos, conecta cada partido com seus devidos candidatos por meio do número do partido, variável do tipo int comum às classes (Candidato e Partido). Se destaca também com a função ordenaVotosNominais(), que ordena a lista de candidatos com base nos votos nominais, sendo a idade o critério de desempate.

As demais funções implementadas nessa classe focam nos relatórios relativos aos candidatos.

2.1.6 RelatoriosPartidos

Foi criada com o intuito de analisar e processar as informações relacionadas a uma lista de ponteiros para Partido, que é armazenada no próprio objeto.

Essa classe se destaca através das funções de ordenação e, principalmente, pelo destrutor, sendo a única classe com um implementado. Nela há três funções de

ordenação, ordenarVotosTotais(), ordenarVotosLegenda() e ordenarVotosNominais(), e há o destrutor, que realiza o delete de todos os partidos e candidatos alocados dinamicamente com o prefixo new durante a leitura dos arquivos de entrada.

As demais funções implementadas nessa classe focam nos relatórios relativos aos partidos.

2.1.7 Excecoes

Essa classe tem a função de gerar eventuais exceções lançadas no decorrer do código, principalmente nas funções de leitura e inicialização, juntando lógica C e C++ para maior eficiência e utilidade nos tratamentos.

Nela se encontra a superclasse Excecoes, que será capturada nos catchs, e há também as classes derivadas, que são específicas para cada exceção lançada.

2.2 RELACIONAMENTO ENTRE CLASSES

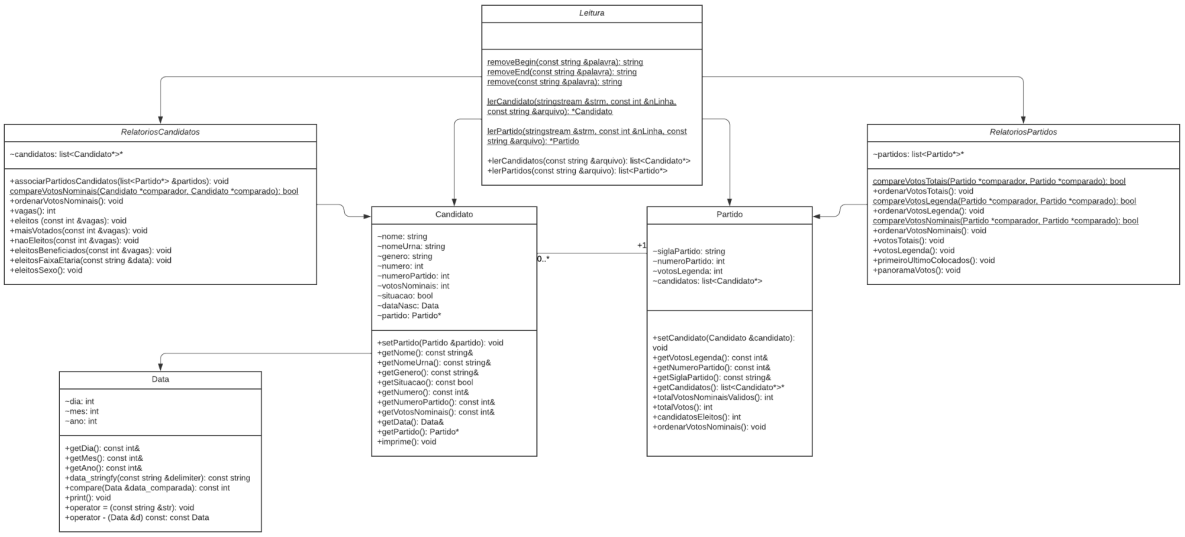

Figura 2 - Diagrama UML de classes do programa. Link para melhor visualização através do Google Drive embutido na imagem.

2.3 TRATAMENTO DE EXCEÇÕES

Para tratar de forma eficiente as exceções e evitando erros, foi montado um try na main (Client.cpp) para sempre encerrar a execução do programa caso uma exceção seja lançada. Além de exceções nativas da linguagem, também foram criadas classes próprias para tratar as possíveis exceções.

Erros de abertura/leitura de arquivo, erros de inserção de listas nulas nos objetos das classes de Relatório e erros de leitura/armazenamento de partidos geram exceções que serão capturadas na main e que encerraram o programa, sendo impresso na saída padrão informações a respeito do erro e da exceção causada. Porém, erros de leitura/armazenamento de candidatos lidos não encerraram o programa, sendo apenas interpretadas como linhas inadequadas e ignoráveis.

2.3.1 invalid_argument

Exceção gerada ao falhar na conversão de uma String para uma formatação válida do tipo int. Tal exceção poderia ser gerada pelo programa ao tentar realizar o parsing das strings de número e votos dos partidos e candidatos. Por se tratar de um argumento indispensável para o funcionamento do programa e que é passível de erros, é necessário cobrir a exceção gerada nesses casos.

2.3.2 bad_alloc

Exceção gerada ao falhar em alocar dinamicamente na memória as variáveis de partidos e candidatos lidos, retornando uma exceção para a ain e encerrando o programa, no caso dos partidos. Para os candidatos, ela é tratada dentro das funções de leitura, que são as únicas que realizam allocs através do prefixo new.

2.3.3 ErroAbertura

Lançada quando ocorre um erro na abertura dos arquivos de entrada. Através de um if que checa se o arquivo de candidatos e partidos não foi aberto, lança a

exceção caso a condição seja verdadeira. E a mesma só será capturada e tratada na main, encerrando o programa caso seja lançada.

2.3.4 ErroLeituraArquivo

Lançada quando ocorre um erro na leitura dos arquivos de entrada. Através de um if que checa se a leitura ocorreu com sucesso, lança a exceção caso a condição seja falsa. E a mesma só será capturada e tratada na main, encerrando o programa caso seja lançada.

2.3.5 ErroLeituraLinhaCandidatos

Lançada quando ocorre um erro na leitura de uma linha dos arquivos de entrada dos candidatos. Através de um if que checa se a leitura ocorreu com sucesso, seguindo o padrão esperado, lança a exceção caso esta condição seja falsa. E a mesma será capturada e tratada na função lerCandidato(), ignorando a linha e continuando a execução do programa, pois pode haver partidos sem candidatos.

2.3.6 ErroLeituraLinhaPartidos

Lançada quando ocorre um erro na leitura de uma linha dos arquivos de entrada dos partidos. Através de um if que checa se a leitura ocorreu com sucesso, seguindo o padrão esperado, lança a exceção caso esta condição seja falsa. E a mesma só será capturada e tratada na main, encerrando o programa, pois não pode haver candidatos sem partidos.

2.3.7 Nulos

Ocorre nos construtores de RelatoriosPartidos e RelatoriosCandidatos quando, por algum erro desconhecido, é inserido uma lista vazia em algum deles. Através de um if que checa se a lista está vazia, a exceção é lançada caso a condição seja verdadeira, sendo capturada e tratada na main, encerrando o programa.

3 TESTES

O programa foi submetido a uma bateria de testes disponibilizada pelo professor João Paulo Andrade, a qual contém 7 casos de testes ao todo. As cidades que compõem a bateria de testes são: Belo Horizonte, Cariacica, Rio de Janeiro, São Paulo, Serra, Vila Velha e Vitória.

Além de apresentar os inputs e outputs relativos a cada uma dessas cidades, a bateria de testes continha também scripts que automatizam o processo de teste, de forma a rodar sequencialmente os testes para cada uma das cidades e apontar erros entre as saídas construídas e as esperadas do programa.

Assim sendo, o script de teste fornecido apontou congruência em todos os casos de teste, sem quaisquer diferenças até onde se notou.

```
carlos@carlos-Inspiron-5420:~/Projetos/cpp/Trabalho 2/script$ ./test.sh
Script de teste PROG3 2020/2 - Trabalho 2

[I] Testando trabalho...
[I] Testando trabalho: teste belo-horizonte
[I] Testando trabalho: teste belo-horizonte, tudo OK em output.txt
[I] Testando trabalho: teste cariacica
[I] Testando trabalho: teste cariacica, tudo OK em output.txt
[I] Testando trabalho: teste rio-de-janeiro
[I] Testando trabalho: teste rio-de-janeiro, tudo OK em output.txt
[I] Testando trabalho: teste sao-paulo
[I] Testando trabalho: teste sao-paulo, tudo OK em output.txt
[I] Testando trabalho: teste serra
[I] Testando trabalho: teste serra, tudo OK em output.txt
[I] Testando trabalho: teste vila-velha
[I] Testando trabalho: teste vila-velha, tudo OK em output.txt
[I] Testando trabalho: teste vitoria
[I] Testando trabalho: teste vitoria, tudo OK em output.txt
[I] Testando trabalho: pronto!
```

Figura 3 - Resultados dos casos de teste fornecidos pelo professor.

4 BUGS

Apesar de ser tratar a exceção para casos de invalid_argument, a função nativa stoi() retorna exceção apenas em casos em que as letras estão no inicio, ex:

“56hhg767”, usando a função stoi() receberemos 56.

“er4556yt7”, usando a mesma função, receberemos uma exceção.

.

5 DISCUSSÕES

Pelo fato da dupla ter desenvolvido a lógica para o programa anteriormente, acreditamos ter entendido e aplicado bem nossos conhecimentos adquiridos ao longo do período, não tendo encontrado grande dificuldade na execução deste trabalho, precisando apenas de tempo e maior pesquisa para compreender plenamente o uso de exceções como classes próprias. Percebemos que ao longo do desenvolvimento do trabalho entendemos mais e melhoramos nossas habilidades na programação em C++, continuando o uso do Git e GitHub, estudando-os para a colaboração no desenvolvimento dos códigos de forma remota.

6 CONCLUSÕES

Em conclusão, é possível apontar que o programa apresentado se exibe capaz de processar com êxito dados obtidos da Justiça Eleitoral referentes à votação de vereadores, de forma a levantar informações e gerar relatórios sobre as eleições de 2020 em todos os municípios testados.

A implementação em C++ do sistema em questão permitiu entender o potencial dessa linguagem no tratamento de dados, de forma a confirmar a utilização da Programação Orientada a Objetos na resolução de problemas reais do mundo contemporâneo.

REFERÊNCIAS

STD::SORT. 1 mar. 2011. Disponível em: https://www.cplusplus.com/reference/algorithm/sort/. Acesso em: 1 mar. 2022.

TRIM a string in C++ – Remove leading and trailing spaces. 1 fev. 2022. Disponível em: https://www.techiedelight.com/trim-string-cpp-remove-leading-trailing-spaces. Acesso em: 14 mar. 2022.

LOCALE. 1 fev. 2011. Disponível em: https://www.cplusplus.com/reference/locale/. Acesso em: 11 mar. 2022.

STD::NUMPUNCT. 1 fev. 2011. Disponível em: https://www.cplusplus.com/reference/locale/numpunct/. Acesso em: 11 mar. 2022.

PRINT integer with thousands and millions separator. 8 jul. 2013. Disponível em:https://stackoverflow.com/questions/17530408/print-integer-with-thousands-and-millions-separator. Acesso em: 11 mar. 2022.

THROWING Exceptions in C++. 9 set. 2021. Disponível em: https://rollbar.com/blog/error-exceptions-in-c/. Acesso em: 13 mar. 2022.

STD::GETLINE (string)., 1 fev. 2011. Disponível em: https://www.cplusplus.com/reference/string/string/getline/. Acesso em: 4 mar. 2022.