IMD0030 - 2020.2 - Atividade 2

Esta atividade tem como foco nos assuntos (mas não exclui os assuntos anteriores):

- Sobrecarga de funções e passagem de parâmetro
- Sobrecarga de operadores

Introdução

Na atividade anterior exploramos alguns aspectos da modelagem de problemas através de classes e objetos. A atividade consistia principalmente de uma Lista Ligada e dois tipos de classes Playlist e Música que eram gerenciados pelo usuário.

Pare este trabalho você terá duas opções de implementação: a primeira opção você usará como base o código do seu trabalho da atividade 1; na segunda você usará como base um código do trabalho da unidade 1 de um colega de turma, neste caso você deve sinalizar, no README de seu trabalho, de quem é o código base que está usando!

Tarefa

Nesta atividade iremos explorar um pouco mais sobre os conceitos de orientação à objetos em c++ usando os conceitos de sobrecarga de operadores e de funções aplicadas aos objetos que já criamos e conhecemos de acordo com as especificações abaixo:

Melhorias na classe que representa a lista ligada

Nossa classe que representa uma Lista Ligada de músicas tem um método relacionado à adicionar novos elementos à lista.

1 - Os novos métodos são:

A. Adicionar **elementos**: Este método é uma versão sobrecarregada do método que adiciona um elemento à lista, porém esse método recebe como parâmetro, por referência, uma Lista Ligada. Ao final da operação espera-se que todos os elementos da lista recebida sejam adicionados à lista atual.

B. Remover **elementos**: Este método é uma versão sobrecarregada do método que remove um elemento da lista, porém este método recebe como parâmetro, por referência, uma lista de elementos que devem ser removidos.
C. **Construtor cópia**: Este é um método especial que recebe como parâmetro uma referência para uma lista e retorna uma cópia da mesma. Ele é útil quando fazemos atribuições entre objetos do tipo lista ou quando retornamos uma lista em uma função, sem fazer alocação dinâmica, por exemplo.

2 - Usando sobrecarga de operadores implemente também na lista as seguintes operações:

A. **Operador "+"**: implementa a concatenação de duas listas, por exemplo, sejam duas listas "a" e "b", a operação a + b causa que uma nova lista seja criada contendo todas as músicas da lista a, seguido de todas as músicas da lista b, na mesma ordem. Observe que a operação não deve modificar nem a lista "a" nem a lista "b" originais, retornando uma nova lista como resultado. Veja que essa operação não faz distinção entre os elementos da lista, logo, no resultado podem haver elementos repetidos.
B. **Operador de extração ">>"**: extrai o último elemento da lista atribuindo seus valores ao nó recebido como argumento. Caso não existam elementos na lista, o valor preenchido no nó recebido pelo operador deve ser nullptr.
C. **Operador de inserção "<<"**: insere um nó no fim da lista. Caso o valor recebido seja nullptr, nada deve ser feito.

Melhorias na classe que representa a Playlist

Nossa classe que representa uma Playlist e possui uma lista ligada deve espelhar as melhorias que foram feitas na classe Lista Ligada.

3 - Implemente os seguintes métodos adicionais:

A. Adicionar **músicas:** Este método é uma versão sobrecarregada do método que adiciona uma música à Playlist, porém esse método recebe como parâmetro, por referência, outra Playlist. Ao final da operação espera-se que todos os elementos da playlist recebida sejam adicionados no final da playlist atual.

B. Remover **músicas**: Este método é uma versão sobrecarregada do método que remove uma música da Playlist, porém esse método recebe como parâmetro, por referência, outra Playlist e remove todas as músicas da playlist atual, que estejam na lista recebida por parâmetro. O método deve retornar um número com a quantidade de elementos que foram removidos.

C. **Construtor cópia**: Este método é similar ao construtor cópia criado para a lista, porém feito para a playlist.

4 - Implemente os seguintes operadores para a Playlist:

A. **Operador "+"**: implementa a **união** de duas PlayLists. Diferente da **concatenação** de Listas Ligadas, a **união** de playlists não permite músicas repetidas na playlist resultante. A operação não deve modificar a playlist original, retornando uma nova lista como resultado.

B. **Operador "+"**: Uma versão sobrecarregada do operador "+" porém recebendo uma música como parâmetro. Nesse caso, a música deve ser adicionada ao final da playlist que é retornada como resultado do operador. (a playlist original continua inalterada).

C. **Operador "-"**: implementa a diferença entre duas playlists. Sejam duas Playlists a e b, a operação a - b retorna a uma nova playlist contendo todos os elementos da playlist a que não estão na playlist b. Da mesma forma que a operação de união, a playlist original não deve ser modificada após a operação.

D. **Operador "-"**: uma versão sobrecarregada do operador "-" porém recebendo uma música como parâmetro. Nesse caso, a música deve ser removida da playlist que é retornada como resultado do operador. (a playlist original continua inalterada).

E. **Operador de extração ">>"**: remove a última música da playlist atual e preenche na Música recebida como argumento. Caso não existam músicas na playlist, nullptr deve ser preenchido no parâmetro.

F. **Operador de inserção "<<"**: insere a música recebida na última posição da playlist atual. Caso nullptr seja recebido, nada deve ser feito.

Da mesma forma que na Atividade 1, você deve implementar um conjunto de testes através de um programa para mostrar as implementações de cada um dos requisitos listados acima.

De forma análoga à atividade anterior, não utilize classes da STL ou Boost, qualquer dúvida sobre as possibilidades de uso de bibliotecas externas perguntem no servidor do discord.

Autoria e política de colaboração

Esta atividade é individual. A cooperação entre estudantes da turma é estimulada, sendo aceitável a discussão de ideias e estratégias. Contudo, tal interação não deve ser entendida como permissão para utilização de (parte de) código fonte de outros colegas, o que pode caracterizar situação de plágio. Trabalhos copiados em todo ou em parte de outros colegas ou da Internet serão rejeitados.

Entrega

Você deverá submeter um único arquivo compactado no formato .zip contendo, de forma organizada, todos os códigos fonte e o arquivo makefile, sem erros de compilação e devidamente testados e documentados, através da opção Tarefas na Turma Virtual do SIGAA.

Seu arquivo compactado deverá incluir também um arquivo texto README contendo: a identificação completa do estudante; a descrição de como compilar e rodar o programa, incluindo um roteiro (exemplo) de entradas e comandos que destaquem as funcionalidades; a descrição das limitações (caso existam) do programa e quaisquer dificuldades encontradas. Inclua no readme um conjunto de tópicos explicando como cada um dos itens 1 à 6 e seus subitens foram implementados através de um conjunto de casos de teste. Por exemplo:

1. A > caso de teste: na linha 144 do main.cpp
   B > caso de teste: inicie o programa selecione o menu 1 > menu 2 > entre com o valor "Abc"

Orientações gerais

1) Utilize estritamente recursos da linguagem C++.

2) Durante a compilação do seu código fonte, você deverá habilitar a exibição de mensagens de aviso (warnings), pois eles podem dar indícios de que o programa potencialmente possui problemas em sua implementação que podem se manifestar durante a sua execução.
3) Outra flag _MUITO UTIL_ é a -fsanitize=address(use g++ <arquivo.cpp> -fsanitize=address -g), que linka o address sanitizer(asan) no executável final do programa. Usando essa flag na compilação, seu programa se comportará de forma semelhante à programas escritos em linguagem de script (ou linguagens interpretadas como java), gerando erros que normalmente c e c++ deixam de lado por serem linguagens mais permissivas. Erros como acesso à posições inválidas de array, uso de ponteiros não inicializados e até uma melhor descrição quando um segmentation fault ocorre.
4) Aplique boas práticas de programação. Codifique o programa de maneira legível (com indentação de código fonte, nomes consistentes, etc.). Modularize e comente seu código.

Avaliação

O trabalho será avaliado sob critérios:

(i) Completude dos requisitos propostos através de casos de teste fornecidos pelo próprio aluno.

(ii) A aplicação correta de boas práticas de programação, incluindo legibilidade, organização e documentação de código fonte.

(iii) A presença de mensagens de aviso (warnings) ou de erros de compilação e/ou de execução, a modularização inapropriada e a ausência de documentação serão penalizados.