Atividade 3: Concordo

Clone do Discord, para humanos mais civilizados.

Nessa atividade você irá criar um sistema chamado "Concordo" com recursos similares ao Discord, porém vai funcionar somente em modo texto e sem recursos de rede.

Para realizar a atividade vocês irão utilizar os seguintes recursos do C++, que são assuntos das próximas aulas:

- Herança e Classes abstratas e Manipulação de Arquivos
- Standard Template Library (STL)
    - vector

Links com os recursos para esses tópicos serão enviados no canal da atividade 3 no Discord.

Resumidamente, as seguintes entidades devem existir no sistema:

- Usuários: informações de uma conta no sistema.
- Servidores: Com vários canais.
    - CanalTexto (herda de Canal): Com várias mensagens
        - Mensagens: Escrita por um usuário, em uma data/hora
    - CanalVoz (herda de Canal):
        - Não teremos recursos reais de "voz". Esse tipo especializado do Canal guarda somente a última mensagem postada

O sistema deverá ser operado através de comandos de uma única linha, compostos por um nome, seguido de parâmetros. O sistema deve interpretar, processar e gerar uma saída para cada comando, de acordo com o resultado de seu processamento. Cada comando será especificado em detalhes nesse documento. Exemplo de um comando:

```
create-user <email> <senha_sem_espaços> <nome com espaços>
```

As funcionalidades do sistema serão separadas em três partes, que serão entregues separadamente.

Na primeira parte, você deve implementar o processamento dos comandos relacionados a Usuário (todos), Servidores (todos) e Canais (somente a criação de canais). Não é necessário nessa primeira parte implementar o recurso de entrar em um canal nem enviar e ler mensagens dentro dele.

Na segunda parte deverá ser implementado o restante dos comandos do sistema, complementando as operações em Usuário, Servidor, Canais e Mensagens.

Na terceira parte deverá ser implementado o recurso de persistência dos dados em disco e recuperação desses dados para a memória quando o sistema inicia.

Estrutura dos atributos das classes

- **Usuario**
  - Atributos:
    - int id
    - string nome
    - string email
    - string senha
- **Mensagem**
  - Atributos
    - string dataHora
    - int enviadaPor (guardar o ID do usuário que enviou a mensagem)
    - string conteudo
- **Canal** (classe base)
  - Atributos
    - string nome
- **CanalTexto** herda de: **Canal**
  - Atributos
    - std::vector <Mensagem> mensagens
- **CanalVoz** herda de: **Canal**
  - Atributos
    - Mensagem ultimaMensagem
- **Servidor**
  - Atributos:
    - int usuarioDonoId
    - string nome
    - string descricao
    - string codigoConvite (código necessário para se entrar no servidor)
    - std::vector<Canal *> canais;
    - std::vector<int> participantesIDs (lista de IDs de usuários que já estão no servidor)

Observe que você precisará também de uma classe chamada **Sistema** que armazenará também as seguintes informações:

- Um vector com todos os usuários do sistema
- Um vector com todos os servidores do sistema
- Qual o usuário atualmente logado no sistema
- Qual o servidor que o usuário está visualizando no momento
- Qual o canal que o usuário está no visualizando no momento

A classe **Sistema** deve ter somente um objeto criado durante o ciclo de vida da aplicação e é responsável por manter os dados "raiz" da aplicação.

Repare que a classe Usuário tem um atributo chamado "id" que é inteiro. Esse ID deve ser gerado automaticamente pelo sistema durante o cadastro e deve ser incremental, ou seja, o primeiro usuário criado deve ter id==1, o segundo id==2 e assim por diante.

Todas as referências para um usuário devem ser feitas por esse ID, por exemplo na classe Servidor existe um atributo chamado usuarioDonoId que é o ID do usuário que criou o servidor. Assim como existe também um vetor de inteiros chamado participantesIDs, que trata-se de uma lista de IDs dos usuários que estão no servidor.

Por esse motivo você precisa criar no sistema uma forma de encontrar o objeto Usuario, da lista de usuários, buscando pelo seu ID.

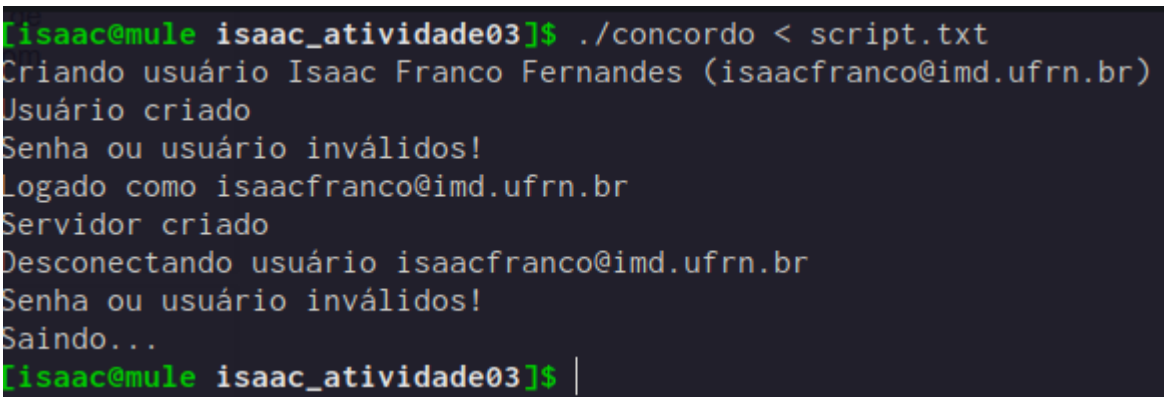
Nas Partes 1 e 2, você implementará uma série de comandos do sistema, todos os comandos são possíveis de serem executados em qualquer ponto do programa, no entando alguns comandos só fucionam caso alguns comandos anteriores tenham sido executados, por exemplo, o comando create-server só funciona se o usuário estiver logado, assim como o comando create-channel só funciona se o usuário estiver em algum servidor. Essas informações(se o usuário está logado, se ele está em algum servidor, etc) são chamadas de *estado* da aplicação, e devem ser controladas pelo programa à medida que o usuário interage com o sistema

Funcionalidades (parte 1):

Geral - Neste trabalho, usaremos comandos de texto ao invés de menus, por isso o sistema não será dividido em “telas” mas receberá comandos de texto para executar as funcionalidades. Cada comando altera o estado interno do programa e retorna um texto com o resultado.
Exemplo:

```
[isaac@mule isaac_atividade03]$ ./concordo
create-user isaacfranco@imd.ufrn.br senha12345 Isaac Franco Fernandes
Criando usuário Isaac Franco Fernandes (isaacfranco@imd.ufrn.br)
Usuário criado
login isaacfranco@imd.ufrn.br senhaerrada
Senha ou usuário inválidos!
login isaacfranco@imd.ufrn.br senha12345
Logado como isaacfranco@imd.ufrn.br
create-server disciplina_lp1
Servidor criado
disconnect
Desconectando usuário isaacfranco@imd.ufrn.br
login joaquim@uol.com.br senhadojoaquim
Senha ou usuário inválidos!
quit
Saindo...
[isaac@mule isaac_atividade03]$
```

Para testar o seu sistema, você pode digitar os comandos, um a um, e observar a saída de cada um deles (como no exemplo acima) ou se desejar pode criar um arquivo de texto com uma sequência de comandos e usar como entrada do seu sistema utilizando redirecionamento, assim:



Repare que nesse caso só é exibido na tela os resultados dos comandos, já que os comandos em si estão no arquivo script.txt (um por linha). Se você implementou seu sistema para receber os comandos do teclado normalmente, esse recurso de redirecionamento deve funcionar sem a necessidade de se alterar nada no sistema, é só uma dica para facilitar os testes sem digitar as mesmas coisas várias vezes.

A1 - Se não estiver logado (Assim que entra no sistema)

Assim que executa o sistema o usuário não está logado e somente as os seguintes comandos não retornam uma mensagem de erro:

- A1.1 Sair do sistema : comando *`quit`*
- A1.2 Criar usuário : comando *`create-user <email> <senha_sem_espaços> <nome com espaços>`*
- A1.3 Entrar no sistema : `login <email> <senha>`

A1.1 - Sair do sistema

Fechar a aplicação, **este comando pode ser executado a qualquer momento pelo usuário**.

Exemplo de entrada/saída

```
quit
"Saindo do Concordo"
```

A1.2 - Criar usuário

Deve ser informado e-mail, senha e nome do usuário e tentar cadastrar o usuário no sistema. Caso consiga, deve emitir um texto informando que o usuário foi inserido com sucesso.

Lembre-se que o ID do usuário deve ser criado automaticamente pelo sistema, como dito anteriormente.

O cadastro só deve ser efetivado se o e-mail não existir no cadastro geral de usuários.

Exemplo de entrada/saída

`create-user julio.melo@imd.ufrn.br 12ab34cd Julio Melo` `"Usuário criado"`
`create-user julio.melo@imd.ufrn.br 12ab34cd Julio Cesar` `"Usuário já existe!"`

A1.3 - Entrar no sistema

Esse procedimento verifica se já existe um usuário no cadastro geral com esse e-mail e senha digitados. Se existir, então o usuário efetuou o login com sucesso.

Observação: Efetuar o login significa que você precisa armazenar (da forma que preferir) a informação de que usuário está logado no sistema para que as outras operações possam saber quem está logado no momento. Por exemplo, se um servidor for criado, é necessário que o dono do mesmo seja o usuário logado.

Exemplo de entrada/saída

`login julio.melo@imd.ufrn.br 12ab34cd` `"Logado como julio.melo@imd.ufrn.br"`
`login julio.melo@imd.ufrn.br 1326` `"Senha ou usuário inválidos!"`
`login julio.engcomp@gmail.com 1326` `"Senha ou usuário inválidos!"`

A2 - Interações básicas com Servidores (Se estiver logado)

Caso o usuário já tenha feito seu cadastro e realizado o procedimento de entrar no sistema, as seguintes opções são apresentadas para ele:

- A2.1 - Desconectar do Concorda : comando `disconnect`
- A2.2 - Criar servidores : comando `create-server <nome>`
- A2.3 - Mudar a descrição do servidor : comando `set-server-desc <nome> "<descrição>"`
- A2.4 - Setar código de convite para o servidor : comando `set-server-invite-code <nome> <codigo>`
- A2.5 - Listar servidores : comando `list-servers`
- A2.6 - Remover servidor : comando `remove-server <nome>`

- A2.7 - Entrar em um servidor : comando `enter-server <nome>`

A2.1 - Desconectar do Concorda

Desconecta o usuário atual, ou seja, altera a variável que você criou que armazena o usuário atualmente logado. O sistema continua executando, mas é necessário que seja feito login novamente para que o restante dos comandos (com exceção de create-user) sejam executados corretamente novamente.

Exemplo de entrada/saída

```
disconnect
"Desconectando usuário julio.melo@imd.ufrn.br"
```

```
disconnect
"Desconectando usuário isaacfranco@imd.ufrn.br"
disconnect
"Não está conectado"
```

A2.2 - Criar servidores (nome)

Seu sistema deve ter uma funcionalidade de criar um novo servidor passando o nome dele. O comando `create-server <nome-do-servidor>` cria um novo servidor se ele não existir com esse nome.

Observe que o servidor tem um "dono", e na criação de um novo servidor esse dono deve ser o usuário logado.

Você pode mudar a descrição para um servidor já criado (se ele for seu) com o comando set-server-desc <nome-do-servidor>.

Observe também que todo servidor é criado sem codigoDeConvite ou seja, qualquer usuário pode entrar no servidor. Mais à frente está detalhado como adicionar um código de convite no servidor.

Exemplo de entrada/saída

```
create-server minha-casa
"Servidor criado"
```

```
create-server minha-casa
"Servidor com esse nome já existe"
```

A2.3 - Mudar a descrição do servidor

Deve ser verificado se o servidor que você está tentando mudar a descrição é seu.

```
set-server-desc minha-casa “Este servidor serve como minha casa, sempre estarei nele”
“Descrição do servidor ‘minha-casa’ modificada!”
```

```
set-server-desc minha-casa2 “Este servidor serve como minha casa, sempre estarei nele”
“Servidor ‘minha-casa2’ não existe”
```

```
set-server-desc minha-casa55 “Trocando a descrição de servidor dos outros”
“Você não pode alterar a descrição de um servidor que não foi criado por você”
```

A2.4 - Setar código de convite para o servidor

Deve ser verificado se o servidor que você está tentando mudar o código de convite é seu.

Se você utilizar o comando sem nenhum código, então o servidor muda seu código para "" e fica liberado para qualquer pessoa entrar.

```
set-server-invite-code minha-casa 4567
"Código de convite do servidor 'minha-casa' modificado!"
```

```
set-server-invite-code minha-casa
"Código de convite do servidor 'minha-casa' removido!"
```

A2.5 - Listar servidores

Retorna o *vector* de servidores disponíveis, exibindo os seus nomes, descrição e se ele é aberto (tem código de convite) ou não.
Exemplo de entrada/saída

```
list-servers
minha-casa
minha-casa2
RPG-galera
Bate-papo-uol
```

A2.6 - Remover servidor

Deve remover um servidor (informando o seu nome).

Só pode ter sucesso na remoção se o dono do servidor for o usuário logado.
Exemplo de entrada/saída

```
remove-server minha-casa
“Servidor ‘minha-casa’ removido”
```

```
remove-server minha-casa
“Você não é o dono do servidor ‘minha-casa’”
```

```
remove-server minha-casa3
“Servidor ‘minha-casa3’ não encontrado”
```

2.7 - Entrar em um servidor

Na tentativa de entrar em um servidor, se o servidor for aberto (não tiver código de convite), o ID do usuário deve ser inserido na lista de participantes do servidor automaticamente e é necessário salvar a informação que o usuário logado está no servidor escolhido.

Se o servidor não for aberto, só deve adicionar como participante do servidor se o mesmo digitar o código correto.

Todas as vezes que um usuário tentar entrar em um servidor fechado ele precisa informar o código de convite.

Se o usuário logado criou o servidor ele pode entrar nele sem código de convite, mesmo que o mesmo não seja aberto.

Exemplo de entrada/saída

```
enter-server dotalovers
“Entrou no servidor com sucesso”
```

```
enter-server concordo-members
“Servidor requer código de convite”
```

```
enter-server concordo-members 123456
“Entrou no servidor com sucesso”
```

Opções para quando se está dentro de um servidor ainda nas funcionalidades da parte 1:

- A2.8 - Sair do servidor : comando `leave-server`
- A2.9 - Listar pessoas no servidor : comando `list-participants`

A gestão de canais será implementada nas funcionalidades (parte 2).

A2.8 - Sair do servidor

Deve desconectar do servidor que o usuário está atualmente conectado. Repare que o usuário se mantém na lista de participantes. O registro de qual servidor o usuário está visualizando no momento (na classe Sistema) deve mudar para um valor que represente "nenhum".

Exemplo de entrada/saída (caso ele esteja visualizando algum servidor)

```
leave-server
“Saindo do servidor ‘minha-casa’”
```

Exemplo de entrada/saída (caso ele não esteja visualizando nenhum servidor)

```
leave-server
“Você não está visualizando nenhum servidor”
```

A2.9 - Listar pessoas no servidor

Exibe todos os usuários que estão no servidor que o usuário está atualmente conectado(somente o nome de cada).

Dica: A classe Servidor deve conter uma lista dos ids dos usuários que estão no servidor. Essa lista é de inteiros, mas você pode utilizar esses inteiros para obter o usuário realizando uma busca de usuários por ID na lista de usuários geral do sistema.

Dica 2: As funcionalidades que permitem que mais de um usuário fique “online” ao mesmo tempo só serão implementadas na parte 3 do trabalho. Para poder testar essa funcionalidade, você pode adicionar participantes ao servidor diretamente no código.

Exemplo de entrada/saída

```
list-participants
tomé
bebé
eu
eu-mesmo
irene
```

Funcionalidades (parte 2)

Com as funcionalidades da parte 1 prontas, seu sistema deve ter implementado as operações em usuários e servidores. Nesta parte, iremos implementar os recursos de canais e mensagens.

B1 - Gestão de canais (se tiver entrado no servidor)

- B1.1 - Listar canais do servidor : comando `list-channels`
- B1.2 - Criar um canal no servidor : comando `create-channel <nome> <tipo>`
- B1.3 - Entrar em um canal do servidor: comando `enter-channel <nome>`
- B1.4 - Sair do canal : comando `leave-channel`

B1.1 - Listar canais do servidor (de áudio ou texto)

Exibe todos os canais do servidor, mostrando primeiro os nomes dos canais de texto, em seguida os nomes dos canais de voz.

Exemplo de entrada/saída

```
list-channels
#canais de texto
casa-de-mae-joana
aqui-nós-faz-o-trabalho
#canais de audio
Professor-Me-Ajude
Revolta-dos-Alunos
```

B1.2 - Criar um canal do servidor

Permite criar um canal no servidor informando seu nome e seu tipo (voz ou texto).

Observe que canais dentro do servidor não podem ter o mesmo nome somente se for do mesmo tipo, ou seja, é possível um canal de texto ter o nome "bate-papo" e um canal de voz também ter o nome "bate-papo" no mesmo servidor, mas não é possível dois canais de texto ter o mesmo nome no mesmo servidor, por exemplo.

Observação: Os canais de texto e voz são classes diferentes que herdam da classe base **Canal**. Elas têm uma implementação muito parecida, a única diferença é que ao se criar

uma mensagem em um canal de voz as mensagens anteriores são removidas (ele só armazena a última).

Exemplo de entrada/saída

```
create-channel casa-de-mae-joana texto
“Canal de texto ‘casa-de-mae-joana’ criado”
```

```
create-channel casa-de-mae-joana2 voz
“Canal de voz ‘casa-de-mae-joana2’ criado”
```

```
create-channel casa-de-mae-joana2 voz
“Canal de voz ‘casa-de-mae-joana2’ já existe!”
```

```
create-channel casa-de-mae-joana texto
“Canal de texto ‘casa-de-mae-joana’ já existe!”
```

B1.3 - Entrar em um canal

Entra em um canal presente na lista de canais do servidor.

Exemplo de entrada/saída

```
enter-channel casa-de-mae-joana
“Entrou no canal ‘casa-de-mae-joana’”
```

```
enter-channel introspecção
“Canal ‘introspecção’ não existe”
```

B1.4 - Sair do canal:

Sai do canal, ou seja, seta a variável que guarda o canal atual do usuário logado como "" (nenhum).

Exemplo de entrada/saída

```
leave-channel
“Saindo do canal”
```

B2 - Gestão de mensagens (se tiver entrado no servidor e em algum canal)

Quando o usuário entra em um canal os seguinte comandos são possíveis:

- B2.1 - Enviar mensagem para o canal : comando `send-message <mensagem>`

- B2.2 - Visualizar mensagens do canal : comando `list-messages`

B2.1 - Enviar mensagem para o canal

Cria uma mensagem e adiciona na lista de mensagens do canal atual. A mensagem deve ser criada com conteúdo digitado, a data/hora atual e com o atributo enviadaPor com o ID do usuário logado.

Exemplo de entrada/saída

```
send-message Oi pessoal querem TC?
```

B2.2 - Visualizar mensagens do canal:

Lista todas as mensagens do canal exibindo:

- O nome do usuário que criou a mensagem
- Data/hora da criação da mensagem
- Seu conteúdo

Exemplo de entrada/saída

```
list-messages
Julio<08/03/2021 - 11:53>: Assim não tem condições, como que a
galera vai conseguir terminar isso tudo em 4 semanas?
Isaac<08/03/2021 - 12:00>: Eles conseguem confio na galera
Renan<08/03/2021 - 12:00>: Semestre passado fizemos assim e
ninguém entregou :/
```

```
list-messages
"Sem mensagens para exibir"
```

Funcionalidades (parte 3)

C1 - Implementar Persistência dos dados em disco

Nessa última parte do projeto você deve implementar um mecanismo para salvar todos os dados da classe **Sistema** (que guarda todo o estado da aplicação) em disco. Isso inclui todos os Usuários, Servidores, Canais e Mensagens.

Para isso você deverá implementar esses métodos em **Sistema**:

- privados:
  - void salvarUsuarios()
  - void salvarServidores()

- público
    - void salvar()

O método **salvar()** simplesmente executa salvarUsuarios() e em seguida salvarServidores().

O método **salvarUsuarios()** utiliza os dados armazenados no *vector* de usuários do sistema para salvar um arquivo chamado "usuarios.txt" com um formato similar ao descrito abaixo:

- Primeira linha deve conter um inteiro com o total "U" de usuários
- Em seguida, para cada usuário, deve existir 4 linhas, uma para cada atributo do usuário na seguinte ordem: id, nome, email, senha

Exemplo do arquivo "usuarios.txt":

```
3
1
Maria Trindade
mariatrindade@imd.ufrn.br
senhabemseguradema ria
2
Joaquim Silveira
joaquimsilveira@gmail.com
senhabemseguradejoaquim
3
Carla Souza
carlasouza@hotmail.com
senhadecarla
```

A função **salvarServidores()** deve utilizar os dados no vetor de servidores e salvar um arquivo chamado "servidores.txt". Esse arquivo não só terá as informações dos servidores, mas também dos canais e mensagens, já que essas informações estão aninhadas nos objetos dos servidores. Ele terá o formato descrito abaixo:

- Primeira linha deve conter um inteiro com o número de servidores
- Em seguida, para cada servidor devemos ter:
    - Uma linha com o ID do usuário dono do servidor
    - Uma linha com o nome do servidor
    - Uma linha com a descrição do servidor
    - Uma linha com o código de convite do servidor (se for aberto deve ser uma linha vazia,)
    - Uma linha com um inteiro informando o número de usuários participantes do servidor
    - Para cada usuário participante do servidor:
        - Uma linha com o ID do usuário participante
    - Uma linha com um inteiro informando o número de canais do servidor
    - Para cada canal no servidor:
        - Uma linha com o nome do canal
        - Uma linha com o tipo do canal: 'TEXTO' ou 'VOZ'
        - Uma linha com um inteiro com o número de mensagens do canal
        - Para cada mensagem no canal
            - Uma linha com o ID do usuário que a escreveu
            - Uma linha com a data/hora da mensagem

- Uma linha com o conteúdo da mensagem

Exemplo:
Considere o seguinte cenário:
2 Servidores (IMD e UFRN):

- Servidor IMD com:
  - 2 participantes: Usuário com ID 1 e Usuário com ID 2
  - 1 canal de texto ("geral") e um de voz ("fale com o professor")
    - Canal "geral" com duas mensagens, uma do usuário 1 com o conteúdo "oi" e outra do usuário 2 com o conteúdo "olá"
    - Canal "fale com o professor" com 1 mensagem apenas do usuário 1 com o conteúdo "alguém aí?"
- Servidor UFRN com:
  - 1 participante: Usuário com ID 3
  - Um canal de texto somente ("informativos")
  - Canal "informativos" com duas mensagens, ambas do usuário 3 com os conteúdos "aulas remotas #ativar" e outra "aulas remotas #continuar"

Para o exemplo acima segue a saída do arquivo servidores.txt:

```
2
1
IMD
Descrição do servidor do IMD

2
1
2
2
geral
TEXTO
2
1
02/02/2021
oi
2
03/02/2021
olá
fale com o professor
VOZ
1
1
03/02/2021
alguém aí?
1
UFRN
Descrição do servidor da UFRN

1
3
1
```

```
informativos
TEXTO
2
3
04/02/2021
aulas remotas #ativar
3
04/02/2021
aulas remotas #continuar
```

Observe que no caso dos canais de voz somente uma mensagem é armazenada, então ele sempre terá um total de 1 mensagem no arquivo.

O método salvar() deve ser executado SEMPRE depois da execução de qualquer comando que alterar algum dado do sistema (que adiciona ou modifica informações).

C2 - Implementar restauração dos dados do disco na inicialização do sistema

Na inicialização do sistema devem ser carregados os dados nos arquivos usuarios.txt e servidores.txt para que o sistema retome o estado desde a última execução. Além disso, ANTES de qualquer comando ser executado os dados do sistema precisam ser carregados novamente do disco, pois eles podem ter sido alterados por outro usuário.

Para isso você deve implementar os métodos na classe **Sistema**:

- privados:
  - void carregarUsuarios()
  - void carregarServidores()
- públicos:
  - void carregar()

O método **carregar()** simplesmente executa carregarUsuarios() e em seguida carregarServidores().

O método **carregarUsuarios()** deve abrir o arquivo usuarios.txt (se existir) e ler todas as suas linhas, preenchendo o vetor de usuários com as informações salvas.

O método **carregarServidores()** deve abrir o arquivo servidores.txt (se existir) e processar todas as suas linhas preenchendo o *vector* de servidores já com as suas informações, usuários participantes, seus canais e mensagens.

LEMBRE-SE:

- Antes de qualquer comando, seja qual for, executar **carregar()**
- Depois de qualquer comando que altera dados executar **salvar()**

Dica: Uma forma simples e mais fácil de depurar pra fazer isso é você encapsular a execução dos comandos em uma classe que chama as funções carregar e salvar para todos em um ponto só do código.

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

Serão realizadas três entregas, e os prazos estarão registrados no SIGAA. A primeira entrega deverá conter as funcionalidades (parte 1) descritas nesse documento, a segunda entrega a parte 2 das funcionalidades e terceira entrega a parte 3. Para cada uma delas 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 e seus subitens foram implementados através de um conjunto de casos de teste.

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.