ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia de Computação e Sistemas Digitais

PCS 3111 - LABORATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A ENGENHARIA ELÉTRICA

EXERCÍCIO PROGRAMA 2 – 1º SEMESTRE DE 2022

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar uma *Rede Social* simples, similar ao Facebook. Neste segundo EP serão feitas algumas melhorias na rede social implementada no primeiro EP.

1 Introdução

Neste segundo EP serão feitas alterações no programa já desenvolvido para melhorar o projeto, considerando outros conceitos da orientação a objetos e permitir novas funcionalidades. As principais novas funcionalidades são: permitir a criação de um número ilimitado de contatos, com o uso do *vector* e a persistência da rede social.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor, herança, classe abstrata, membros com escopo de classe, programação defensiva, persistência em arquivo e os containers da STL. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

2. Projeto

Deve-se implementar em C++ as classes **Perfil**, **Pessoa**, **PessoaVerificada**, **Pagina**, **Postagem**, **Story**, **PerfilInexistente**, **PersistenciaDaRede** e **RedeSocial**, além de criar um `main` que permita o funcionamento do programa como desejado.

Atenção:

- O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. As classes não devem possuir outros membros (atributos ou métodos) **públicos** além dos especificados, a menos dos métodos definidos na superclasse e que precisaram ser redefinidos.
- Pode-se definir atributos e métodos protegidos e privados, conforme necessário.
- Coloque a palavra reservada `virtual` nos métodos conforme necessário.
- Não é permitida a criação de outras classes além dessas.
- Não faça `#defines` de constantes.

O não atendimento a esses pontos pode resultar erro de compilação na correção e, portanto, nota `0` na correção automática.

Por simplicidade, admita que os nomes dos perfis e das páginas não possuem espaços em branco. Isso facilitará a persistência.

Cada uma das classes deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Os arquivos devem ter exatamente o nome da classe. Por exemplo, deve-se ter os arquivos "Perfil.cpp" e "Perfil.h". Note que você deve criar os arquivos necessários.

Apesar de a especificação ser longa, as classes possuem um comportamento muito similar ao especificado no EP1 e algumas outras funcionalidades desenvolvidas em aula.

2.1 Classe Perfil

Um **Perfil** é o elemento básico da rede social. Com isso, um **Perfil** é uma classe **abstrata** e possui apenas um nome e um identificador (id), além de seguidores e publicações. Escolha o método mais adequado para ser **abstrato**.

Para permitir um número ilimitado de contatos e de postagens, a classe deve usar os containers da STL. Os contatos deverão ser guardados em um `vector` e as postagens devem ser armazenadas em `lists`. Por causa dessa mudança, não são mais necessários os parâmetros no construtor.

A classe **Perfil** deve possuir os seguintes métodos **públicos:**

```
Perfil(string nome);
Perfil(int id, string nome);
virtual ~Perfil();

string getNome();

void adicionar(Perfil* contato);
void adicionar(Postagem* p);

list<Postagem*>* getPostagens();
list<Postagem*>* getPostagensDosContatos(int data);
list<Postagem*>* getPostagensDosContatos();
vector<Perfil*>* getContatos();

void imprimir();

int getId();
static int getUltimoId();
static void setUltimoId(int ultimoId);
```

O Perfil tem dois construtores: um que recebe apenas um nome e outro que recebe um nome e um id. O que recebe o nome e o id deve ser usado **apenas pela persistência** (discutida na Seção 3). A criação de um **Perfil** pela interface com o usuário deve chamar o construtor que recebe apenas o nome e deve gerar automaticamente um identificador (id). O primeiro **Perfil** criado deve ter id = 1, o segundo perfil deve ter id = 2 e assim por diante (note que isso *independe* do tipo dele – se **Pessoa**, **PessoaVerificada** ou **Pagina**). Para isso você deverá usar um atributo com escopo de classe (`static`).

O destrutor deve ser implementado como especificado na Aula 5.

O método `getUltimoId` retorna o valor do último ID gerado. Caso nenhum `Perfil` tenha sido criado, ele deve retornar 0. O método `setUltimoId` serve para a persistência, no carregamento de redes sociais.

Ele permite definir o valor do último id gerado e, portanto, tem escopo de classe. Apenas a persistência deve usar esse método.

Os métodos `getId` e `getNome` devem retornar os valores do id e do nome.

Os métodos para adicionar um contato e adicionar uma postagem foram renomeados para `adicionar`, usando o conceito de sobrecarga. O método `adicionar` que recebe um **Perfil** deve adicionar um contato ao **Perfil**. Caso se tente adicionar o próprio **Perfil** como contato dele mesmo ou se o **Perfil** informado já seja um contato, esse método deve jogar uma exceção do tipo `invalid_argument`. O motivo deve ser "Perfil adicionando ele mesmo" no primeiro caso e "Perfil ja adicionado" no segundo caso. Para fazer uma postagem existe o `adicionar` recebendo uma **Postagem** (esse método não faz verificações). Note que diferentemente do EP1, esses métodos são *void*.

O método `getContatos` permite obter os **Perfis** que são contatos deste **Perfil.** O método deve retornar um ponteiro para um `vector` de ponteiros de **Perfis** (ou um `vector` vazio caso o **Perfil** não tenha contatos).

O método `getPostagens` permite obter as **Postagens** feitas pelo **Perfil** (ou um `list` vazio caso o **Perfil** não tenha postagens). O método `getPostagensDosContatos (com data)` retorna as postagens dos contatos do **Perfil**, feitas no dia atual e nos últimos <u>três</u> dias (assim como explicado no EP1) – retornando um `list` vazio caso não haja –, enquanto o método `getPostagensDosContatos` sobrecarregado retorna **<u>todas</u>** as postagens dos contatos (ou um `list` vazio, caso não haja **Postagens** dos contatos).

Implemente o método `imprimir` como feito na Aula 8 (usando o ID), fazendo as adaptações necessárias para se adequar ao uso dos containers STL.

2.2 Classe Pessoa

Essa classe deve ser subclasse de **Perfil**, mas é **<u>concreta</u>**. Ela deve possuir os seguintes <u>métodos públicos específicos a essa classe</u>:

```
Pessoa(string nome);
Pessoa(int id, string nome);
virtual ~Pessoa();
```

Note que você pode redefinir métodos da superclasse, **Perfil**, se necessário, além de criar métodos privados. O destrutor se comporta da mesma forma que a superclasse. Assim como no **Perfil**, a classe **Pessoa** possui dois construtores. O que recebe o `id` deverá ser usado apenas pela persistência.

Em relação ao método `imprimir`, ele deve seguir a especificação da Aula 8. Um exemplo de impressão é apresentado a seguir:

```
Nome: Ana - id: 1
Numero de postagens feitas: 2
Postagens na data 5 - Texto: Ola
Postagens na data 1 - Texto: Story
Postagens na data 3 do contato Maria - Texto: Prova
Postagens na data 2 do contato Maria - Texto: Teste
Postagens na data 1 do contato PCS3111 - Texto: P2
```

2.3 Classe PessoaVerificada

Uma **PessoaVerificada** é um tipo de **Pessoa** que possui adicionalmente a informação do seu email. Para isso, a classe deve ser subclasse de **Pessoa**. A seguir são apresentados os métodos públicos específicos a essa classe (note que a classe pode ter outros métodos privados e pode ter que redefinir métodos da superclasse).

```
PessoaVerificada(string nome, string email);
PessoaVerificada(string nome);
PessoaVerificada(int id, string nome, string email);
virtual ~PessoaVerificada();

string getEmail();
```

Existem três construtores. Um deve receber o nome e o e-mail, o segundo recebe apenas um nome e o email atribuído no construtor deve ser *email.padrao@usp.br*, enquanto o último deve ser utilizado pela persistência. O método `getEmail` deve retornar o endereço de email. O destrutor se comporta da mesma forma que a superclasse.

Em relação ao método `imprimir`, ele deve seguir a especificação da Aula 8. Um exemplo de impressão é apresentado a seguir:

```
email.padrao@usp.br
Nome: Maria - id: 2
Numero de postagens feitas: 2
Postagens na data 3 - Texto: Prova
Postagens na data 2 - Texto: Teste
Postagens na data 1 do contato PCS3111 - Texto: P2
Postagens na data 5 do contato Ana - Texto: Ola
Postagens na data 1 do contato Ana - Texto: Story
```

2.4 Classe Pagina

Uma **Pagina** é um tipo de **Perfil** com alguns comportamentos específicos. Portanto, essa classe deve ser subclasse de **Pessoa**. A seguir são apresentados os métodos públicos específicos a essa classe (note que a classe pode ter outros métodos privados e pode ter que redefinir métodos da superclasse).

```
Pagina(string nome, PessoaVerificada* proprietario);
Pagina(int id, string nome, PessoaVerificada* proprietario);
virtual ~Pagina();

PessoaVerificada* getProprietario();
```

Uma página possui um nome e um proprietário, os quais devem ser informados no construtor. Ao criar a **Pagina**, a **PessoaVerificada** responsável deve ser automaticamente adicionada como contato, **mas no construtor que recebe *id*, não adicione (isso simplifica a persistência).**

Além disso, o método `getProprietario` deve apenas retornar o proprietário definido pelo construtor. O destrutor se comporta da mesma forma que a superclasse (não destrua o proprietário no destrutor).

Em relação ao método `imprimir`, ele deve seguir a especificação da Aula 8. Um exemplo de impressão é apresentado a seguir:

```
Nome: PCS3111 - id: 3 - Proprietario: Maria
```

2.5 Classe Postagem

Uma **Postagem** é uma mensagem publicada na rede social e que é divulgada aos contatos do seu autor. Toda postagem possui um texto, um autor e uma data. A classe **Postagem** deve possuir os seguintes métodos **públicos**:

```
Postagem(string texto, int data, Perfil* autor);
virtual ~Postagem();

Perfil* getAutor();
string getTexto();
int getData();

void imprimir();
```

O construtor deve receber o **Perfil** do autor, um texto e uma data, os quais são informados pelos métodos de acesso `getAutor` e `getTexto` e `getData`. O destrutor deve ser implementado como especificado na Aula 5, fazendo:

```
cout << "Destrutor de postagem: " << texto << endl;
```

O método `imprimir` está especificada na aula 7. Ele deve imprimir em tela o texto da **Postagem** e o autor no seguinte formato (pule uma linha no final):

`Texto: <texto> - Data: <data> - Autor: <nome do autor>`

Onde `<texto>` é o atributo texto da Postagem, `<data>` é o atributo data e `<nome do autor>` é o nome do autor da **Postagem**. Por exemplo, uma impressão de **Postagem** seria:

```
Texto: Ola - Data: 5 - Autor: Ana
```

2.6 Classe Story

O **Story** é um tipo de **Postagem** que possui uma data de fim. Para isso, a classe deve ser subclasse de **Postagem**. A seguir são apresentados os métodos públicos específicos a essa classe (note que a classe pode ter outros métodos privados e pode ter que redefinir métodos da superclasse).

```
Story(string texto, int data, int dataDeFim, Perfil* autor);
virtual ~Story();

int getDataDeFim();
```

O construtor deve receber o autor, o texto e as datas do **Story**. O método `getDataDeFim` deve retornar a data de encerramento do **Story** definida no construtor. O destrutor se comporta da mesma forma que a superclasse.

Redefina o método `imprimir`, o qual deve imprimir no seguinte formato:

```
Texto: <texto> - Data: <data> - Data de Fim: <dataDeFim> - Autor: <nome do autor>
```

Onde <texto> é o atributo texto da **Story**, <data> é o atributo data, <dataDeFim> é o atributo dataDeFim e <nome do autor> é o nome do autor da **Story**. Por exemplo:

```
Texto: Ola - Data: 1 - Data de Fim: 4 - Autor: Ana
```

2.7 Classe RedeSocial

Assim como no EP1, a **RedeSocial** é a classe responsável por ter a lista de **Perfis** existentes na rede. Ela é bastante similar ao definido no EP1; a diferença é o uso de um `vector` para guardar os **Perfis** e dois novos métodos de apoio. Com isso ela deve possuir os seguintes métodos **públicos**:

```
RedeSocial();
virtual ~RedeSocial();

vector<Perfil*>* getPerfis();
Perfil* getPerfil(int id);
void adicionar(Perfil* perfil);

void imprimir();
void imprimirEstatisticas();
```

Agora será usado um `vector` para armazenar os **Perfis**, ou seja, não existe o atributo *capacidade*. Para adicionar um **Perfil** à rede deve ser usado o método `adicionar` (note que o método é agora `void`). Caso o **Perfil** já exista, a classe deve jogar uma exceção `invalid_argument` (a mensagem não importa). Para facilitar, use a função `find` para saber se o **Perfil** já está no `vector` (http://www.cplusplus.com/reference/algorithm/find/). O destrutor deve ser implementado como especificado na aula 5.

Também há um método `getPerfil` que obtém o **Perfil** a partir do *id* dele. Esse método é útil para a interface com o usuário (note que ela é agora baseada em ids) e para a persistência. Caso não exista o **Perfil** com o *id* informado deve-se jogar uma exceção do tipo **PerfilInexistente**.

Os **Perfis** adicionados com sucesso à rede devem ser obtidos pelo método `getPerfis`, que retorna um vector de **Perfis**.

O método `imprimir` da classe **RedeSocial** deve seguir o código fornecido da aula 5, adaptado para trabalhar com o vector.

O método `imprimirEstatisticas` deve imprimir a quantidade de cada tipo de **Perfil**, usando o `dynamic_cast` para calcular a quantidade. Um exemplo de saída é:

```
Pessoa: 2
PessoaVerificada: 2
Pagina: 1
Perfil: 0
```

(Apesar de não ser possível existir objetos do tipo **Perfil**, contabilize nele os tipos de **Perfil** que não são nem **Pessoa**, nem **PessoaVerificada** e nem **Pagina**).

2.8 Classe PerfilInexistente

A classe **PerfilInexistente** representa que não existe um **Perfil** com o id informado na rede social. Ela deve ser subclasse de **logic_error** (da biblioteca padrão) e os métodos públicos específicos a essa classe são apresentados a seguir.

```
PerfilInexistente();
virtual ~PerfilInexistente();
```

A mensagem de erro informada pelo método `what` da exceção deve ser "Perfil Inexistente".

3 Persistência

A persistência, por simplicidade, se limitará aos **Perfis** e suas relações. Ou seja, não se persistirão as postagens publicadas pelos **Perfis**. Uma outra simplificação é que o nome do **Perfil** não deverá conter espaços.

A seguir é apresentado o formato do arquivo, exemplos de arquivos e a especificação das classes.

3.1 Formato do arquivo

A persistência da rede social deve seguir o formato de arquivo especificado a seguir. Entre "<" e ">" são especificados os valores esperados. Por simplicidade, será utilizado um caractere de nova linha ('\n') como delimitador e *assuma* que não existem espaços nos campos que sejam `string`.

```
<último id do Perfil>
<quantidade de pessoas verificadas>
<id da PessoaVerificada 1> <nome> <email>
<id da PessoaVerificada 2> <nome> <email>
...
<quantidade de pessoas>
<id da Pessoa 1> <nome>
<id da Pessoa 2> <nome>
...
<quantidade de páginas>
<id da Pagina 1> <id do proprietário> <nome>
<id da Pagina 2> <id do proprietário> <nome>
...
<id de um Perfil> <id do contato do Perfil>
<id de um Perfil> <id do contato do Perfil>
...
```

Esse formato permite que sejam definidos inúmeros **Perfis**, independentemente do tipo deles. No começo é guardado o id do último **Perfil** cadastrado, obtido por `getUltimoId`. Isso é importante para evitar que existam ids repetidos (não se preocupe em verificar essa consistência).

O arquivo sempre apresenta a quantidade de um tipo de **Perfil** e então o dado de cada um daqueles **Perfis**. Primeiramente são as **PessoaVerificadas**, depois as **Pessoas** (note que são os objetos instâncias *diretas* dessa classe), e então as **Paginas**. No fim, são apresentados os contatos dos **Perfis**. A ordem dos **Perfis** e dos contatos não é relevante, mas essa lista deve apresentar todos os **Perfis**, inclusive o responsável pela **Pagina**.

3.2 Exemplo

Um exemplo desse arquivo, com quatro **Perfis** é apresentado a seguir. Existem três pessoas verificadas: Maria (id 1), Jose (id 3) e Claudio (id 2). Claudio, com id 2 é proprietário da página PCS3111, com id 4.

```
4
3
1 Maria maria@usp.br
3 Jose jose@usp.br
2 Claudio claudio@usp.br
0
1
4 2 PCS3111
1 2
1 3
2 1
2 4
3 1
4 2
```

As relações entre os Perfis são as seguintes:

- Maria é contato de Cláudio e de José;
- PCS3111 é contato de Cláudio;

Note que José, por exemplo, adicionou Maria na sua lista de contatos, por reciprocidade. Por isso os contatos aparecem repetidos. Ao adicionar os contatos, ignore os casos de erro em que o **Perfil** já foi adicionado (devido à repetição) e também de o **Perfil** adicionar ele mesmo.

Um outro exemplo é de um arquivo com apenas uma pessoa.

```
1
0
1
1 Paula
0
```

3.3 Classe PersistenciaDaRede

A classe **PersistenciaDaRede** é a classe responsável pela persistência da **RedeSocial** em um arquivo texto. Ela deve permitir salvar uma rede e carregá-la. Os únicos métodos <u>públicos</u> que a classe deve possuir são:

```
PersistenciaDaRede(string arquivo);
virtual ~PersistenciaDaRede();

void salvar(RedeSocial* r);
RedeSocial* carregar();
```

O construtor deve receber o nome do arquivo que será usado para carregar e salvar a **RedeSocial**. O método `carregar` deve obter a **RedeSocial** que está no arquivo texto, seguindo o formato especificado anteriormente. Caso o arquivo não exista, retorne uma **RedeSocial** vazia. Ao carregar, atualize o último id definido no arquivo. Caso haja algum problema de leitura (erro de formato ou outro problema), jogue uma exceção do tipo `logic_error`.

O método `salvar` salva a **RedeSocial** passada como parâmetro no arquivo informado no construtor. Caso não seja possível escrever no arquivo ou haja algum erro de escrita, jogue uma exceção do tipo `logic_error`.

4 Interface com o usuário

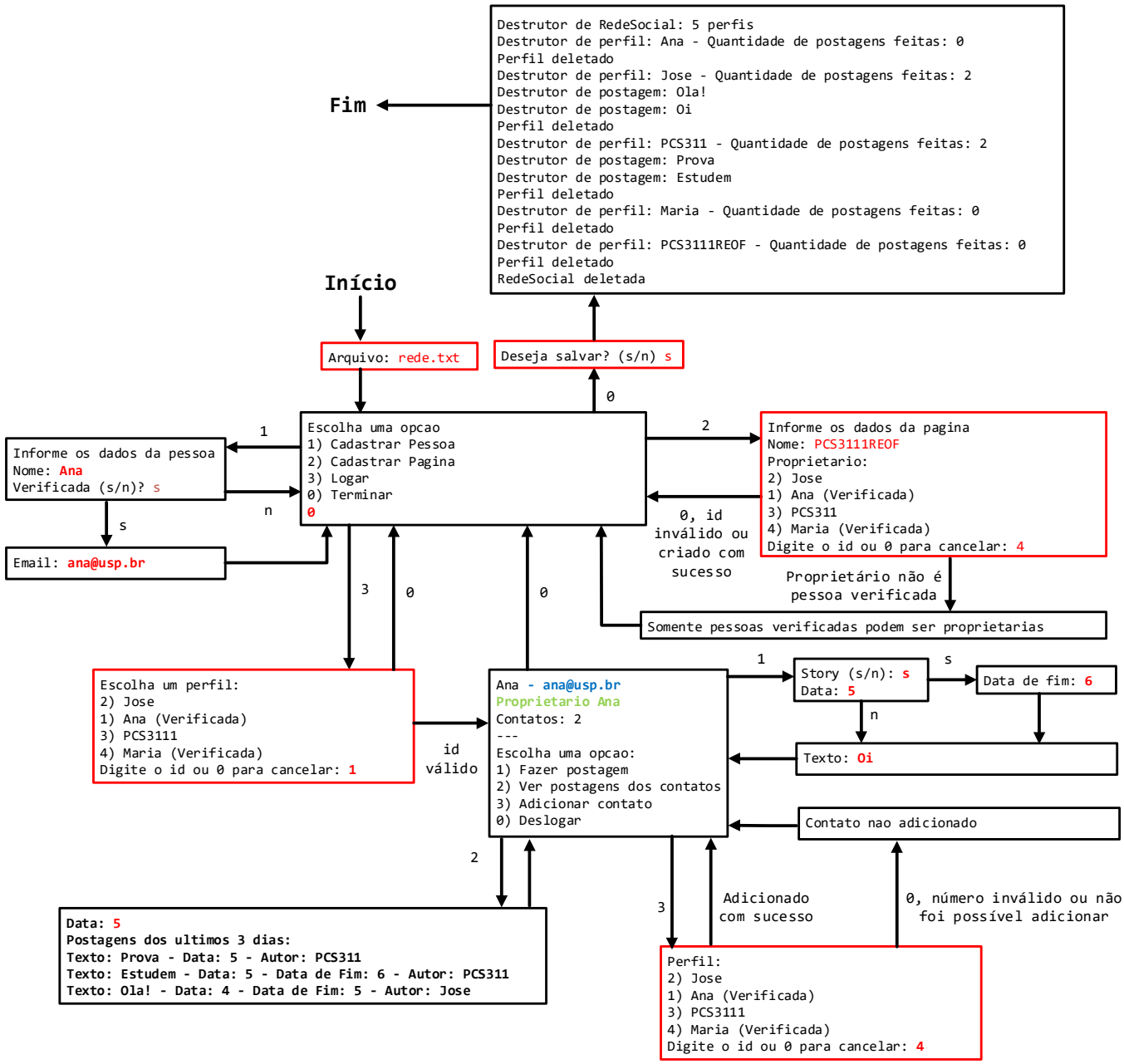
Coloque a main em um arquivo em separado, chamado `main.cpp`. Ela é apresentada esquematicamente no diagrama abaixo. Cada retângulo representa uma "tela", ou seja, o conjunto de informações apresentadas e solicitadas. As telas com borda **vermelha** possuem diferenças em relação ao EP1 (são novas ou foram alteradas). As setas representam as transições de uma tela para outra – os textos na seta representam o valor que deve ser digitado para ir para a tela destino ou a condição necessária (quando não há um texto é porque a transição acontece incondicionalmente). Em **vermelho** são apresentados exemplos de dados inseridos pelo usuário. Em **verde** são apresentadas impressões específicas para uma **Pagina** e em **azul** específica para **PessoaVerificada**:

- Quando for uma **PessoaVerificada**, deve aparecer após o nome "- <email>" como, "Ana – ana@usp.br". Quando for um outro tipo de **Perfil**, é só apresentado o nome como "PCS3111" ou "Jose".
- A informação do proprietário só deve aparecer se o **Perfil** for uma **Pagina** (nos outros casos não deve existir essa linha).

> **Atenção:** A interface com o usuário deve seguir exatamente a ordem definida (e exemplificada). Se a ordem não for seguida, haverá desconto de nota.

Alguns detalhes:

- Em telas que listem os **Perfis**, deve ser apresentado " (Verificada)" após o nome de uma **PessoaVerificada**.
- No diagrama não são apresentados casos de erro (por exemplo, a digitação de um caractere ao invés de um número). Não é necessário fazer tratamento disso. Assuma que o usuário *sempre* digitará um valor correto.
- Por simplicidade, sempre liste todos os perfis nas telas que fazem listagens de perfis. Ou seja, inclua também o perfil logado na opção "Adicionar Contato" e **Paginas** e **Pessoas** na seleção do proprietário.
- Ao sair do programa, chame o destrutor de **RedeSocial**.
- A interface é bastante similar à do EP1. As diferenças:
    - No início do programa se pergunta o arquivo a ser carregado. Se ele não existir, deve ser criada uma **RedeSocial** vazia (já é o comportamento de **PersistenciaDaRede**).
    - No final do programa se pergunta se deseja salvar a **RedeSocial**. Caso se responda sim, a rede deve ser salva no arquivo informado no início da execução.
    - A seleção de **Perfis** é feita pelo **id** deles (use o método `getPerfil` da **RedeSocial**). Portanto, é possível que os ids dos **Perfis** não estejam ordenados (após carregar um arquivo).
- Caso alguma operação jogue uma exceção, apresente a mensagem da exceção e termine o programa.



5 Entrega

O projeto deverá ser entregue até dia **10/07** no Judge disponível em <https://laboo.pcs.usp.br/ep/>. Você deverá fazer duas submissões (Entrega 1 e Entrega 2). **Todas as entregas deverão possuir o mesmo conteúdo**.

> **Atenção**: não copie código de um outro colega. Qualquer tipo de cópia será considerado plágio e ambos os alunos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um outra pessoa! Cópias de trabalhos de anos anteriores também receberão 0.

Entregue todos os arquivos, inclusive o `main` (que deve obrigatoriamente ficar em um arquivo "main.cpp"), em um arquivo comprimido (não entregue o arquivo "teste.cpp"). Cada entrega deve ser

feita em um arquivo comprimido (zip – não pode ser rar). Os arquivos não devem ser colocados em pastas.

Siga a convenção de nomes para os arquivos ".h" e ".cpp". O não atendimento disso pode levar a erros de compilação e, consequentemente, **nota zero**.

Ao submeter os arquivos no Judge será feita uma verificação básica de modo a evitar erros de digitação no nome das classes e dos métodos públicos. Você poderá submeter quantas vezes você desejar. Mas note que a nota gerada por essa verificação **não é a nota final**: não são executados testes – o Judge apenas tenta chamar todos os métodos definidos para todas as classes.

Você pode submeter quantas vezes quiser, sem desconto na nota.

6 Dicas

- Reaproveite o `main` do EP1 para implementar o `main` do EP2. Se você organizou bem o `main` (por exemplo, com uma função específica para selecionar um **Perfil**, como sugerido), a criação do novo `main` será muito rápida.
- Caso o programa esteja travando, execute o programa no modo de depuração do Code::Blocks. O Code::Blocks mostrará a pilha de execução do programa no momento do travamento, o que é bastante útil para descobrir onde o erro acontece!
- Faça `#include` apenas das classes que são usadas naquele arquivo. Por exemplo, se o arquivo .h não usa a classe **X**, mas o .cpp usa essa classe, faça o `include` da classe **X** *apenas* no .cpp. Incluir classes desnecessariamente pode gerar erros de compilação estranhos (por causa de referências circulares).
    - Inclua todas as dependências necessárias. Não dependa de `#includes` feitos por outros arquivos incluídos.
- Crie métodos auxiliares (privados) para organizar os métodos de persistência.
- Teste o método `carregar` e o `salvar` da persistência separadamente. Use um `main` de teste, aproveitando o método `imprimir` da **RedeSocial**.
- Use os métodos da biblioteca padrão para facilitar o uso de `vectors` e `lists`.
- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- É trabalhoso testar uma classe específica usando o `main` *com menus*, já que pode ser necessário passar por várias telas até testar o que se deseja. Para simplificar o teste, crie um `main` que cria os objetos do jeito que você quer testar. Só não se esqueça de entregar o `main` correto!
- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *RuntimeErrors* e outros tipos de erros no Judge que não aparecem no Code::Blocks (especialmente nas versões antigas do Code::Blocks que usam um outro compilador). Veja a mensagem de erro do Judge para descobrir o problema. Caso você queira testar o projeto em um compilador similar ao do Code::Blocks, use o site https://repl.it/ (note que ele não tem depurador *ainda*).

- Em geral *RuntimeErrors* acontecem porque você não inicializou um atributo que é usado. Por exemplo, caso você não crie um vetor ou não inicialize o atributo quantidade, para controlar o tamanho do vetor, ocorrerá um *RuntimeError*.

- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**