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 1 – 1º SEMESTRE DE 2022

1. 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. A rede é composta por diversos perfis que podem seguir outros perfis. Cada perfil faz postagens que são recebidas pelos seus contatos.

1 Introdução

Deseja-se criar uma *Rede Social s*imples, no estilo do Facebook. Este projeto será desenvolvido incrementalmente e **individualmente** nos **dois** Exercícios Programas de PCS 3111.

Para este primeiro EP a rede social deverá permitir a adição de diferentes tipos de perfis: perfis normais, pessoas verificadas, pessoas não verificadas e páginas. Cada perfil poderá ter vários contatos e poderá fazer postagens (texto ou stories) para seus contatos. Por simplicidade, o número de contatos e de postagens será limitado neste EP. Pelo mesmo motivo, o tamanho da rede social será fixo.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor, e herança – o que representa o conteúdo até a Aula 7. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

1. Projeto

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 classe pai e que precisaram ser redefinidos. Note que você poderá definir atributos e método protegidos e privados, conforme necessário.
- Não é permitida a criação de outras classes além dessas.
- Não faça outros #defines além dos definidos neste documento.

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

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

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.

Por simplicidade considere que os textos das postagens e os nomes não possuem espaços.

1.1 Classe Perfil

Um **Perfil** é o elemento básico da rede social. Ele possui um nome. Além disso, ele pode adicionar contatos, fazer postagens e receber postagens feitas pelos contatos.

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

```
Perfil(string nome, int maximoDeContatos, int maximoDePostagens);
virtual ~Perfil();

string getNome();

virtual bool adicionarContato(Perfil* contato);
virtual bool adicionarPostagem(Postagem* p);

virtual Postagem** getPostagens();
virtual Postagem** getPostagensDosContatos(int data , int &quantidade);
virtual int getQuantidadeDePostagens();

virtual Perfil** getContatos();
virtual int getQuantidadeDeContatos();

virtual void imprimir();
```

O método `getNome` deve retornar o valor do nome informado no construtor. O destrutor deve ser implementado como especificado na Aula 5, destruindo as postagens feitas e contendo a seguinte saída:

```
...
// Destrutor
Perfil::~Perfil(){
    cout << "Destrutor de perfil: " << nome << " - Quantidade de postagens feitas: "
         << quantidadeDePostagens << endl;
    // COMPLETE
    cout << "Perfil deletado" << endl;
}
```

O método `adicionarContato` deve adicionar um contato ao **Perfil**. 5. Quando um **Perfil** adiciona um outro **Perfil** como contato, este último também deve adicionar o primeiro **Perfil** como seu contato. Em outras palavras, quando X adiciona Y como contato, Y também deve adicionar X como contato. Caso não seja possível adicionar o contato (por falta de espaço no vetor), o **Perfil** já seja um contato ou se tente adicionar o próprio **Perfil** como contato dele mesmo, deve-se retornar *falso* – não adicionando o contato. Retorne *verdadeiro* caso contrário. Para fazer uma postagem existe o `adicionarPostagem`, que deve retornar *falso* caso não haja espaço disponível no vetor de postagens do **Perfil**. Caso haja espaço, ele deve retornar *verdadeiro*.

Os métodos `getPostagens` e `getQuantidadeDePostagens` permitem obter as **Postagens** feitas pelo **Perfil** (não inclui as **Postagens** de contato adicionado). O método `getPostagensDosContatos` retorna as postagens dos contatos do **Perfil**, feitas no dia atual e nos últimos três dias. Note que o parâmetro quantidade é um parâmetro de retorno, indicando a quantidade de elementos no vetor. Por exemplo, imagine que o Perfil tenha dois contatos, Ana e João. Ana possui as mensagens "M1" com data 1, "M3" com data 3 e "M5" com data 5. João possui as mensagens "M2" com data 2, "M4" com data 4 e "M6 com data 6. Ao chamar **getPostagensDosContatos** com data 5, deve-se retornar as postagens das datas 5, 4, 3 e 2, ou seja, "M5", "M4", "M3" e "M2" – e, portanto, quantidade valendo 4. Note que contatos podem ter mensagens na mesma data. Não se preocupem com a data de fim da **Story**.

Os métodos `getContatos` e `getQuantidadeDeContatos` permitem obter os contatos do **Perfil**. O método `getContatos` retorna o vetor com os **Perfis**; o método `getQuantidadeDeContatos` retorna a quantidade de **Perfis** nesse vetor.

Implemente o método `imprimir` como feito na Aula 7 e apresentado a seguir. E veja o exercício da aula 4 para entender e resolver o problema da dependência circular com a classe **Postagem**.

```
void Perfil::imprimir() {
  cout << endl << "Nome: " << nome << endl;
  cout << "Numero de postagens feitas: " << quantidadeDePostagens << endl;
  for (int i = 0; i < quantidadeDePostagens; i++)
    cout << "Postagens na data: " << postagens[i]->getData()
         << " - Texto: " << postagens[i]->getTexto() << endl;

  if (quantidadeDeContatos == 0)
    cout << "Sem contatos " << endl;
  else {
    for (int i = 0; i < quantidadeDeContatos; i++) {
      for (int j = 0; j < contatos[i]->getQuantidadeDePostagens(); j++)
        cout << "Postagens na data " << contatos[i]->getPostagens()[j]->getData()
             << " do contato  "<< contatos[i]->getNome()
             << " - Texto: " << contatos[i]->getPostagens()[j]->getTexto() << endl;
    }
  }
}
```

1.2 Classe PessoaNaoVerificada

Uma **PessoaNaoVerificada** é um tipo de **Perfil** e não tem nenhum atributo adicional e nenhum comportamento específico diferente da superclasse.

```
PessoaNaoVerificada(string nome, int maximoDeContatos, int maximoDePostagens);
virtual ~PessoaNaoVerificada();
```

O construtor recebe apenas o nome a ser atribuído e os máximos de contatos e postagens, enquanto o destrutor se comporta da mesma forma que a superclasse. O método imprimir deve ser o método herdado pelo perfil.

1.3 Classe PessoaVerificada

Uma **PessoaVerificada** é um tipo de **Perfil** que possui adicionalmente a informação do seu email. Para isso, a classe deve ser subclasse de **Perfil**. 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, int maximoDeContatos, int maximoDePostagens);
PessoaVerificada(string nome, int maximoDeContatos, int maximoDePostagens);
virtual ~PessoaVerificada();

string getEmail();
```

Existem dois construtores. Um deve receber o nome e o e-mail e os máximos de contatos e postagens, enquanto o outro recebe apenas um nome e os máximos (o email atribuído no construtor deve ser *email.padrao@usp.br*). O método `getEmail` deve retornar o endereço de email. O destrutor se comporta da mesma forma que a superclasse.

O método `imprimir` deve, utilizando refinamento, imprimir o email da **PessoaVerificada** (pulando uma linha depois) e então invoca o método da superclasse.

1.4 Classe Pagina

Uma **Pagina** é um tipo de **Perfil** com alguns comportamentos específicos. Portanto, essa classe deve ser subclasse de **Perfil**. 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, int maximoDeContatos,
  int maximoDePostagens);
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. Além disso, o método `getProprietario` deve apenas retornar o proprietário definido pelo construtor. O destrutor se comporta da mesma forma que a superclasse (especificado na aula 5).

O método `imprimir` deve, utilizando substituição, imprimir apenas o nome da página e o nome do proprietário responsável pela página:

```
cout << "Nome: " << getNome() << " - Proprietario: "
     << proprietario->getNome() << endl;
```

1.5 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();
virtual 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.

O método `imprimir` deve seguir o especificado na Aula 7, ou seja, 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**.

1.6 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 (especificado na aula 5).

Redefina o método `imprimir`. Ele deve imprimir em tela o novo atributo `dataDeFim` no seguinte formato:

```
Texto: <texto> - Data: <data> - Data De Fim: <dataDeFim> - 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.

1.7 Classe RedeSocial

A **RedeSocial** é a classe responsável por ter a lista de **Perfis** existentes na rede. Ela deve possuir os seguintes métodos **públicos**:

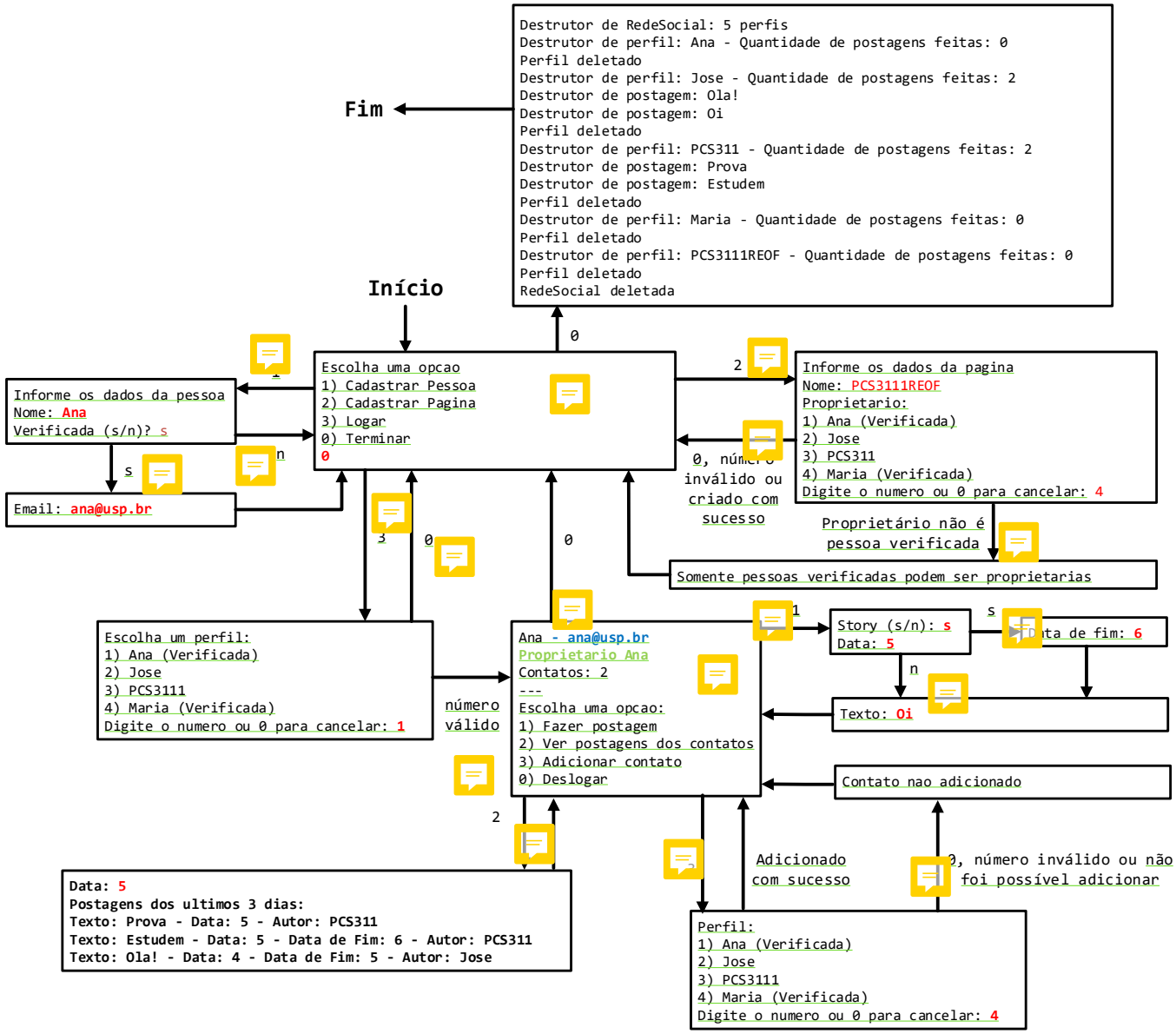
```
RedeSocial();
virtual ~RedeSocial();

virtual bool adicionar(Perfil* perfil);
virtual Perfil** getPerfis ();
virtual int getQuantidadeDePerfis();

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

O construtor da rede social não deve receber nenhum parâmetro. Use uma constante (`const`) para definir o tamanho máximo da rede, que deve ser de 100 perfis. Um **Perfil** deve ser adicionado na rede através do método `adicionar`. Caso a rede já possua o número máximo de **Perfis**, esse método deve retornar falso. Caso contrário, ele deve retornar verdadeiro e adicionar o **Perfil**. O destrutor deve ser implementado como especificado na Aula 5.

Os **Perfis** adicionados com sucesso à rede devem ser obtidos pelo método `getPerfis`, que retorna um vetor de **Perfis**. A quantidade de **Perfis** nesse vetor deve ser obtida pelo método `getQuantidadeDePerfis`.

Por fim, o método `imprimir` da classe **RedeSocial** está no código fornecido da Aula 5. Siga o especificado na Aula 7 para o método `imprimirEstatisticas`.

2. Interface com o usuário



Coloque o main em um arquivo em separado, chamado `main.cpp`. Ela é apresentada esquematicamente no diagrama acima. Cada retângulo representa uma "tela", ou seja, o conjunto de informações apresentadas e solicitadas. 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:

- Crie os perfis colocando sempre o valor 20 como máximo de contatos e de postagens
- 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 **PessoasNaoVerificadas** quando da seleção do proprietário.
- Ao sair do programa, chame o destrutor de **RedeSocial**.

3. Entrega

O projeto deverá ser entregue até dia **29/05** em um Judge específico, disponível em <https://laboo.pcs.usp.br/ep/> (nos próximos dias vocês receberão um login e uma senha). Você deverá fazer duas submissões (Entrega 1 e Entrega 2). **As entregas deverão possuir o mesmo conteúdo** (ou seja, é necessário submeter repetidas vezes por uma limitação do Judge).

> **Atenção**: não copie código de um outra pessoa. Qualquer tipo de cópia será considerada plágio e ambos os alunos terão **nota `0` no EP**. Portanto, **não envie** o seu código para um outro colega! 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. Cada entrega deve ser feita em um arquivo comprimido. Os fontes 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 dada **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.

4. Dicas

- Separe o main em várias funções para reaproveitar código. Planeje isso!
- O Code::Blocks tem um commando bastante útil para implementar os arquivos cpp. Ao clicar com o botão direito, vá em "insert/refactor"->"All class methods without implementation..." para que ele crie um esqueleto de todos os métodos da classe.
- 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 (por causa de referências circulares).
    - Inclua todas as dependências necessárias. Não dependa de `#includes` feitos por outros arquivos incluídos.

- Note que algumas telas do main selecionam perfis da mesma maneira (por exemplo, para escolher o responsável por uma disciplina ou escolher o perfil para seguir). Crie uma função auxiliar no main para reaproveitar isso!
- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- É muito trabalhoso testar o programa ao executar o main *com menus*, já que é necessário informar vários dados para inicializar a rede. Para testar, crie um main mais simples, 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.**