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 2020

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 Twitter. A rede é composta por diversos perfis que podem seguir outros perfis. Cada perfil faz publicações que são recebidas pelos seus seguidores.

1 Introdução

Deseja-se criar uma *Rede Social s*imples, no estilo do Twitter. 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, professores e disciplinas (note que não há **alunos**). Cada perfil poderá ter vários seguidores e poderá fazer publicações (texto ou eventos) para seus seguidores. Por simplicidade, o número de seguidores e de publicações 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.).

2. Projeto

Deve-se implementar em C++ as classes **Perfil**, **Professor**, **Disciplina**, **Publicacao**, **Evento** 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 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.

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.

2.1 Classe Perfil

Um **Perfil** é o elemento básico da rede social. Ele possui um número USP, um nome e um e-mail. Além disso, ele pode adicionar seguidores, fazer publicações e receber publicações feitas pelas pessoas que o seguem.

Por simplicidade, neste EP o número de seguidores é limitado a `MAXIMO_SEGUIDORES`, um "define" que deve ser feito em `Perfil.h`. Da mesma forma, o número de publicações é limitado a `MAXIMO_PUBLICACOES`, também definida em `Perfil.h`. Não faça outros "defines" além desses.

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

```
#define MAXIMO_SEGUIDORES 20
#define MAXIMO_PUBLICACOES 20

Perfil(int numeroUSP, string nome, string email);
virtual ~Perfil();

int getNumeroUSP();
string getNome();
string getEmail();

virtual bool adicionarSeguidor(Perfil* seguidor);

virtual bool publicar(string texto);
virtual bool publicar(string texto, string data);

virtual bool receber(Publicacao* p);

virtual Publicacao** getPublicacoesFeitas();
virtual int getQuantidadeDePublicacoesFeitas();

virtual Publicacao** getPublicacoesRecebidas();
virtual int getQuantidadeDePublicacoesRecebidas();

virtual Perfil** getSeguidores();
virtual int getQuantidadeDeSeguidores();

virtual void imprimir();
```

Os métodos `getNumeroUSP`, `getNome` e `getEmail` devem retornar os valores do número USP, nome e email informados no construtor.

O método `adicionarSeguidor` deve adicionar um seguidor ao **Perfil**. Caso não seja possível adicionar o seguidor (por falta de espaço no vetor), o **Perfil** já seja um seguidor ou se tente adicionar o próprio **Perfil** como seguidor dele mesmo, deve-se retornar *falso* – não adicionando o seguidor. Retorne *verdadeiro* caso contrário. Quando um **Perfil** tem um seguidor adicionado, ele deve ter uma **Publicação** adicionada à sua lista de publicações recebidas com o seguinte texto:

```
Novo seguidor: <nome do Seguidor>
```

Note que essa publicação **não deve** ser enviada aos seguidores do **Perfil**. Ela deve apenas aparecer como uma **Publicação** na lista de publicações recebidas. Esta **Publicação** deve ter o próprio **Perfil** como autor. Por exemplo, se o **Perfil** João é adicionado como seguidor do **Perfil** da Maria, o **Perfil** da Maria deve adicionar o João como seu seguidor e a **Publicação** "Novo seguidor: Joao", cujo *autor* é a Maria,

deve ser adicionada à lista de **Publicações** de Maria. Porém, José, que já seguia a Maria, não receberá essa **Publicação**.

Para fazer uma publicação existem dois métodos com o mesmo nome: `publicar`. O que recebe apenas um texto deve fazer uma **Publicação** *normal*; o que recebe um texto e uma data deve publicar um **Evento**. Ambos os métodos têm comportamento similar: eles devem retornar *falso* caso não haja espaço disponível no vetor de publicações do **Perfil**. Caso haja espaço, ele deve retornar *verdadeiro*. Havendo espaço, a publicação, além de adicionada à lista de publicações feitas, deve ser enviada a todos os seguidores. O método `receber` é o método que recebe essa **Publicação**. Ao receber uma **Publicação**, o **Perfil** deve apenas adiciona-la à sua lista de publicações recebidas e retornar *true*. Caso não haja espaço no vetor de **Publicações**, o método receber deve retornar *falso* e ignorar a **Publicação**.

Os métodos `getPublicacoesFeitas` e `getQuantidadeDePublicacoesFeitas` permitem obter as **Publicações** feitas pelo **Perfil** (não inclui as **Publicações** de seguidor adicionado). De forma similar, os métodos `getPublicacoesRecebidas` e `getQuantidadeDePublicacoesRecebidas` permitem obter as **Publicações** recebidas pelo **Perfil**.

Os métodos `getSeguidores` e `getQuantidadeDeSeguidores` permitem obter os seguidores do **Perfil**. O método `getSeguidores` retorna o vetor com os **Perfis**; o método `getQuantidadeDeSeguidores` retorna a quantidade de **Perfis** nesse vetor.

Não é especificado no enunciado o funcionamento do método `imprimir`. Implemente-o como desejado. E veja o exercício da aula 4 para entender e resolver o problema da dependência circular com a classe **Publicação**.

2.2 Classe Professor

Um **Professor** é um tipo de **Perfil** que possui adicionalmente a informação do seu departamento. 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).

```
Professor(int numeroUSP, string nome, string email, string departamento);
virtual ~Professor();

string getDepartamento();
```

O construtor deve receber o número USP, nome, e-mail e o departamento do professor. O método `getDepartamento` deve retornar o departamento.

2.3 Classe Disciplina

Uma **Disciplina** 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).

```
Disciplina(string sigla, string nome, Professor* responsavel);
virtual ~Disciplina();

Professor* getResponsavel();
string getSigla();
```

Uma disciplina possui uma sigla, um nome e um responsável, os quais devem ser informados no construtor. Apesar de ser um **Perfil**, disciplinas não precisam de um número USP e tampouco e-mail. Portanto, ao criar uma **Disciplina** atribua o valor `0` ao número USP e vazio ("") como e-mail. Ao criar a **Disciplina**, o **Professor** responsável deve ser automaticamente adicionado como seguidor.

Os métodos `getResponsável` e `getSigla` devem apenas retornar os valores definidos pelo construtor, o responsável e a sigla, respectivamente.

Diferentemente de outros **Perfis**, a lista de publicações de uma **Disciplina** deve ser limitada a apenas **Publicações** feitas pela própria **Disciplina**. Ou seja, quando um seguidor é adicionado à **Disciplina** não se deve adicionar a mensagem de aviso. Além disso, como não faz sentido uma disciplina seguir outros **Perfis**, ela não deve fazer nada ao receber uma **Publicação**.

2.4 Classe Publicação

Uma **Publicação** é um mensagem publicada na rede social e que é divulgada aos seguidores do seu ator. Toda publicação possui um texto, um autor e pode ser curtida por outros **Perfis** diferentes do autor. A classe **Publicação** deve possuir os seguintes métodos **públicos**:

```
Publicacao(Perfil* autor, string texto);
virtual ~Publicacao();

Perfil* getAutor();
string getTexto();

virtual void curtir(Perfil* quemCurtiu);
virtual int getCurtidas();

virtual void imprimir();
```

O construtor deve receber o **Perfil** do autor e um texto, os quais são informados pelos métodos de acesso `getAutor` e `getTexto`.

Uma **Publicação** pode ser curtida por outros **Perfis** (que não o autor). Para isso, o método `curtir` deve receber o **Perfil** que curtiu. Se quem curtir for o autor, a curtida deve ser ignorada. Por simplicidade, um mesmo **Perfil** pode curtir várias vezes a mesma **Publicação**. A quantidade de curtidas obtidas pela **Publicação** deve ser obtida pelo método `getCurtidas`.

Assim como no **Perfil**, a implementação do método `imprimir` não está especificada e pode ser implementada como desejado.

2.5 Classe Evento

O **Evento** é um tipo de **Publicação** que possui uma data. Para isso, a classe deve ser subclasse de **Publicação**. 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).

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

string getData();
```

O construtor deve receber o autor, o texto e a data do **Evento** (o formato da data não é relevante, já que ela será armazenada como uma string). O método `getData` deve retornar a data definida no construtor.

2.6 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(int numeroMaximoDePerfis);
virtual ~RedeSocial();

int getQuantidadeDePerfis();
Perfil** getPerfis();

bool adicionar(Perfil* p);
virtual void imprimir();
```

O construtor da rede social deve receber o número máximo de perfis que a rede pode possuir. Ou seja, **não use** um `#define` ou uma constante para isso: o tamanho máximo da rede deve ser definido durante a sua criação. Um **Perfil** deve ser adicionador 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**.

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`.

O método `imprimir` da classe **RedeSocial** também não é especificado e pode ser implementado conforme desejado.

3 Texto com espaço

O `cin` usa o espaço ou o fim de linha para identificar o fim do texto digitado. Com isso não é possível obter pelo `cin` uma string cujo texto possui espaços. Para que você possa obter nomes e textos com espaço, é necessário usar a função `getline(entrada, variável)` (é preciso fazer `#include <string>` para usa-la). Essa função coloca em `variável` o valor digitado até o '\n' de `entrada` (no nosso caso a entrada é o `cin`). Um cuidado ao usar essa função é que se for usado um `cin` anteriormente, o texto digitado e não aproveitado pode ser capturado pelo `getline`. Por isso é recomendável chamar o `cin.ignore(quantidade, '\n')`, antes de chamar o `getline`. Esse método permite ignorar uma `quantidade` de caracteres até chegar a um caracter '\n'.

Portanto, para pegar uma string com espaço em C++ é necessário fazer:

```
string nome;
cin.ignore(100, '\n'); // Ignorando até 100 caracteres que sobraram de cin anterior
getline(cin, nome); // guardando uma linha em nome
```

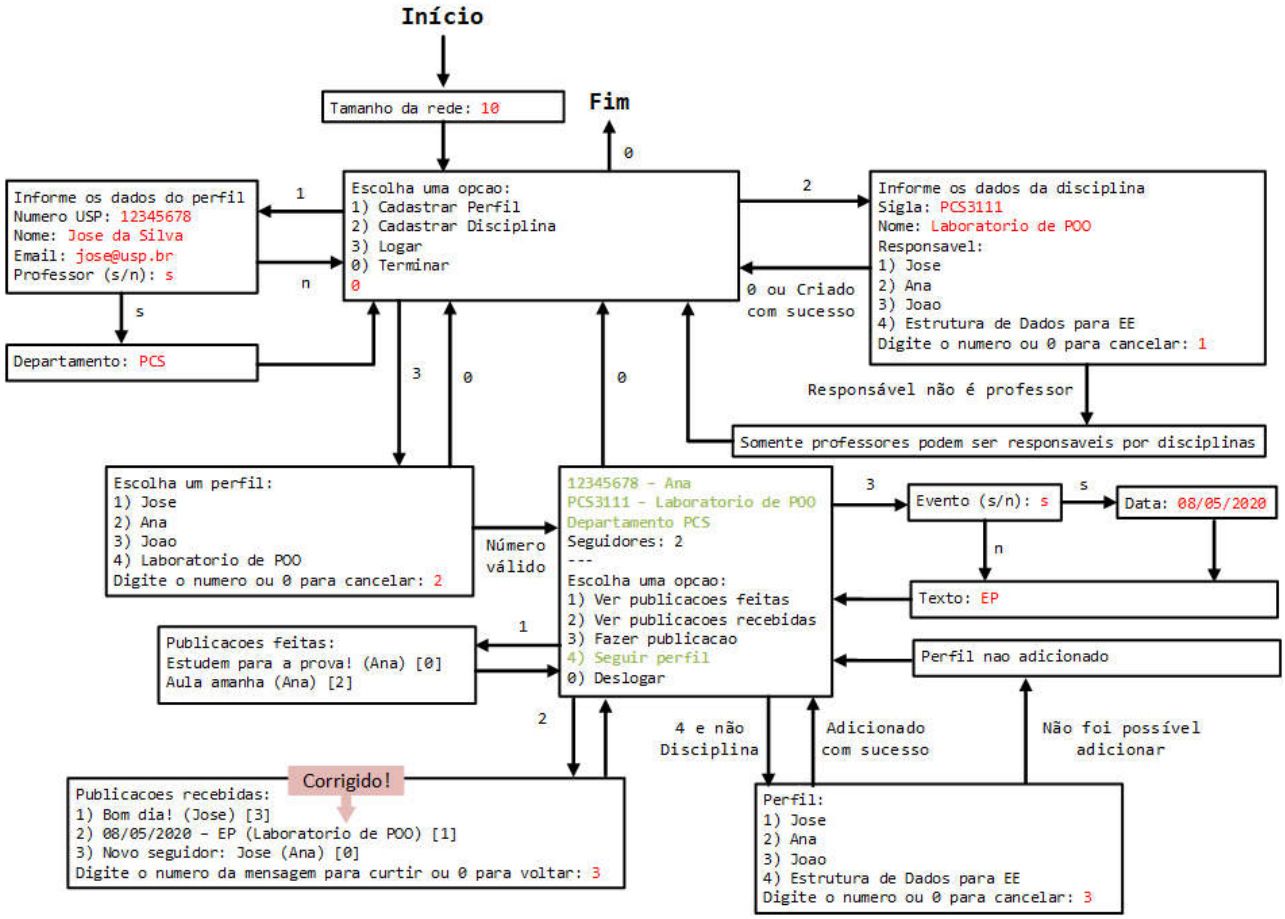
Em uma aula futura veremos com mais detalhes como funciona o `cin` e o `cout` do C++.

4 Interface com o usuário

Coloque o 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 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 mensagens que acontecem dependendo do tipo do perfil:

- Quando for um **Professor** ou um objeto **Perfil**, deve ser apresentado o texto "`<número USP>` - `<nome>`" como, no exemplo, "123456789 - Ana". Quando for uma **Disciplina**, o texto a ser apresentado é "`<sigla>` - `<nome>`" como, no exemplo, "PCS3111 – Laboratorio de POO".
- A informação do departamento só deve aparecer se o perfil for de um **Professor** (nos outros casos não deve existir essa linha).
- A opção "4) Seguir perfil" não deve aparecer se o perfil logado for de uma **Disciplina**.

> **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:

- O tamanho da rede informado no início do programa deve definir o numeroMaximoDePerfis da **RedeSocial**.

- Um evento deve ser impresso como "<Data> - <Texto> (<Autor>) [<Curtidas>]", como mostrado na publicação recebida 2 da imagem. A publicação deve ser impressa como "<Texto> (<Autor>) [<Curtidas>]".
- No diagrama não são apresentados casos de erro (por exemplo, a digitação de um número de perfil inválido). 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 "Seguir perfil" e perfis normais e disciplinas quando da seleção do responsável.

5 Entrega

O projeto deverá ser entregue até dia **08/05** em um Judge específico, disponível em <http://judge.pcs.usp.br/pcs3111/ep/> (nos próximos dias vocês receberão um login e uma senha). Você deverá fazer três submissões (Entrega 1, Entrega 2 e Entrega 3). **Todas as entregas deverão possuir o mesmo conteúdo** (é submeter repetidas vezes por uma limitação do Judge).

> **Atenção**: não copie código de um outro colega. 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.

6 Dicas

- Separe o `main` em várias funções para reaproveitar código. Planeje isso!
- 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!
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**