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 – 2º SEMESTRE DE 2024

1 Introdução

Uma startup de ex-alunos de Engenharia Elétrica da Poli está desenvolvendo uma catraca eletrônica para ser usada em prédios de escritório. Além do controle de acesso, a catraca permite aos gestores cuidarem do controle de ponto dos funcionários, o que é importante do ponto de vista da legislação trabalhista.

Além das funcionalidades criadas para o primeiro EP, neste segundo EP o software deve ser melhorado para permitir seu uso na USP – para funcionários, alunos e visitantes –, e permitir que as informações sejam carregadas e salvas.

1.1 Objetivo

O objetivo deste projeto é fazer o software para o controle de catracas eletrônicas, evoluindo o que foi criado no EP1. 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.).

Para desenvolver o EP deve-se manter a mesma dupla do EP1. Será possível apenas **desfazer a dupla**, mas não formar uma nova.

Atenção:

- Deve ser mantida a mesma dupla do EP1. É possível apenas *desfazer* a dupla. Com isso, cada aluno deve fazer uma entrega diferente (e em separado). Caso você deseje fazer isso, envie um e-mail para levy.siqueira@usp.br até dia **11/11** informando os números USP dos alunos e **mandando-o com cópia para a sua dupla**.
- Não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e os grupos envolvidos terão nota 0 no EP. Portanto, não envie o seu código para um colega de outro grupo!

2 Projeto

Deve-se implementar em C++ as classes `Data`, `Registro`, `Entrada`, `Saida`, `Usuario`, `Funcionario`, `Aluno`, `Visitante`~~s~~, `GerenciadorDeUsuario`, `Catraca` e `PersistenciaDeUsuario`, além de criar uma interface com o usuário que permita o funcionamento do programa como desejado. Um diagrama de classes apresentando essas classes e as relações entre elas é apresentado na Figura 1.

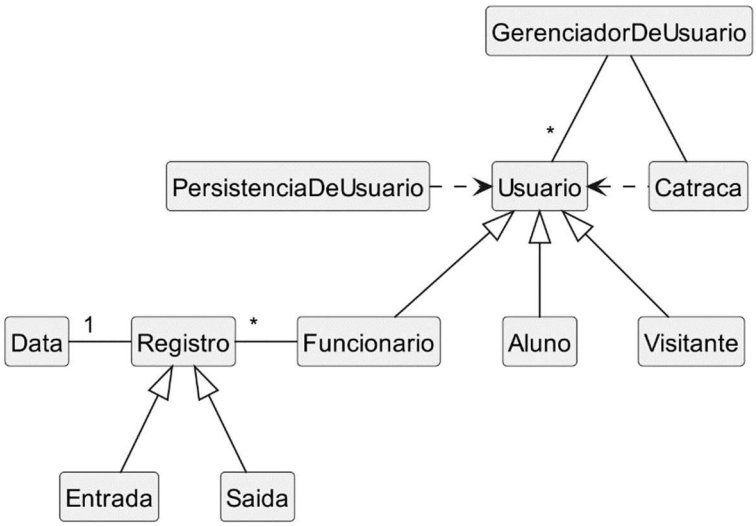


Figura 1: Diagrama de classes com as classes do EP2.

Atenção:

1. 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. **Note que você poderá definir atributos e métodos privados** e **protegidos, caso necessário**.
2. Não é permitida a criação de outras classes além dessas.
3. As classes filhas podem (*e às vezes devem*) redefinir métodos da classe mãe.
4. Não faça `#define` para constantes. Você pode (e deve) fazer `#ifndef/#define` para permitir a inclusão adequada de arquivos.

O não atendimento a esses pontos pode resultar em **erro de compilação** na correção automática 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 "Data.cpp" e "Data.h". Note que você deve criar os arquivos necessários.

Em relação às exceções (assunto da Aula 9), todas as especificadas são da biblioteca padrão (não se esqueça de fazer `#include <stdexcept>`). O texto usado como motivo da exceção não é especificado no enunciado e não será avaliado. Jogue as exceções criando um objeto usando new. Por exemplo, para jogar um `logic_error` faça algo como:

```
throw new logic_error("Mensagem de erro");
```

Caso a exceção seja jogada de outra forma, pode haver erros na correção e, consequentemente, desconto na nota.

2.1 Classe Data

A `Data` representa um instante de tempo, com dia, mês, ano, hora, minuto e segundo. A única alteração nela é no construtor, que agora deve fazer algumas verificações. Essa classe deve possuir apenas os seguintes **métodos públicos**:

```
Data(int hora, int minuto, int segundo, int dia, int mes, int ano);
virtual ~Data();

int getHora();
int getMinuto();
int getSegundo();
int getDia();
int getMes();
int getAno();

int diferenca(Data* d);
```

O construtor deve receber a hora, minuto, segundo, dia, mês e ano da data. A hora deve ser entre 0 e 23, inclusive, os minutos entre 0 e 59, inclusive, o segundo entre 0 e 59, inclusive, o dia entre 1 e 31, inclusive, e o mês entre 1 e 12, inclusive. Caso seja passado algum valor fora das faixas esperadas, jogue um `logic_error`. Essas informações são retornadas pelos *getters*. Se você criar algum objeto, destrua-o no destrutor.

Assim como no EP1, o método `diferenca` recebe uma outra `Data` e retorna a diferença em segundos entre a data atual e a data recebida. Por exemplo, considere um objeto que guarda a data 10:00:00 de 21/10/2024. A chamada de `diferenca` nesse objeto passando a data 9:00:00 de 21/10/2024 deve retornar 3600. Caso a chamada de `diferenca` nesse mesmo objeto receba 10:00:00 de 22/10/2024 (mesmo horário no dia seguinte), o retorno deve ser -86400. Consulte o enunciado do EP1 sobre os detalhes de implementação.

2.1.1 Classe Registro

O `Registro` é um evento, manual ou não, de um funcionário em uma catraca. As classes filhas `Entrada` e `Saida` representam agora se o evento é de entrada ou de saída (elas servem apenas para identificar o tipo de evento). Essa classe deve ser **abstrata** e deve possuir apenas os seguintes **métodos públicos**:

```
Registro(Data* d);
Registro(Data* d, bool manual);
virtual ~Registro() = 0;

Data* getData();
bool isManual();
```

A classe possui dois construtores. Um recebe uma `Data` e se a entrada foi manual (`true`) ou não (`false`). O outro recebe apenas uma `Data` e considera que a entrada não foi manual (`false`). Em qualquer um dos construtores deve-se jogar um `invalid_argument` caso a data informada seja nula (nullptr). No destrutor destrua a data.

Os métodos `getData` e `isManual` simplesmente retornam as respectivas informações passadas no construtor.

Dica: como no C++ é obrigatório ter pelo menos um método abstrato para que a classe seja abstrata, um recurso útil é fazer com que o destrutor seja abstrato. Com isso, todos os demais métodos podem ser concretos. Porém, apesar de o destrutor ser definido como abstrato, ele precisa ser implementado no .cpp.

2.2 Classe Entrada

A `Entrada` representa um registro de entrada de um funcionário. Ela deve ser filha de `Registro` e não tem métodos adicionais. Essa classe deve possuir apenas os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Entrada(Data* d);
Entrada(Data* d, bool manual);
virtual ~Entrada();
```

Os construtores devem possuir o mesmo comportamento da classe mãe.

2.3 Classe Saida

A `Saida` representa um registro de saída de um funcionário. Assim como a `Entrada`, ela deve ser filha de `Registro` e não tem métodos adicionais. Essa classe deve possuir apenas os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Saida(Data* d);
Saida(Data* d, bool manual);
virtual ~Saida();
```

Os construtores devem possuir o mesmo comportamento da classe mãe.

2.4 Classe Usuario

O `Usuario` representa alguém que terá acesso ao edifício pelo sistema. Como agora existem três tipos de usuário, com comportamentos diferentes, essa classe deve ser **abstrata**. Escolha apropriadamente os métodos (um ou mais) abstratos. A parte de registro ficou apenas na classe relativa ao funcionário. Com isso, essa classe deve possuir os seguintes métodos públicos:

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

string getNome();
int getId();

bool entrar(Data *d);
bool sair(Data *d);
bool registrarEntradaManual(Data *d);
bool registrarSaidaManual(Data* d);
```

O construtor deve receber o identificador do usuário (usado pela `Catraca`) e o nome. Os métodos `getNome` e `getId` devem retornar as informações passadas ao construtor.

Assim como no EP1, o método `entrar` deve receber uma `Data` e verificar se o usuário pode entrar no edifício e o método `sair` recebe uma `Data` e verifica se o usuário pode sair do edifício. Porém, cada tipo de usuário possui um comportamento específico. O mesmo acontece com os métodos `registrarEntradaManual` e `registrarSaidaManual`, os quais registram entradas e saídas manuais (ou seja, que serão registradas manualmente pela segurança do prédio). Vejam nas classes `Funcionario`, `Aluno` e `Visitante` o comportamento desses métodos.

2.5 Classe Funcionario

O `Funcionario` representa um `Usuario` que terá seus registros armazenados. Portanto, ela deve ser filha de `Usuario`. Essa classe deve possuir apenas os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Funcionario(int id, string nome);
Funcionario(int id, string nome, vector<Registro*>* registros);
virtual ~Funcionario();

int getHorasTrabalhadas(int mes, int ano);
vector<Registro*>* getRegistros();
```

O construtor deve receber o número identificador (`id`) e nome do funcionário. No construtor com dois parâmetros, crie também um `vector` que armazenará os `Registros` do funcionário. No outro construtor é passado um `vector`, potencialmente com alguns valores, o qual deve ser o usado internamente pelo `Funcionario`. Isso será útil para a persistência. No destrutor destrua o `vector`, independente se ele foi criado na classe ou recebido. Destrua também cada um dos `Registros`.

Em relação à entrada e saída, os métodos seguem o comportamento do `Usuario` do EP1. O método `entrar` deve receber uma `Data` e verificar se o funcionário pode entrar no edifício. Não é permitida a entrada caso o último registro dele foi uma entrada ou se a data informada for anterior à data do último registro. O método `sair` tem lógica similar: não deve permitir a saída caso o último registro foi uma saída ou se a data for anterior à data do último registro. Note que o primeiro registro pode ser de entrada ou de saída. Os registros criados por esse método não devem ser manuais, dado que eles serão feitos pela `Catraca`. Note que não há mais a restrição de falta de espaço, dado que é usado um `vector`.

Os métodos `registrarEntradaManual` e `registrarSaidaManual` tem o mesmo comportamento de `entrar` e `sair`, com a diferença que eles registram entradas e saídas manuais (ou seja, que serão registradas manualmente pela segurança do prédio). Esses métodos devem seguir as mesmas regras explicadas anteriormente: não permitir a entrada caso o último registro foi uma entrada (se `registrarEntradaManual`) ou saída (se `registrarSaidaManual`), ou se a data for anterior à data do último registro.

O método `getRegistros` deve retornar o `vector` de registros desse `Funcionario`. Não é mais necessário um método para obter a quantidade, já que essa informação é acessível pelo método `size()` do `vector`. O método deve retornar um `vector` vazio caso ainda não tenha se adicionado registros.

O método `getHorasTrabalhadas` deve fazer o cálculo das horas trabalhadas pelo usuário naquele mês e ano. Ou seja, ele deve somar a diferença entre as saídas e entradas naquele mês. O valor deve ser truncado, ou seja, caso a soma dê 10,5, o método deve retornar 10. Ou seja, primeiro some as diferenças e depois faça o cálculo necessário para que o resultado seja em horas. O filtro considera apenas a data de entrada, ou seja, se a chamada do método foi com 5 e 2024 (05/2024) e a entrada foi no mês 05/2024 e a saída foi no mês 06/2024, essas horas devem ser computadas; porém se a entrada foi no mês 04/2024 e a saída foi em 05/2024, essas horas não devem ser consideradas para a chamada do método com 5 e 2024. Além disso, se a entrada no mês solicitado for o último registro do usuário, ela não deve ser considerada no cálculo de horas trabalhadas (já que não há ainda uma saída registrada). Por exemplo, considere um usuário com os seguintes registros:

Entrada	Saída
09:04:30 de 30/09/2024	15:00:20 de 30/09/2024
23:00:00 de 30/09/2024	08:06:00 de 01/10/2024
09:05:30 de 02/10/2024	12:08:00 de 02/10/2024
13:10:00 de 02/10/2024	18:10:30 de 02/10/2024
09:10:30 de 03/10/2024	

A chamada `getHorasTrabalhadas(9, 2024)` deve retornar 15, enquanto que `getHorasTrabalhadas(10, 2024)` deve retornar 8.

2.6 Classe Aluno

O `Aluno` representa um `Usuario` que não tem seus acessos registrados e que tem apenas restrições de horários de entrada e de saída. Portanto, ela deve ser filha de `Usuario`. Essa classe deve possuir apenas os seguintes membros públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
static const int HORARIO_INICIO = 6;
Aluno(int id, string nome);
virtual ~Aluno();

static void setHorarioFim(int hora, int minuto);
static int getHoraFim();
static int getMinutoFim();
```

O construtor recebe o identificador do aluno e seu nome. O destrutor não tem comportamento definido.

Os alunos possuem uma janela de horário em que eles podem entrar no prédio. O início da janela é definido pela constante `HORARIO_INICIO` e o fim pelos métodos estáticos `setHorarioFim`, `getHoraFim` e `getMinutoFim`. Ou seja, o horário de início é fixo, enquanto que o horário de fim pode ser alterado e será usado para todos os alunos. O método `setHorarioFim` deve receber a hora e o minuto do fim da janela. Caso o horário seja menor que o horário de início ou maior que 23, deve-se jogar um `logic_error`. Também se deve jogar um `logic_error` caso o minuto seja menor que 0 ou maior que 59. Os métodos estáticos `getHoraFim` e `getMinutoFim` retornam, respectivamente, a hora e o minuto de fim da janela. Considere que inicialmente a hora de fim é 23 e o minuto de fim é 0.

Considerando a janela, o método `entrar` deve permitir que um `Aluno` entre no prédio apenas se o horário esteja dentro (inclusive as fronteiras) da janela. Por exemplo, para um horário de fim 22:59, a entrada do aluno será permitida das 6:00 até 22:59 (inclusive). Fora dessa janela o método deve retorna `false`. A saída do `Aluno` é sempre permitida, ou seja, o método `sair` deve sempre retornar `true`. Por simplicidade, o método `registrarEntradaManual` deve sempre retornar `false` e o método `registrarSaidaManual` deve sempre retornar `true`. Por simplicidade, permita que entradas sejam depois de outras entradas e saídas sejam depois de outras saídas.

2.7 Classe Visitante

O `Visitante` representa um `Usuario` que só pode acessar o prédio durante um período específico. Portanto, ela deve ser filha de `Usuario`. Essa classe deve possuir apenas os seguintes membros públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Visitante(int id, string nome, Data* inicio, Data* fim);
virtual ~Visitante();

Data* getDataInicio();
Data* getDataFim();
```

O construtor deve receber um id, um nome e as datas de início e fim de acesso ao prédio. Caso a data de início ou de fim seja nula, ou o fim seja menor que o início, o construtor deve jogar um `logic_error`. No destrutor destrua as datas de início e fim.

As datas de início e fim informadas no construtor são retornadas pelos métodos `getDataInicio` e `getDataFim`.

A entrada e saída do prédio só deve ser permitida dentro da janela delimitada pelo início e fim informado no construtor. Portanto, os métodos `entrar` e `sair` devem retornar `true` apenas se a data de entrada estiver entre o início e fim informados (intervalo aberto). Por exemplo, para um `Visitante` com início em 1/10/2024 às 9:00:00 e fim em 03/10/2024 às 19:00:00 não terá a entrada permitida no dia 04/10/2024 às 08:00:00. Por simplicidade, os métodos `registrarEntradaManual` e `registrarSaidaManual` devem sempre retornar `false`. Por simplicidade, permita que entradas sejam depois de outras entradas e saídas sejam depois de outras saídas.

2.8 Classe GerenciadorDeUsuario

A classe `GerenciadorDeUsuario` gerencia os usuários conhecidos pelas catracas. A diferença para o EP1 é que ela usa um `vector` e o método `adicionar` não retorna mais um booleano. Essa classe deve possuir os seguintes métodos públicos:

```
GerenciadorDeUsuario();
GerenciadorDeUsuario(vector<Usuario*>* usuarios);
virtual ~GerenciadorDeUsuario();

void adicionar(Usuario* u);
Usuario* getUsuario(int id);
vector<Usuario*>* getUsuarios();
```

Existem dois construtores. O que não recebe parâmetros deve criar um `vector` de `Usuario`. O que recebe um `vector` deve usá-lo internamente para armazenar os usuários. Isso será útil para a persistência. O destrutor deve destruir os usuários e o `vector`.

O método `adicionar` deve adicionar um `Usuario` ao gerenciador. Caso o usuário já tenha sido adicionado (considere que usuários são iguais caso tenham ids iguais), o método deve jogar uma exceção do tipo `invalid_argument`.

O método `getUsuario` deve retornar o objeto `Usuario` com o id informado que foi adicionado ao gerenciador. Caso não tenha sido adicionado um `Usuario` com esse id, o método deve retornar `nullptr`.

O método `getUsuarios` deve retornar o `vector` de `Usuario` desse gerenciador. Caso ainda não tenha sido adicionado usuários no gerenciador, retorne um `vector` vazio. Note que não é mais necessário um método separado para obter a quantidade de elementos do `vector`.

2.9 Classe Catraca

A `Catraca` representa um equipamento que permite a entrada e saída de usuários conhecidos. Ela tem o mesmo comportamento do EP1. Essa classe deve possuir os seguintes métodos públicos:

```
Catraca(GerenciadorDeUsuario* g);
virtual ~Catraca();

bool entrar(int id, Data* d);
bool sair(int id, Data* d);
```

O construtor deve receber um `GerenciadorDeUsuario`, o qual será consultado para encontrar o `Usuario` com o id informado na tentativa de entrada ou saída. Note que um sistema pode ter várias catracas e elas podem compartilhar o gerenciador. Por isso, o destrutor não deve destruir o `GerenciadorDeUsuario`.

O método `entrar` deve procurar no `GerenciadorDeUsuario` o `Usuario` com o id informado e, para o `Usuario` encontrado, chamar o método `entrar` (representando uma entrada automática) e retornar o valor que foi retornado por esse método. Caso o usuário não seja encontrado, o método deve retornar `false`. O método `sair` tem comportamento similar – a única diferença é que ele chama o método `sair` do `Usuario` ao invés do `entrar`.

2.10 Classe PersistenciaDeUsuario

A `PersistenciaDeUsuario` permite salvar e carregar os dados dos `Usuarios`, fazendo a persistência deles. Essa classe deve possuir os seguintes métodos públicos:

```
PersistenciaDeUsuario();
virtual ~PersistenciaDeUsuario();

vector<Usuario*>* carregar(string arquivo);
void salvar(string arquivo, vector<Usuario*>* v);
```

O construtor e o destrutor não possuem comportamento definido. O método `carregar` deve carregar do arquivo informado os usuários e retornar um `vector` deles. Caso o arquivo não exista ou não

esteja no formato esperado, o método deve jogar um `logic_error`. O método `salvar` deve fazer o contrário: receber um `vector` de usuários e salvá-los no arquivo informado. Caso haja algum problema ao salvar, jogue um `logic_error`.

Siga o formato especificado a seguir para salvar e carregar. Por simplicidade, será utilizado um caractere de nova linha ('\n') ou espaço (' ') como delimitador (pode ser qualquer um deles). Portanto, considere que os nomes não possuem espaço. Considere também que **há uma linha em branco no final do arquivo**.

O arquivo contém uma sequência de usuários (em qualquer ordem) no seguinte padrão:

- Caso o `Usuario` seja um `Funcionario`, o formato deve ser:

```
F <id> <nome>
<quantidade de registros>
<tipo 1> <data 1> <manual 1>
<tipo 2> <data 2> <manual 2>
...
<tipo n> <data n> <manual n>
```

  Onde:
  - `<id>` é o identificador do funcionário;
  - `<nome>` é o nome do funcionário;
  - `<quantidade de registros>` é a quantidade de registros (que podem ser de entrada ou saída);
  - Para cada registro são apresentadas as seguintes informações:
    - `<tipo>`: "`E`" se for entrada ou "`S`" se for saída (sem as aspas);
    - `<data>:` é a data, no formato descrito adiante;
    - `<manual>`: 1 se for registro manual e 0 se for registro normal.
- Caso o `Usuario` seja um `Aluno`, o formato deve ser:

```
A <id> <nome>
```

  Onde:
  - `<id>` é o identificador do aluno; e
  - `<nome>` é o nome do aluno.
- Caso o `Usuario` seja um `Visitante`, o formato deve ser:

```
V <id> <nome> <data inicio> <data fim>
```

  Onde:
  - `<id>` é o identificador do visitante; e
  - `<nome>` é o nome do visitante;
  - `<data inicio>` é a data de início, seguindo o formato descrito adiante; e
  - `<data fim>` é a data de fim, seguindo o formato descrito adiante.
- O formato de uma Data é o seguinte:

```
<hora> <minuto> <segundo> <dia> <mês> <ano>
```

A seguir é apresentado um exemplo de arquivo:

```
A 1234 Jose
F 1235 Maria
4
E 9 4 30 30 9 2024 0
S 15 0 22 30 9 2024 0
E 23 0 0 30 9 2024 0
S 8 6 0 1 10 2024 1
V 1240 Silvia    9 4 30 30 9 2024 9 4 30 1 10 2024
A 1236 Antonio
```

Nesse arquivo há 4 `Usuarios`. O primeiro é um `Aluno` de id 1234 e nome Jose; o segundo é um `Funcionario`, de id 1235, nome Maria e 4 registros (duas entradas normais e uma saída normal e uma manual); um `Visitante` de id 1240, nome Silvia e entrada permitida entre 30/09/2024 09:04:30 e 1/10/2024 09:04:30; e um `Aluno` de id 1236 e nome Antonio.

Junto com o enunciado é fornecido o arquivo com esse exemplo (`usuarios.txt`).

Dica: para facilitar o processamento do funcionário você pode carregar os registros usando um for de 0 até a quantidade.

3 Main e menu.cpp

Coloque a `main` em um arquivo separado, chamado `main.cpp`. Nele você deverá simplesmente chamar uma função `menu`, a qual ficará no arquivo `menu.cpp`. Não faça include de menu no arquivo com o main (*jamais faça include de arquivos .cpp*). Portanto, o `main.cpp` deve ser.

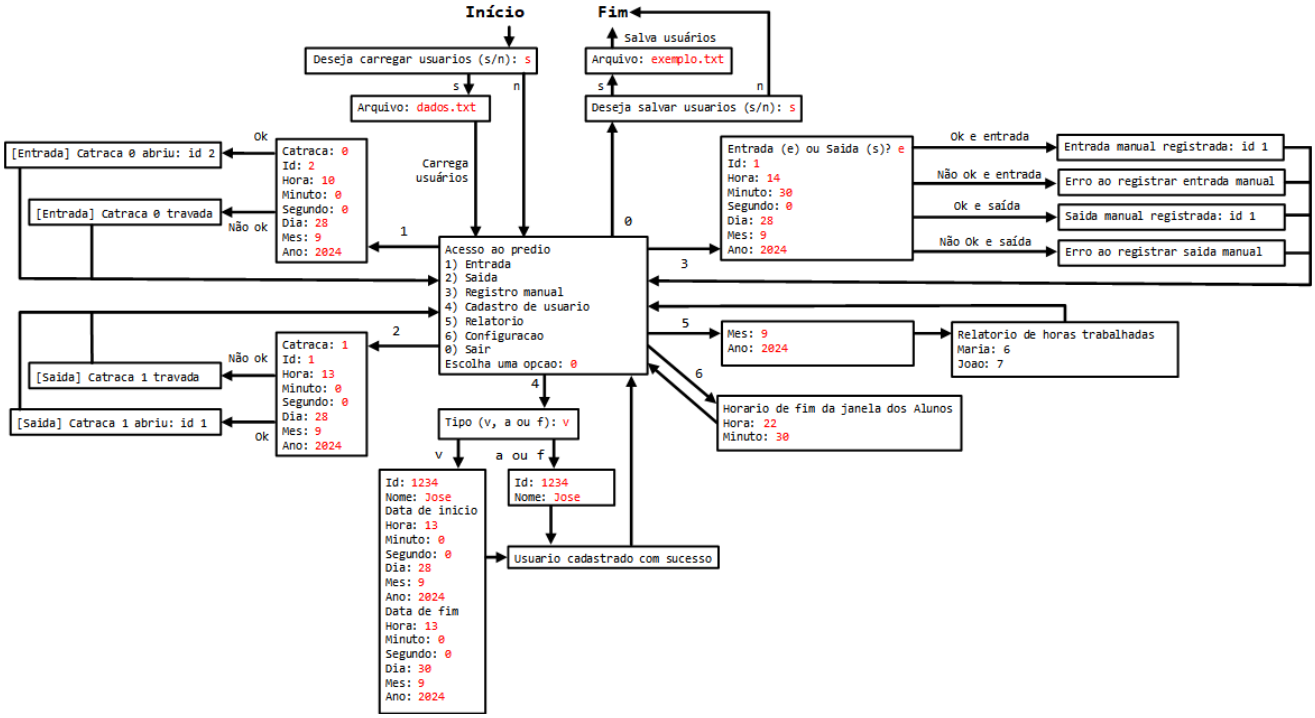
```cpp
void menu();

int main() {
    menu();
    return 0;
}
```

O menu foi alterado para permitir as novas funcionalidades do EP2, mas a ideia é a mesma. O usuário poderá carregar os usuários e, caso ele deseje, crie um `GerenciadorDeUsuario` com ele. Se não, crie um `GerenciadorDeUsuario` vazio. Crie dois objetos `Catraca` (catraca 0 e 1), os quais usam a mesmo `GerenciadorDeUsuario`. No final do programa, deve ser dada a opção de persistir os `Usuarios`.

3.1 Interface

A seguir é apresentado um diagrama que contém as telas da interface em console que deve ser criada. Cada retângulo representa uma "tela", ou seja, um conjunto de texto impresso com possivelmente entrada de dados. 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.



Observação: pode-se considerar que o usuário somente digitará entradas com escolhas possíveis. Ou seja, não é preciso tratar entradas incorretas.

> **Atenção:** A interface com o usuário deve seguir exatamente o especificado (incluindo ":" e espaços entre ")" e ":" e o restante do texto - os demais espaços não serão verificados). Se ela não for seguida, haverá desconto de nota. Não adicione outros textos além dos apresentados no diagrama e especificados.

As telas funcionam da seguinte forma:

- Ao entrar no programa, caso o usuário deseje carregar `Usuarios`, deve-se ler o arquivo texto informado usando a `PersistenciaDeUsuario` e usar os `Usuarios` carregados para criar o `GerenciadorDeUsuario`.
- Ao selecionar sair, caso o usuário queira salvar, deve-se salvar os `Usuarios` do `GerenciadorDeUsuario` em arquivo usando a `PersistenciaDeUsuario`.
- Opção 1: deve-se chamar o método entrar na `Catraca` informada com o id e `Data` informados.
- Opção 2: deve-se chamar o método sair na `Catraca` informada com o id e `Data` informados.
- Opção 3: deve-se fazer um registro manual (de entrada ou de saída, dependendo do que foi informado) no `Usuario` com id e data informados.
- Opção 4: deve-se cadastrar um novo usuário (v é para `Visitante`, a é para `Aluno` e f é para `Funcionario`). O novo `Usuario` deve ser adicionado ao `GerenciadorDeUsuario`.
- Opção 5: deve-se gerar um relatório de horas trabalhadas para todos os `Funcionarios`, apresentando o nome e as horas trabalhadas (método `getHorasTrabalhadas`)
- Opção 6: deve-se alterar o horário de fim (`setHorarioFim`) do `Aluno`.

3.2 Sugestões de implementação

Aproveite o código do EP1, fazendo as mudanças necessárias. Como já sugerido no EP1, crie funções auxiliares para reaproveitar comportamento das telas.

3.3 Exemplo

Segue um exemplo de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário.

```
Deseja carregar usuarios (s/n): s
Arquivo: usuario.txt
Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 4

Tipo (v, a ou f): a
Id: 1237
Nome: Claudio
Usuario cadastrado com sucesso

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 5

Mes: 9
Ano: 2024

Relatorio de horas trabalhadas
Maria: 15

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 6

Horario de fim da janela dos Alunos
Hora: 22
```

```
Minuto: 0

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 1

Catraca: 1
Id: 1237
Hora: 22
Minuto: 1
Segundo: 0
Dia: 28
Mes: 9
Ano: 2024
[Entrada] Catraca 1 travada

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 1

Catraca: 1
Id: 1235
Hora: 8
Minuto: 5
Segundo: 20
Dia: 2
Mes: 10
Ano: 2024
[Entrada] Catraca 1 abriu: id 1235

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 2

Catraca: 0
Id: 1235
```

```
Hora: 11
Minuto: 59
Segundo: 10
Dia: 2
Mes: 10
Ano: 2024
[Saida] Catraca 0 abriu: id 1235

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 3

Entrada (e) ou Saida (s)? e
Id: 1235
Hora: 13
Minuto: 0
Segundo: 5
Dia: 2
Mes: 10
Ano: 2024
Entrada manual registrada: id 1235

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 5

Mes: 10
Ano: 2024

Relatorio de horas trabalhadas
Maria: 3

Acesso ao predio
1) Entrada
2) Saida
3) Registro manual
4) Cadastro de usuario
5) Relatorio
6) Configuracao
0) Sair
Escolha uma opcao: 0
```

```
Deseja salvar usuarios (s/n): n
```

4 Entrega

O projeto deverá ser entregue até dia **22/11** em um Judge específico, disponível em <https://laboo.pcs.usp.br/ep/> (nos próximos dias vocês receberão um login e uma senha).

Atenção:

- Deve ser mantida a mesma dupla do EP1. É possível apenas *desfazer* a dupla. Com isso, cada aluno deve fazer uma entrega diferente (e em separado). Caso você deseje fazer isso, envie um e-mail para levy.siqueira@usp.br até dia **11/11** informando os números USP dos alunos e **mandando-o com cópia para a sua dupla**.
- Não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e os grupos envolvidos terão nota `0` no EP. Portanto, não envie o seu código para um colega de outro grupo!

Entregue todos os arquivos, inclusive o `main` e o `menu` (que devem **obrigatoriamente** ficar nos arquivos "main.cpp" e "menu.cpp", respectivamente), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, *podem* não ser reconhecidos e acarretar **nota 0**). O nome do arquivo não pode conter espaço, ".", acentos ou ter mais de 11 caracteres. Os códigos fonte não devem ser colocados em pastas. A submissão pode ser feita por qualquer um dos membros da dupla – recomenda-se que os dois submetam.

> **Atenção:** faça a submissão do mesmo arquivo nos 4 problemas (Parte 1, Parte 2, Parte 3 e Parte 4). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

Entregue todos os arquivos, inclusive o `main` e o `menu` (que devem **obrigatoriamente** ficar nos arquivos "main.cpp" e "menu.cpp", respectivamente), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, *podem* não ser reconhecidos e acarretar **nota 0**). O nome do arquivo não pode conter espaço, ".", acentos ou ter mais de 11 caracteres. Os códigos fonte não devem ser colocados em pastas. A submissão pode ser feita por qualquer um dos membros da dupla – recomenda-se que os dois submetam.

> **Atenção:** não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e **todos** os alunos dos grupos envolvidos terão **nota `0` no EP**. Portanto, **não envie** o seu código para um colega de outro grupo!

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 **apenas** uma verificação básica buscando evitar erros de compilação devido à erros de digitação do nome das classes e dos métodos públicos. **Note que a nota dada não é a nota final**: neste momento não são executados testes – o Judge apenas tenta

chamar todos os métodos definidos neste documento para todas as classes. Por exemplo, essa verificação é a seguinte para a classe `Registro`:

```
Registro* r = new Entrada(new Data(1, 1, 1, 1, 1, 2024));
Data* d = r->getData();
bool b = r->isManual();
delete r;
return true;
```

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

5 Dicas

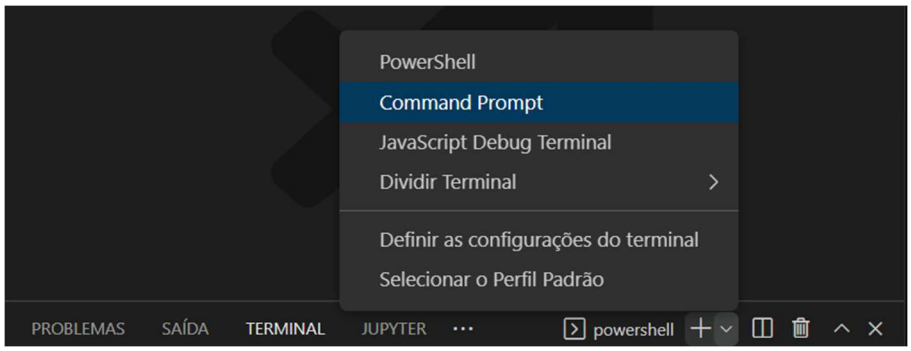
- **Entregue um EP que compila**! Caso você não consiga implementar um método (ou ele esteja com erro de compilação), faça uma implementação *dummy* dele. Uma implementação *dummy* é a implementação mais simples possível do método, que permite a compilação. Por exemplo, uma implementação *dummy* – e errada – do método salvar da classe `PersistenciaDoUsuario` é:

  ```
  void salvar(string arquivo, vector<Usuario*>* v) {
  }
  ```

- Caso o programa esteja travando, execute o programa no modo de depuração. O depurador informará o erro que aconteceu – além de ser possível depurar 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.
- É muito trabalhoso testar o programa ao executar o `main` *com menus*, já que é necessário informar vários dados para inicializar os registradores e a memória de dados. Para testar o programa faça o `main` chamar uma função de teste que cria objetos com valores interessantes para testar, sem pedir entrada para o usuário. Não se esqueça de remover a função de teste ao entregar a versão final do EP.
  - Uma opção é usar o recurso de paste no powershell (o terminal padrão do VSCode). É só clicar com o botão direito no terminal do VSCode que ele cola o texto que você tiver copiado.
  - Uma outra opção para testar é usar o comando:

    ```
    ep < entrada.txt > saida.txt
    ```

    Esse comando executa o programa `ep` usando como entrada do teclado o texto no arquivo `entrada.txt` e coloca em `saída.txt` os textos impressos pelo programa (sem os valores digitados). No caso do Windows, para rodar esse comando você precisa de um prompt de comando (por uma limitação do *PowerShell*). Para fazer isso, clique na seta para baixo do lado do + no terminal e escolha Command Prompt.



- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- 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 ao executar o programa no Windows. Veja a mensagem de erro do Judge para descobrir em qual classe acontece o problema. Caso você queira testar o projeto em um compilador similar ao do Judge, use o site https://github.com/features/codespaces.
    - 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*.

Atenção: jamais deixe o código fonte do seu EP público na Internet - algum aluno pode usá-lo e isso será identificado como plágio. Se você usar o GitHub, deixe o repositório privado (adicionando o outro membro da dupla como colaborador).

- Use o Fórum de dúvidas do EP no e-Disciplinas para esclarecer dúvidas no enunciado ou problemas de submissão no Judge.
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**