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 2023

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar um simulador de processamentos de sinais.

1 Introdução

O tema desse Exercício Programa continua sendo o programa de simulação análogo ao Simulink.

1.1 Objetivo

Neste segundo EP será implementada uma arquitetura mais completa de blocos operadores que processam sinais entre suas entradas e suas saídas, a partir de melhorias sobre aquilo já feito no primeiro EP.

Os componentes operadores terão funcionalidades idênticas àqueles desenvolvidos no EP1. Já os sinais terão também o mesmo princípio, porém com funcionalidades a mais. Além disso, será criado mais um tipo de módulo, uma persistência para os módulos e serão criadas outras classes para organização e complementação do código (melhor explicadas a seguir). Para isso serão empregados os conceitos aprendidos durante toda a disciplina, o que inclui herança, polimorfismo, classe abstrata, programação defensiva e persistência em arquivo.

2 Projeto

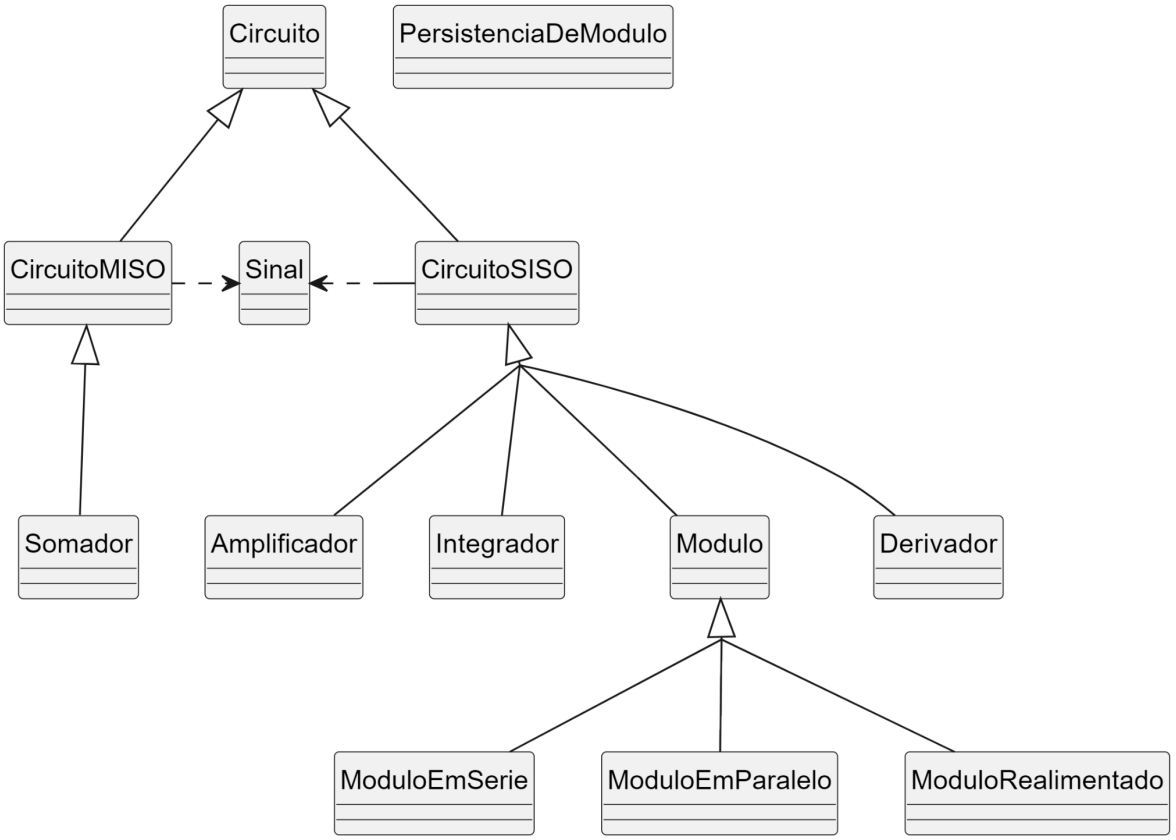
Deve-se implementar em C++ todas as classes presentes na diagrama a seguir além de criar uma `main` que permita o funcionamento do programa como desejado.

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 "Sinal.cpp" e "Sinal.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.



O enunciado não apresenta a palavra reservada `virtual` nos métodos. Coloque-a quando necessário.

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.

2.1 Classe Sinal

Um `Sinal` guarda um vetor `sequência` de tamanho `comprimento`, conforme explorado no EP1. Essa classe deve possuir apenas os seguintes **métodos públicos**:

```
Sinal(double *sequencia, int comprimento);
Sinal(double constante, int comprimento);
~Sinal();
double* getSequencia();
int getComprimento();
void imprimir(string nomeDoSinal);
void imprimir(); // para testes
void imprimir(int tamanho);
```

- A primeira opção de construtor é idêntica àquela desenvolvida no EP1, ou seja, copiando os valores de **sequência** a um novo vetor alocado dinamicamente de tamanho **comprimento.** Já a segunda versão do construtor deverá criar uma sequência constante de valor **constante** e tamanho **comprimento**. Ambas as versões não podem receber um valor de comprimento menor ou igual a zero. Se isso ocorrer, deve-se jogar uma exceção do tipo **invalid_argument.**
- O destrutor deverá destruir o que for alocado dinamicamente na solução adotada.
- O método **getSequencia** deverá retornar um ponteiro para a sequência interna de valores. Analogamente, **getComprimento** deverá retornar o comprimento dessa sequência.
- Para o método **imprimir** há três opções, bem distintas entre si. A primeira, que recebe como parâmetro uma string, é idêntica àquela implementada no EP1, com a utilização da biblioteca fornecida **Gráfico.h**. A segunda implementação, criada para auxiliar o teste por alunos, não possui nenhum parâmetro e deverá simplesmente imprimir todos os valores da sequência do sinal e terminar com "--", conforme o formato abaixo:

  ```
  0- [primeiro valor]
  1- [segundo valor]
  2- [terceiro valor]
  [...]
  [comprimento-2]- [penúltimo valor]
  [comprimento-1]- [último valor]
  --
  ```

  Exemplo de **comprimento** 6:

  ```
  0- 7
  1- 2.0
  2- 99
  3- 6.54
  4- -3
  5- 12.5
  --
  ```

  A última implementação deverá ter um comportamento análogo à segunda, porém com um limite máximo de valores impressos definido pelo parâmetro **tamanho.** Ou seja, se o comprimento for menor que o tamanho, imprime-se a sequência completa, se não imprime-se até a posição [**tamanho-1**] da sequência.

  Exemplo de **comprimento** 6 e **tamanho** 4::

  ```
  0- 7
  1- 2.0
  2- 99
  3- 6.54
  --
  ```

Atenção: se for instanciado algum objeto com o comando `new`, será necessário o destruir após a sua utilização.

Obs.: o método `Grafico::plot` por padrão restringe os eixos ao imprimir no terminal da seguinte forma: o eixo das coordenadas limita os valores ao intervalo de 0 a 10 e o eixo das abscissas representa sequências de no máximo 60 amostras. Caso queria alterar isso em seus testes, basta alterar as constantes definidas em `Grafico.cpp`. Lembre-se somente de retornar os valores iniciais ao submeter a resolução no Judge.

2.2 Classe Circuito

Essa classe é **abstrata** e deve possuir os seguintes métodos públicos:

```
Circuito();
virtual ~Circuito();
int getID();
void imprimir();
static int getUltimoID();
```

Ela será uma classe mãe de muitas outras e por isso carrega características que serão comuns a todas elas. A principal delas é que cada Circuito instanciado carrega consigo um número identificador único que deverá ser atribuído a partir do valor 1 e crescer sucessivamente. Assim o primeiro circuito possui ID = 1, o segundo possui ID = 2, o terceiro possui ID = 3 e assim por diante (use um atributo estático para saber o último ID criado).

- O construtor, portanto, deverá atribuir ao novo circuito o seu respectivo ID, de forma crescente.
- O destrutor não possui qualquer tarefa especial. Para fazer essa classe ser abstrata, **faça ele abstrato** (note que apesar de ele ser abstrato, você precisa implementá-lo no .cpp - é um detalhe do destrutor abstrato).
- O método `getID` deverá retornar o respectivo ID daquele circuito.
- O método imprimir deverá apresentar no terminal uma mensagem com o seguinte formato:

  `Circuito com ID [ID]`

  Exemplo:

  `Circuito com ID 12`

- Por fim, o método `getUltimoID` deverá ser estático e retornar o último ID atribuído. Caso nenhum circuito tenha sido criado ainda, deverá retornar o valor 0.

2.3 Classe CircuitoSISO

O termo SISO significa "*Single input - Single output*", ou seja, um sistema com apenas uma entrada e uma saída, por exemplo um telefone de lata, um megafone ou até mesmo um abajur. Essa classe abstrata é filha de Circuito e deve possuir os seguintes métodos públicos:

```
CircuitoSISO();
virtual ~CircuitoSISO();
Sinal* processar(Sinal* sinalIN); // abstrato
```

- O Construtor e o Destrutor não possuem qualquer tarefa especial.

- Assim, o papel dessa classe abstrata é definir a assinatura do método `processar` (com somente um sinal de entrada e um de saída) e que será implementado por cada classe filha. **Esse método deve ser abstrato**.

2.4 Classe CircuitoMISO

Já o termo MISO significa "*Multiple input - Single output*", ou seja, um sistema com apenas uma saída, porém várias entradas, como um chuveiro que tem dois registros para quente e frio ou um piano. Essa classe abstrata é filha de Circuito e deve possuir os seguintes métodos públicos:

```
CircuitoMISO();
virtual ~CircuitoMISO();
Sinal* processar(Sinal* sinalIN1, Sinal* sinalIN2);
```

- O Construtor e o Destrutor também não possuem qualquer tarefa especial.
- Assim, o papel dessa classe abstrata é definir a assinatura do método `processar` (com duas entradas e uma saída) que será implementado por cada classe filha. **Esse método deve ser abstrato**.

2.5 Somador

Essa classe é a única filha concreta de CircuitoMISO. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Somador();
virtual ~Somador();
```

- Além do fato de ser filha de CircuitoMISO, o construtor e destrutor não possuem qualquer papel especial.
- Seu método `processar` funciona de modo idêntico àquele método homônimo da classe `Somador` do EP1. Ou seja, ele deve retornar um ponteiro para um novo `Sinal` com uma sequência que seja a soma "termo a termo" dos elementos das sequências dos dois sinais de entrada passados como argumentos. Para mais detalhes veja o enunciado do EP1.

2.6 Amplificador

Esse operador é filho concreto de CircuitoSISO. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Amplificador(double ganho);
virtual ~Amplificador();
void setGanho(double ganho);
double getGanho();
```

- Além do fato de ser filha de CircuitoSISO, o construtor e destrutor não possuem qualquer papel diferente daqueles relativos à classe `Amplificador` do EP1, armazenando assim o ganho recebido no construtor.
- Seus métodos `processar`, `setGanho` e `getGanho` funcionam de modo idêntico aos métodos homônimos da classe `Amplificador` do EP1. O método `processar` deverá criar um novo objeto `Sinal` com uma nova sequência criada a partir da sequência de sinalIN com cada elemento multiplicado pelo ganho, que é informado pelo construtor e pelo método `setGanho` e retornado pelo método `getGanho`. Para mais detalhes veja o enunciado do EP1.

2.7 Derivador

Essa classe é filha concreta de CircuitoSISO. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Derivador();
virtual ~Derivador();
```

- Além do fato de ser filha de CircuitoSISO, seus construtor e destrutor não possuem qualquer papel especial.
- Seu método **`processar`** funciona de modo idêntico àquele método homônimo da classe **`Derivador`** do EP1. Ele deve retornar um ponteiro para um novo Sinal com uma sequência que seja a diferença dos valores da entrada consecutivos. Para mais detalhes veja o enunciado do EP1.

2.8 Integrador

Essa classe é filha concreta de CircuitoSISO. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Integrador();
virtual ~Integrador();
```

- Além do fato de ser filha de CircuitoSISO, seus construtor e destrutor não possuem qualquer papel especial.
- Seu método **`processar`** funciona de modo idêntico àquele método homônimo da classe **`Integrador`** do EP1. Ele deve retornar um ponteiro para um novo Sinal com uma sequência que seja a somatória acumuladora dos valores do sinal de entrada. Para mais detalhes veja o enunciado do EP1.

2.9 Modulo

Essa classe é filha de CircuitoSISO, porém **ainda é abstrata**. Ela representa um circuito do tipo SISO e que possui uma lista de outros circuitos SISOs internos. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
Modulo();
virtual ~Modulo();
void adicionar(CircuitoSISO* circ);
list<CircuitoSISO*>* getCircuitos();
```

- O construtor e o destrutor devem gerir a existência de um **`list<CircuitoSISO*>`**
- O método **`imprimir`** deve ser redefinido para apresentar no terminal uma lista com todos os circuitos listados internamente, conforme o modelo:

```
Modulo com ID [ID do módulo] e:
Circuito com ID [ID do primeiro circuito da lista]
Circuito com ID [ID do segundo circuito da lista]
Circuito com ID [ID do terceiro circuito da lista]
[...]
Circuito com ID [ID do último circuito da lista]
--
```

Um exemplo de módulo com 5 circuitos internos:

```
Modulo com ID 9 e:
Circuito com ID 2
Circuito com ID 1
Circuito com ID 7
Circuito com ID 5
Circuito com ID 3
--
```

- O método `adicionar` deve adicionar um `CircuitoSISO` à lista interna do Módulo.
- O método getter deve retornar a lista interna que armazena os circuitos SISOs.

2.10 ModuloEmSerie

Essa classe é filha concreta de Modulo. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
ModuloEmSerie();
virtual ~ModuloEmSerie();
```

O objetivo dessa classe é conter diversos outros circuitos armazenados em série, ou seja, com a entrada de um na saída do outro.

- O construtor, o destrutor e os métodos `adicionar` e `getCircuitos` não devem fazer nada mais além daquilo já descrito para Modulo.
- O método `processar` deve realizar o processamento do sinal de entrada de forma sequencial (na ordem adicionada, ou seja, o primeiro CircuitoSISO da lista recebe o Sinal de entrada, o segundo CircuitoSISO recebe como entrada o Sinal de saída retornado pelo primeiro CircuitoSISO, o terceiro CircuitoSISO recebe como entrada o Sinal de saída retornado pelo segundo CircuitoSISO, etc.) e retornar a saída do último circuito da lista, pois todos os circuitos devem estar conectados **em série**. Caso a lista não possua nenhum CircuitoSISO, deve-se jogar uma exceção do tipo `logic_error`.

2.11 ModuloEmParalelo

Essa classe é filha concreta de Modulo. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
ModuloEmParalelo();
virtual ~ModuloEmParalelo();
```

O objetivo dessa classe é conter diversos outros circuitos dispostos em paralelo, ou seja, com as entradas partindo do mesmo ponto e as saídas também conectadas..

- O construtor, o destrutor e os métodos `adicionar` e `getCircuitos` devem fazer aquilo já descrito em 2.9
- O método `processar` deve realizar o processamento do sinal de entrada de forma paralela em todos os circuitos da lista e retornar a **soma** de todas as saídas desses circuitos. Para isso, utilize um `somador` internamente. Caso a lista não possua nenhum CircuitoSISO, deve-se jogar uma exceção do tipo `logic_error`.

2.12 ModuloRealimentado

Essa classe é filha concreta de Modulo. Além dos métodos públicos da superclasse (que podem ser redefinidos conforme necessário), essa classe deve possuir apenas os seguintes métodos públicos:

```
ModuloRealimentado();
virtual ~ModuloRealimentado();
```

Essa classe, propositalmente, foi implementada no EP de tal forma que fosse simples sua reconstrução no EP2. Seu princípio é idêntico: simular um circuito realimentado. Porém, a única diferença é que substituiremos um `Piloto` por um `ModuloEmSerie`, a fim de ela ser mais genérica.

- O construtor e o destrutor devem cuidar do `ModuloEmSerie` interno, além de outros objetos internos que desejar-se utilizar.
- Seu método `processar` é idêntico àquele feito no EP1. A única diferença é que no lugar de utilizarmos um `Piloto`, utilizaremos um `ModuloEmSerie`. Ou seja, basta trocar uma classe pela outra no código que está resolvido (note que diferentemente do EP1, o `ModuloRealimentado` não possui ganho - simplesmente use um `ModuloEmSerie`).
- O método `adicionar` deve inserir um circuito na lista do `ModuloEmSerie` interno.
- O método `getCircuitos` deve retornar a lista de circuitos do `ModuloEmSerie` interno.

2.13 PersistenciaDeModulo

Por fim, esta classe, possivelmente a mais trabalhosa, possui um objetivo simples: armazenar em arquivo um `Modulo` de modo que seja possível o recriar posteriormente todos os seus circuitos internos. Vale ressaltar que para simplificar não será exigido que se mantenham os IDs dos circuitos, apenas a ordem deles no `Modulo`. Seus métodos são os seguintes:

```
PersistenciaDeModulo(string nomeDoArquivo);
virtual ~PersistenciaDeModulo();
void salvarEmArquivo(Modulo* mod);
Modulo* lerDeArquivo();
```

- O construtor deverá receber o nome do arquivo que será utilizado para o armazenamento do `Modulo.`
- O destrutor não faz nada especial.
- O método `salvarEmArquivo` deve reescrever o arquivo armazenando o `Modulo` passado como parâmetro.
- O método `lerDeArquivo` deve retornar o Modulo gerado a partir daquilo que está escrito no arquivo. Caso haja um erro ao abrir o arquivo, será necessário jogar uma exceção do tipo `invalid_argument` e caso durante a leitura haja uma formatação incorreta do arquivo, deve-se jogar uma exceção do tipo `logic_error`.

O arquivo deve possuir o seguinte formato. Uma lista vertical em que cada linha representa um circuito armazenado. Os módulos são representados pelas letras 'S', 'P' e 'R' (para `ModuloEmSerie`, `ModuloEmParalelo` e `ModuloRealimentado`, respectivamente) e possuem seus circuitos internos listados logo abaixo dessa linha. Para indicar o final da lista do `Modulo`, há uma linha com a letra 'f'. Se for aberto um segundo modulo dentro do primeiro, a primeira letra 'f' que aparecer no arquivo deve indicar o fechamento do último módulo aberto. Por fim, os circuitos Amplificador, Integrador, Derivador recebem cada um as respectivas letras 'A', 'I' e 'D', de modo que além da letra, o Amplificador tem seu ganho escrito ao lado dela com um espaço de separação.
Exemplos de arquivo:

Exemplo 1:

```
S
I
P
A 0.2
D
f
```

```
A 77
f
```

(note que o amplificador de ganho 0.2 está em paralelo com o Derivador, mas ambos estão em série com o Amplificador de ganho 77 e um integrador)

Exemplo2:

```
R
P
I
A 0.4
D
f
f
```

(note que dessa forma todos os operadores estão em paralelo entre si)

Exemplo 3:

```
S
R
I
A 0.1
P
D
I
f
f
A 0.5
f
```

Note a beleza que é ter um `ModuloEmParalelo` dentro de um `ModuloRealimentado` dentro de um `ModuloEmSerie`!

Observação: Como é possível notar, há bastante recursão nesses métodos de salvar e ler, por isso, recomenda-se a utilização de funções auxiliares.

Atenção: ao passar um arquivo já aberto como parâmetro, é necessário o passar por referência (ofstream&) ou ponteiro (ofstream*), pois não é possível fazer uma cópia dele.

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 deve criar um programa que utiliza as classes especificadas para criar sinais e circuitos que os processem conforme o desejo do usuário. Será possível realizar atividades em **dois modos principais**: um **modo livre** em que o usuário cria um sinal e realiza operações em cima dele e um outro modo que

o circuito utilizado para processar o sinal é obtido a partir de um arquivo, conforme descrito anteriormente.

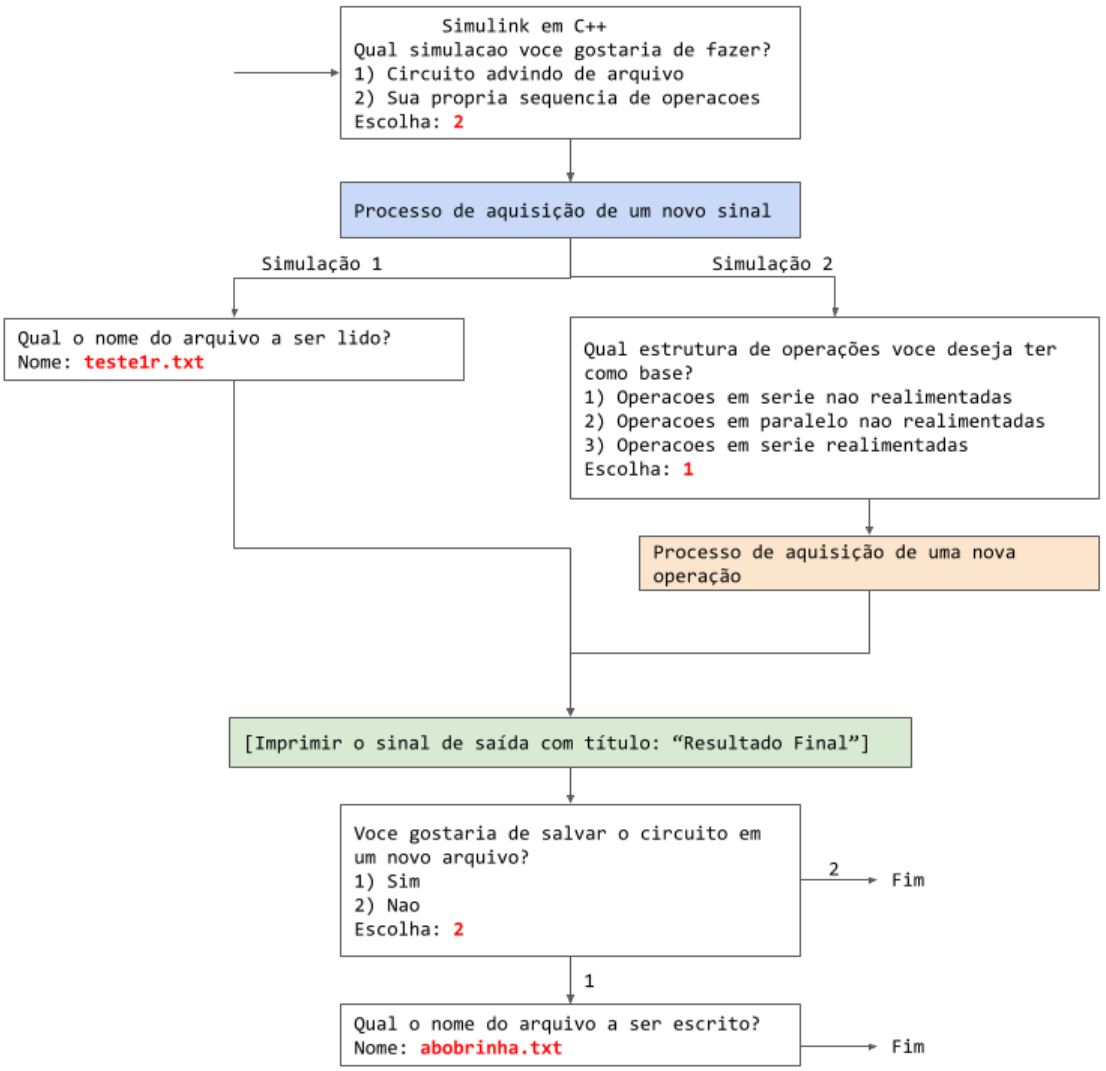
Obs.: considere que todos os sinais criados pelo usuário possuem tamanho 60.

3.1 Biblioteca cmath

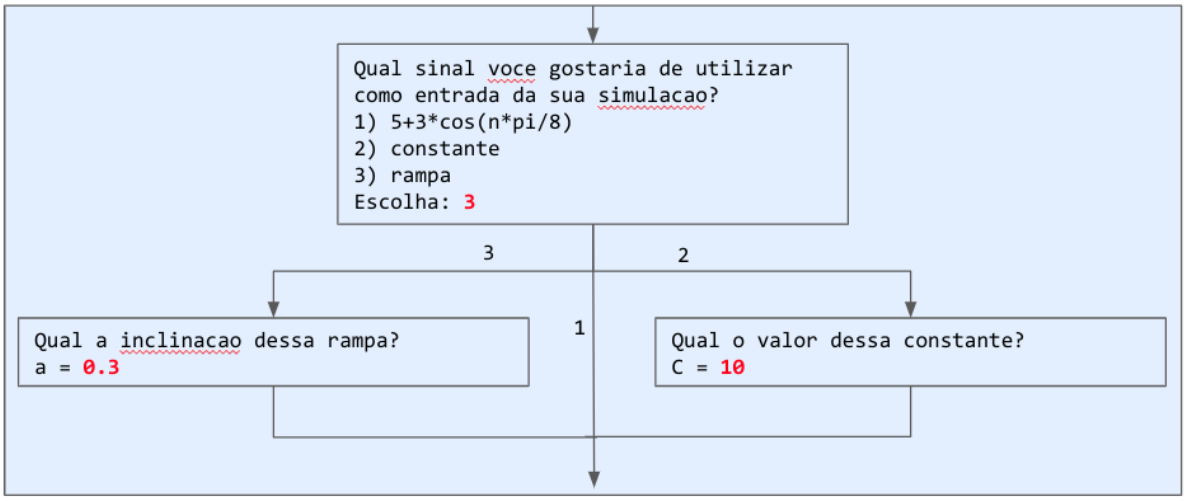
Além das bibliotecas apresentadas no curso e os arquivos ".h" das classes criadas, será necessário incluir a biblioteca <cmath> padrão de C++. Ela será utilizada para criar um sinal cossenoidal com sua função `cos` e sua constante interna π intitulada de M_PI.

3.2 Interface

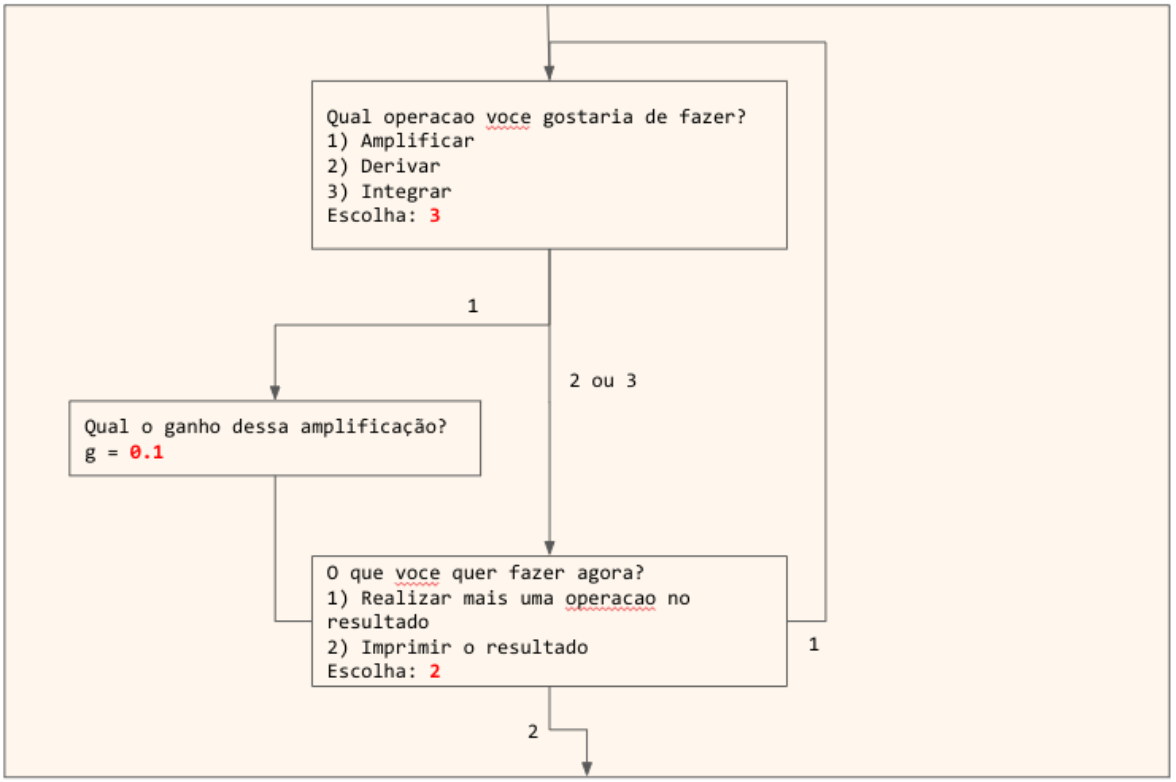
Para que o usuário interaja com o programa é necessária uma interface no terminal. A seguir é apresentado um diagrama que contém as telas do programa. Em **vermelho** estão os dados digitados pelo usuário. Os blocos brancos são as impressões que serão lançadas no terminal. Em verde estão os blocos que indicam a execução de uma função `imprimir(string)` do `Sinal`. Por fim, em azul estão os blocos que referenciam a ação de adquirir um novo sinal e em laranja estão os blocos que referenciam a ação de adquirir uma nova operação, apresentados posteriormente.



O bloco de aquisição de um novo sinal é apresentado a seguir.



O bloco de aquisição de uma nova operação é apresentado a seguir.



Nota-se que o modo em que o usuário cria sua própria sequência de operações é bem mais limitado do que o modo a partir da leitura de um arquivo, pois para simplificar o código foi decidido não permitir o usuário adicionar um módulo dentro de outro no modo customizado.

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

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

3.3 Sugestões de implementação

O processo de aquisição de um **novo sinal** do usuário foi representado isoladamente. Isso indica que uma boa maneira de implementar essa tela seja criando uma função, por exemplo `Sinal* novoSinal()`, que execute todo o processo de aquisição de um novo sinal de entrada e retorne um ponteiro para esse objeto criado. Assim, haverá uma organização melhor no código e, acima de tudo, será possível reaproveitar o código do EP1, pois essa função é idêntica.

Analogamente, pode-se criar um bloco para a aquisição de uma nova operação, haja vista que ela pode ser usada recursivamente quando o usuário quiser realizar uma nova operação após a pergunta **"O que voce quer fazer agora?"**. Assim, há um caso de recursão sobre essa função, o que indica que deve-se criar uma função como `CircuitoSISO* novaOperacao()` para poder ser chamada recursivamente. (Observe que mesmo com nome igual, esta função está diferente daquela homônima do EP1).

3.4 Exemplos

Seguem alguns exemplos de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário.

3.4.1 Exemplo 1

```
        Simulink em C++
Qual simulacao voce gostaria de fazer?
1) Circuito advindo de arquivo
2) Sua propria sequencia de operacoes
Escolha: 2

Qual sinal voce gostaria de utilizar como entrada da sua simulacao?
1) 5+3*cos(n*pi/8)
2) constante
3) rampa
Escolha: 3

Qual a inclinacao dessa rampa?
a = 0.01

Qual estrutura de operações voce deseja ter como base?
1) Operacoes em serie nao realimentadas
2) Operacoes em paralelo nao realimentadas
3) Operacoes em serie realimentadas
Escolha: 2

Qual operacao voce gostaria de fazer?
1) Amplificar
2) Derivar
3) Integrar
Escolha: 1

Qual o ganho dessa amplificacao?
g = 0.2

O que voce quer fazer agora?
1) Inserir mais uma operacao
```

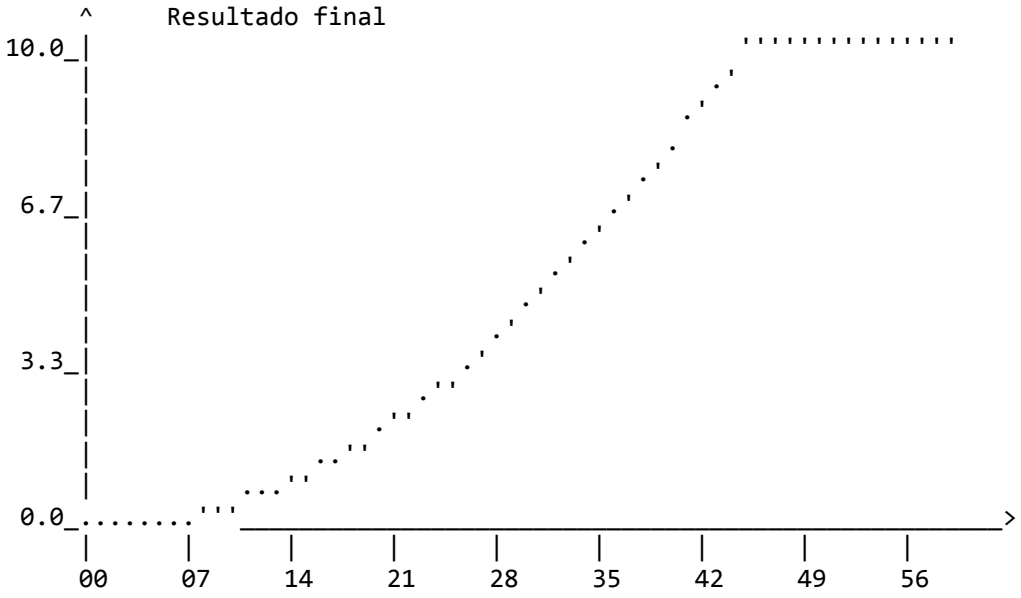
```
2) Imprimir o resultado
Escolha: 1

Qual operacao voce gostaria de fazer?
1) Amplificar
2) Derivar
3) Integrar
Escolha: 2

O que voce quer fazer agora?
1) Inserir mais uma operacao
2) Imprimir o resultado
Escolha: 1

Qual operacao voce gostaria de fazer?
1) Amplificar
2) Derivar
3) Integrar
Escolha: 3

O que voce quer fazer agora?
1) Inserir mais uma operacao
2) Imprimir o resultado
Escolha: 2
```




```
Voce gostaria de salvar o circuito em um novo arquivo?
1) Sim
2) Nao
Escolha: 1

Qual o nome do arquivo a ser escrito?
Nome: abobrinha.txt
```

3.4.2 Exemplo 2: com aplicação

Atenção: a compreensão das contas a seguir não é necessária para a resolução do EP. Apenas o importante é compreender que o projeto possui aplicações reais e pode ser utilizado de fato nos

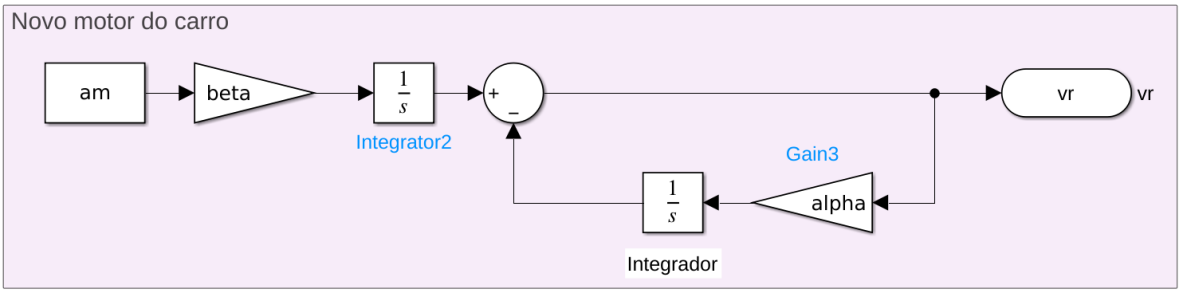
projetos de engenharia. Em amarelo estará destacada a parte que pode ser utilizada como teste do código.

Retomando o exemplo do piloto automático do EP1, é possível aprofundarmos esse caso se modelarmos o motor do carro de forma mais complexa, por exemplo, adicionando o atrito. Assim, além da integral da aceleração, haverá também um atrito para calcularmos a velocidade resultante. O problema é que um modelo aceitável de atrito é proporcional à velocidade, o que torna as equações mais complexas. Mas acredite se quiser: esse nosso simulador dá conta de resolver até esse caso.

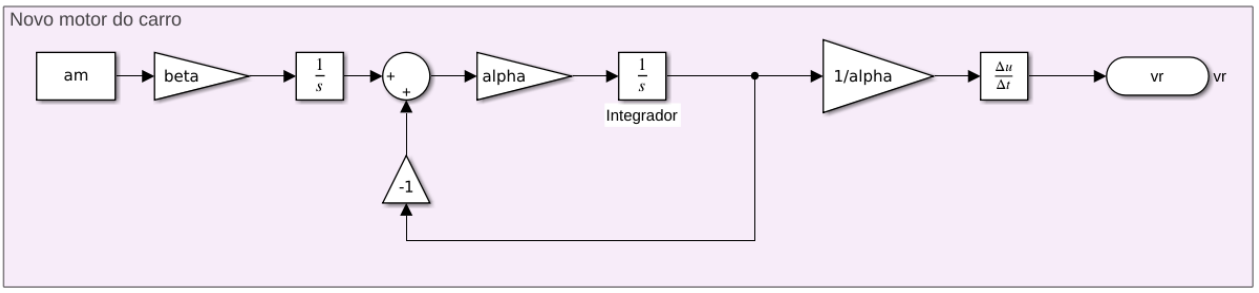
Como nosso objetivo aqui é programação, vou adiantar as contas e contar para vocês que nessa situação a velocidade do carro pode ser descrita pela seguinte equação:

$$v_r = \beta \cdot \int a_m - \alpha \cdot \int v_r$$

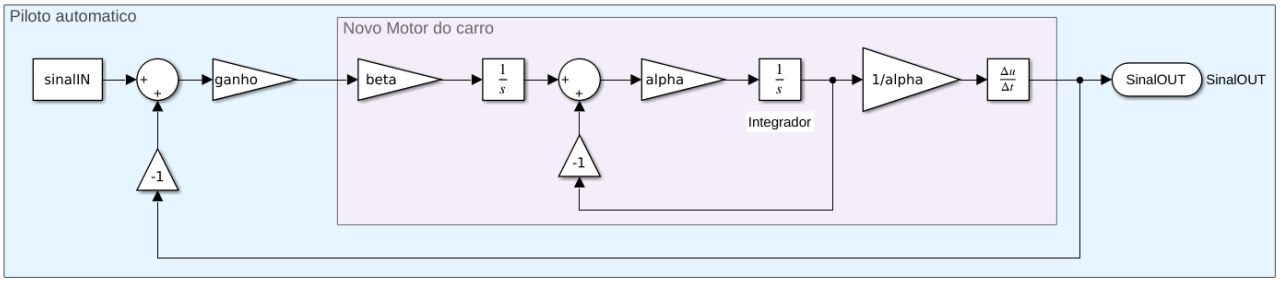
, em que α e β são constantes , a_m é a aceleração do motor que passamos no pedal do acelerador e v_r é a velocidade resultante desejada. Assim, podemos modelar o motor com o seguinte esquema:



Este esquema não é possível de implementarmos com o nosso código, pois a realimentação está diferente. Porém, é possível fazer um remanejamento dos componentes de forma a obtermos o mesmo resultado, porém com um modelo possível de simularmos:



Se inserimos isso, portanto, no nosso piloto automático, temos:



Assim, basta definirmos o valor de alpha, beta e o ganho que poderemos simular esse projeto.

Supondo um exemplo, beta = 0.2, alpha = 0.02 e ganho = 0.4, temos que o nosso arquivo `PilotoAutomatico.txt` terá o seguinte conteúdo:

```
R
A 0.4
A 0.2
I
R
A 0.02
I
f
A 50
D
f
```

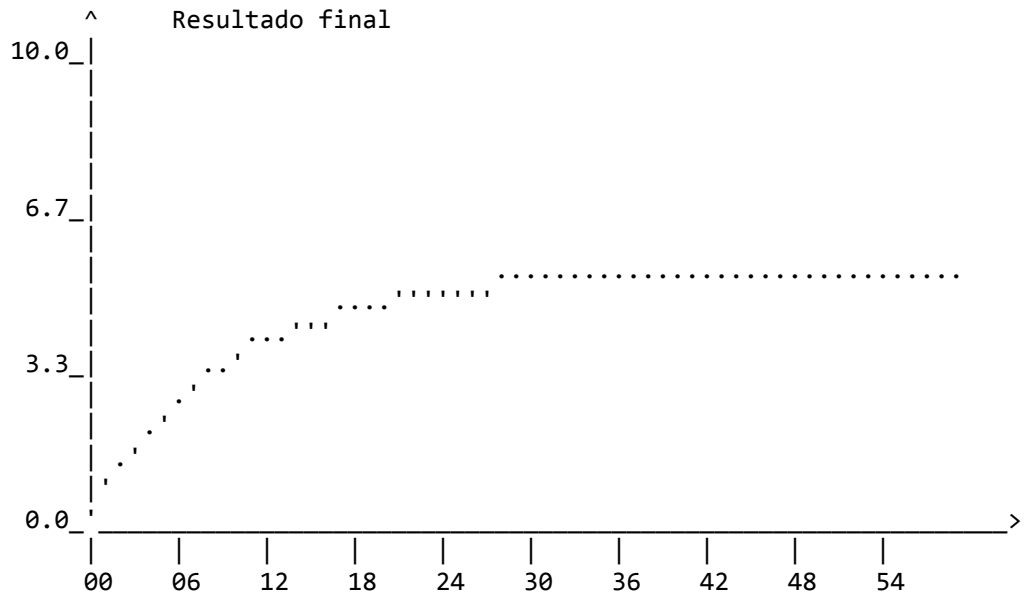
Deste modo, esta será a interface do programa quando a entrada (velocidade almejada) for uma constante de valor 7.0:

```
        Simulink em C++
Qual simulacao voce gostaria de fazer?
1) Circuito advindo de arquivo
2) Sua propria sequencia de operacoes
Escolha: 1

Qual sinal voce gostaria de utilizar como entrada da sua simulacao?
1) 5+3*sen(n*pi/8)
2) constante
3) rampa
Escolha: 2

Qual o valor dessa constante?
C = 7

Qual o nome do arquivo a ser lido?
Nome: pilotoAutomatico.txt
```



```
Voce gostaria de salvar o circuito em um novo arquivo?
1) Sim
2) Nao
Escolha: 2
```

4 Entrega

O projeto deverá ser entregue até dia **08/12** em <https://laboo.pcs.usp.br/ep/>.

> **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 24/11 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 3 problemas (Parte 1, Parte 2 e Parte 3). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

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, parte dessa verificação é a seguinte para a classe `Sinal`:

```
double sequencia[2] = {1, 2};
Sinal* s1 = new Sinal(sequencia, 2);
Sinal* s2 = new Sinal(5, 3);
double* s = s->getSequencia();
int c = s1->getComprimento();
s1->imprimir("a");
s1->imprimir();
s1->imprimir(2);
delete s1;
delete s2;
```

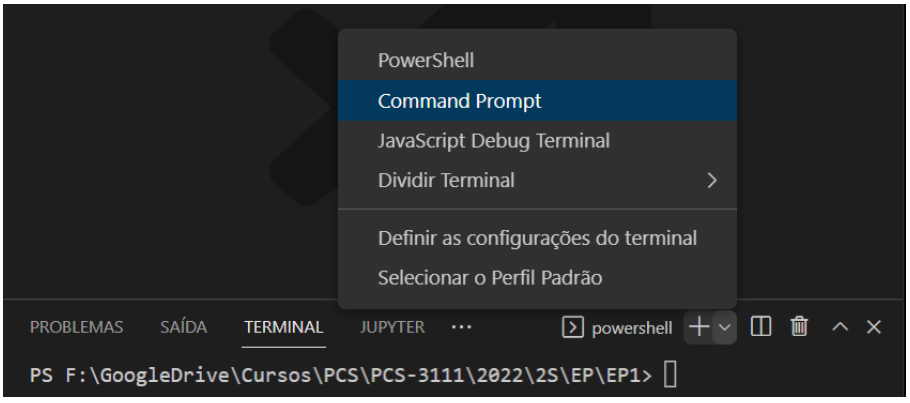
Você pode submeter quantas vezes quiser, sem desconto na nota.

5 Dicas

- 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 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.
- **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 `getCircuitos` da classe **ModuloEmParalelo** é:

```
list<CircuitoSISO*>* ModuloEmParalelo::getCircuitos()
      return nullptr;
}
```

- 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.**