UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

SIMULADOR DE CIRCUITOS DIGITAIS

PROFESSOR: ADELARDO ADELINO DANTAS DE MEDEIROS

O objetivo é desenvolver em C++ um simulador de circuitos lógicos, composto por portas lógicas de 2 ou mais entradas (ou de uma entrada, no caso da NOT) dos seguintes tipos:

- NOT (NEGAÇÃO)
- AND (E), NAND (NOT AND)
- OR (OU), NOR (NOT OR)
- XOR (OU EXCLUSIVO), NXOR (NOT XOR)

As entradas e saídas do circuito e das portas devem lidar com sinais lógicos verdadeiros (T - TRUE), falsos (F - FALSE) ou indefinidos (? - UNDEF), realizando as operações lógicas básicas (AND, OR e NOT) das seguintes maneiras:

A	B	A AND B
?	?	?
?	F	F
?	T	?
F	?	F
F	F	F
F	T	F
T	?	?
T	F	F
T	T	T

A	B	A OR B
?	?	?
?	F	?
?	T	T
F	?	?
F	F	F
F	T	T
T	?	T
T	F	T
T	T	T

A	B	A XOR B
?	?	?
?	F	?
?	T	?
F	?	?
F	F	F
F	T	T
T	?	?
T	F	T
T	T	F

A	NOT A
?	?
F	T
T	F

A simulação deve ser capaz de lidar com circuitos contendo ciclos, calculando as saídas ou informando que uma ou mais saídas ficam UNDEF quando não for possível a sua determinação (TRUE ou FALSE).

Os dados de entrada a serem fornecidos pelo usuário, via interface ou arquivo, são:

- Número de entradas e saídas do circuito.
- Número de portas lógicas do circuito.
- Para cada uma das portas lógicas:
  - O tipo de porta (AND, NOT, etc.).
  - O nº de entradas da porta (exceto NOT).
  - Para cada entrada da porta:
    - A origem do sinal lógico: uma porta ou uma das entradas do circuito.
- Para cada uma das saídas do circuito:
  - A origem do sinal lógico: uma porta ou uma das entradas do circuito.

Tendo em vista que um dos objetivos principais do projeto é praticar a utilização do polimorfismo baseado em métodos virtuais, além de utilizar regras de boa programação, algumas regras devem ser **obrigatoriamente** seguidas:

- O aplicativo deve ser programado baseando-se em objetos polimórficos para modelagem das portas lógicas. Ou seja, não deve haver instruções de controle de fluxo (`if`, `switch`, ternários, etc.) que mudem a forma de execução de acordo com o tipo da porta (OR, NOT, NAND, etc.), exceto no tratamento imediatamente seguinte à leitura (do arquivo ou da interface) do tipo de porta a ser incluído no circuito.
- As classes que representam as portas lógicas e o simulador de circuitos devem se basear e utilizar o tipo `bool3S` fornecido, sem modificá-lo ou ignorá-lo.
- O programa deve se basear na implementação parcial fornecida das classes das portas e da classe `Circuito`, sem modificar suas declarações. Além das funcionalidades já concluídas, devem ser desenvolvidos ou completados **todos** os métodos previstos na implementação parcial, incluindo:
  - Construtores, destrutores e sobrecarga de operadores.
  - Definição de novo circuito pelo console (`digitar`), mesmo que na versão final a interface seja visual.
  - Leitura de circuito de arquivo (`ler`).
  - Impressão em stream (`imprimir`).
  - Salvamento em arquivo (`salvar`), utilizando o método `imprimir`.
  - Geração das saídas para uma dada combinação das entradas (`simular`).

ARQUIVO

Os arquivos de leitura e escrita dos circuitos devem seguir um padrão, de tal forma que possam ser reconhecidos pelos programas de outros alunos. O formato que deve ser seguido **ao salvar**[1] um arquivo é o seguinte:

CIRCUITO *Nin Nout Nportas*
PORTAS
id_port) *type n_in*: *id_orig_in*$_1$ … *id_orig_in*$_{n\_in}$
…
id_port) *type n_in*: *id_orig_in*$_1$ … *id_orig_in*$_{n\_in}$
SAIDAS
id_out) *id_orig_out*
…
id_out) *id_orig_out*

Os trechos em **negrito** devem estar presentes no arquivo salvo. Os trechos em *itálico* correspondem aos locais onde serão salvos no arquivo os valores correspondentes ao circuito. O significado dos valores é o seguinte:

- *Nin*: número de entradas do circuito
- *Nout*: número de saídas do circuito
- *Nportas*; número de portas do circuito
- *id_port*: identificador da porta (1 ≤ *id_port* ≤ *Nportas*)
- *type*: tipo da porta:
  - NT = porta NOT
  - AN = porta AND
  - NA = porta NAND
  - OR = porta OR
  - NO = porta NOR
  - XO = porta XOR
  - NX = porta NXOR
- *n_in*: número de entradas da porta lógica (1 para NOT; 2 ou mais para as outras).
- *id_orig_in*$_i$: identificador da origem do sinal lógico da i-ésima entrada da porta (compatível com o número de entradas *n_in*).
  - > 0 se o sinal vem da saída de uma porta (1 ≤ *id_orig_in* ≤*Nportas*)
  - < 0 se o sinal vem de uma entrada do circuito (-1 ≥ *id_orig_in* ≥ - *Nin*)
- *id_out*: identificador da saída (1 ≤ *id_out* ≤ *Nout*)
- *id_orig_out*: identificador da origem do sinal lógico da saída do circuito:
  - > 0 se o sinal vem da saída de uma porta (1 ≤ *id_orig_out* ≤*Nportas*)
  - < 0 se o sinal vem de uma entrada do circuito (-1 ≥ *id_orig_out* ≥ - *Nin*)

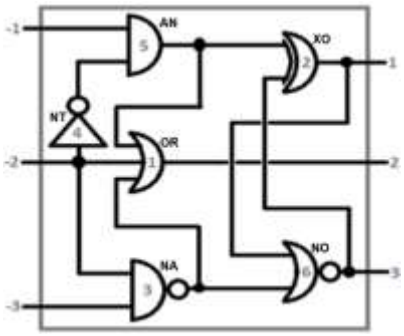
As portas e saídas devem estar ordenadas no arquivo, de modo que as linhas correspondentes à primeira porta e à primeira saída no arquivo devem ter *id_port* e *id_out* iguais a 1; as últimas devem ter *id_port* e *id_out* iguais a *Nportas* e *Nout*, respectivamente.

Durante a leitura, o arquivo de descrição de circuito deve ser verificado quanto à existência dos campos obrigatórios e da validade dos valores. **Não precisa nem deve ser verificada o tipo e quantidade dos separadores entre campos** (espaços, quebras de linha, etc.), embora deva ser verificada a existência de ")" após os identificadores de portas e de saídas e de ":" após o número de entradas de portas, com ou sem separadores antes ou depois deles. Caso haja alguma incoerência **nos dados** (campos obrigatórios faltando, referência a uma *id* inexistente, etc.), o arquivo não deverá ser lido.

DICA: **Não complique** a implementação do método de leitura, pois o `operator>>` já lida com informações separadas por quantidades e tipos arbitrários de separadores (espaço, ENTER, TAB), além de ser capaz de ler strings sem espaços, como as do arquivo. Não há necessidade de usar `getline`, `ignore`, `ws` ou nenhuma função mais avançada de leitura de dados. Basta ler os valores sucessivos com `operator>>` e descartar o arquivo caso algum valor lido seja inválido. O eventual conteúdo excedente no final do arquivo (valores e/ou linhas a mais) será ignorado. Informação excedente em uma linha (por exemplo, uma *id_orig_in* de entrada a mais do que o que deveria existir em uma porta) será lida como se fosse a informação seguinte, não passará na próxima validação (valor incorreto) e o arquivo será rejeitado.

[1] Em leitura, admitem-se arquivos que não sigam exatamente esse formato, conforme detalhado mais à frente, desde que as informações necessárias estejam presentes, válidas e na ordem correta.

EXEMPLOS DE ARQUIVOS




```
CIRCUITO 3 3 6
PORTAS
1) OR 3: 5 -2 3
2) XO 2: 5 3
3) NA 2: -2 -3
4) NT 1: -2
5) AN 2: -1 4
6) NO 2: 2 3
SAIDAS
1) 2
2) 1
3) 6
```

Exemplos de arquivos **válidos** em leitura (o primeiro exemplo também é válido em escrita)	

`CIRCUITO 2 1 3`
`PORTAS`
`1) OR 2: -1 -2`
`2) NT 1: -2`
`3) AN 2: 1 2`
`SAIDAS`
`1) 3`
	`CIRCUITO 2 1 3`
	`PORTAS 1) OR 2:`
	`-1 -2 2) NT 1: -2`
	`3) AN 2: 1`
	
	`2 SAIDAS 1)`
	`3`

`CIRCUITO`
`2 1 3 PORTAS`
 	
`1) OR 2: -1 -2`
`2) NT 1 : -2`
`3) AN 2: 1 2`
`SAIDAS`
`1 ) 3`
	`CIRCUITO 2 1 3`
	`PORTAS`
	`1 ) OR 2 : -1 -2`
	`2 ) NT 1 : -2`
	`3 ) AN 2 : 1 2`
	`SAIDAS`
	`1) 3`
	`4) 22`
	`OUTRA: 3.141592654`

`CIRCUITO 2 1 3`
`PORTAS`
`1)OR 2:-1 -2`
`2)NT 1:-2`
`3)AN 2:1 2`
`SAIDAS`
`1)3`
	`CIRCUITO 2 1 3`
	`PORTAS`
	`1) OR 2 : -1 -2`
	`2) NT 1 : -2`
	`3) AN 2 : 1 2`
	`SAIDAS`
	`1) 3`

Exemplos de arquivos **inválidos** em leitura e em escrita	

`Circuito 2 1 3`
`PORTS`
`1) OR 2: -1 -2`
`2) NT 1: -2`
`3) AN 2: 1 2`
 	
`1) 3`
	`CIRCUITO 2 2 3`
	`PORTAS`
	`1) OR 2: -1 -2`
	`2) NT 1: -2`
	`3) AN 2: 1 2`
	`SAIDAS`
	`1) 3`

`CIRCUITO 2 1 3`
`PORTAS`
`1) OR 2: -1 -2`
`3) NT 1: -2`
`2) AN 2: 1 2`
`SAIDAS`
`0) 3`
	`CIRCUITO 2 1 3`
	`PORTAS`
	`1 OR 2: -1 -2`
	`2 NT 1: -2`
	`3 AN 2: 1 2`
	`SAIDAS`
	`1 3`

`CIRCUITO 2 1 3`
`PORTAS`
`1) OR 2 -1 -2`
`2) NT 1 -2`
`3) AN 2 1 2`
`SAIDAS`
`1) 3`
	`CIRCUITO 2 1 3`
	`PORTAS`
	`1) OR 2: -1 -2`
	`2) NT 1: -3`
	`3) AN 2: 4 2`
	`SAIDAS`
	`1) 0`

`CIRCUITO 2 1 3`
`PORTAS`
`1) OR 2: ) -1 -2`
`2) NT 1 0 : -2`
`3 0 ) AN 2: 1 2`
`SAIDAS`
`1] 3`
	`CIRCUITO 2 1 3`
	`PORTAS`
	`1) OR 2: -1`
	`2) NT 1: -2 2`
	`3) AN 2: 1 2`
	`SAIDAS`
	`1) 3`

`CIRCUITO: 2 1 3`
`PORTAS:`
`1) OR 2: -1 -2`
`2) NT 1: -2`
`3) AN 2: 1 2`
`SAIDAS:`
`1) 3`
	`CIRCUITO 2 1 3`
	`PORTAS`
	`1) OR 2:: -1 -2`
	`2) NT 1:: -2`
	`3) AN 2:: 1 2`
	`SAIDAS`
	`1) 3`

ALGORITMOS

SIMULAR CIRCUITO:

```
// TIPO DE DADO Port
Port:
  vector<int> id_in
    // ids das entradas da porta:
  bool3S out_port // Saída da porta

// DADOS GLOBAIS
vector<Port> ports
  // portas do circuito
vector<int> id_out
  // ids das saídas do circuito

// ENTRADA:
vector<bool3S> in_circ
  // Entradas do circuito

// SAÍDA
vector<bool3S> out_circ
  // Saídas do circuito

// VARIÁVEIS LOCAIS:
bool tudo_def, alguma_def
vector<bool3S> in_port
  // Entradas de uma porta

// INICIALIZAÇÃO DAS PORTAS
Para i de 0 a Num_ports-1
| out_port_ports[i] ← UNDEF
Fim Para

Repita
| tudo_def ← true;
| alguma_def ← false;
|
| Para i de 0 a Num_portas-1
| | Se (out_port_ports[i] == UNDEF)
| | | // Ajusta tamanho de in_port
| | | // igual ao num de entradas
| | | // da porta a ser simulada
| | | ajusta_tamanho(in_port,
| | |                Nin_ports[i])
| | |
| | | Para j de 0 a Nin_ports[i]-1
| | | | // De onde vem a entrada?
| | | | id ← id_in[j]_ports[i];
| | | | // Obtém valor da entrada
| | | | Se (id>0)
| | | | | // De outra porta
| | | | | in_port[j] ←
| | | | |   out_port_ports[id-1]
| | | | Caso contrário
| | | | | // De entrada do circuito
| | | | | in_port[j] ←
| | | | |   in_circ[-id-1]
| | | | Fim Se
| | | Fim Para
| | |
| | | // Simula a porta
| | | simular_ports[i](in_port)
| | |
| | | Se (out_port_ports[i] == UNDEF)
| | | | tudo_def ← false
| | | Caso contrário
| | | | alguma_def ← true
| | | Fim Se
| | |
| | Fim Se
| Fim Para
Enquanto (!tudo_def && alguma_def)

// DETERMINAÇÃO DAS SAÍDAS
Para j de 0 a Num_out-1
| // De onde vem a saída?
| id ← id_out[j];
| // Obtem valor da saída
| Se (id>0)
| | // De uma porta
| | out_circ[j] ←
| |   out_port_ports[id-1]
| Caso contrário
| | // De entrada do circuito
| | out_circ [j] ←
| |   in_circ[-id-1]
| Fim Se
Fim Para
```

SIMULAR PORTA:

Os operadores são associativos:

A AND B AND C = (A AND B) AND C
= A AND (B AND C)
A OR B OR C = (A OR B) OR C
= A OR (B OR C)
A XOR B XOR C = (A XOR B) XOR C
= A XOR (B XOR C)

Portanto, para simular uma porta com mais de 2 entradas, basta realizar a operação lógica entre a entrada de cada porta e o resultado da operação lógica entre todas as entradas anteriores.

```
// ENTRADA:
vector<bool3S> in_port
 // Entradas da porta

// SAÍDA
bool3S out_port
  // Saída da porta

// PORTA NOT
out_port ← NOT(in_port[0])

// PORTA AND
out_port ← in_port[0]
Para i de 1 a Num_input_port-1
| out_port ← (out_port AND
|             in_port[i])
Fim Para

// PORTA NOT XOR
out_port ← in_port[0]
Para i de 1 a Num_input_port-1
| out_port ← (out_port XOR
|             in_port[i])
Fim Para
out_port ← NOT(out_port)
```

GERAR TABELA VERDADE:

```
// TIPO DE DADO bool3S
bool3S: {UNDEF, FALSE, TRUE}
bool3S++:
  UNDEF → FALSE → TRUE → UNDEF → …

// VARIÁVEIS LOCAIS:
vector<bool3S> in_circ
  // Entradas do circuito

Para i de 0 a Num_input_circ-1
| in_circ[i] ← UNDEF
Fim Para

// GERAÇÃO DA TABELA
Repita
| simular_circuito(in_circ)
| exibir_entradas_saidas()
| // Qual input incrementar?
| i ← Num_input_circ-1
| Enquanto (i >= 0 &&
|           in_circ[i] == TRUE)
| | in_circ[i]++ // Se torna UNDEF
| | i--
| Fim Enquanto
| // Incrementa a input escolhida
| Se (i >= 0)
| | in_circ[i]++
| Fim Se
Enquanto (i >= 0)
```

SUGESTÃO DE DESENVOLVIMENTO

1) Implemente todas as funcionalidades das portas. Faça um programa de teste (veja sugestão `teste1.cpp` no SIGAA) que:
   a) Utilize construtores com e sem parâmetros, válidos e inválidos, e use funções de consulta (`getNumInputs`, etc.) para testar se as portas foram criadas corretamente.
   b) Teste se a função `simular` só aceita vetor de `bool3S` com a dimensão correta.
   c) Verifique se a simulação está correta para todas as combinações de entrada.
   d) Crie objetos dinâmicos e teste se os métodos virtuais (`getName`, `simular`, etc.) exibem comportamento polimórfico e correto.

2) Implemente as obrigatoriedades da classe `Circuito` (construtores, destrutor, operadores de atribuição) e as funções `clear`, `resize` e `setPort`. Faça um programa de teste (veja sugestão `teste2.cpp`) que:
   a) Teste o construtor default.
   b) Teste as funções `resize` e `setPort` com parâmetros válidos e inválidos.
   c) Teste o construtor por cópia, verificando se o novo objeto e o antigo têm memórias dinâmicas independentes (alterar um deles não modifica o outro).
   d) Teste o construtor por movimento, verificando se o novo objeto tem memória válida (pode ser alterada).
   e) Teste os operadores de atribuição por cópia e por movimento.

3) Implemente as funções `digitar`, `ler`, `imprimir` e `salvar` da classe `Circuito`. Faça um programa de teste (veja sugestão `teste3.cpp` no SIGAA) que:
   a) Permita digitar um circuito, informando valores inválidos (caso em que deve solicitar nova digitação) e válidos.
   b) Imprima circuitos em tela e salve em arquivo, verificando os resultados.
   c) Leia diversos arquivos válidos e inválidos (podem ser os que são fornecidos no SIGAA) e permita testar se a função `ler` detectou corretamente eventuais erros.

4) Implemente s função `simular` da classe `Circuito`. Teste com o programa principal da avaliação.
   a) Confira os resultados da simulação, utilizando, por exemplo, a avaliação do período letivo anterior fornecida no SIGAA.