Universidade Federal do Rio Grande do Norte
Deptº de Informática e Matemática Aplicada – DIMAp

Linguagem de Programação I • DIM0120
◁ Projeto de Programação #2 ▷

Programas BARES (Basic ARithmetic Expression Evaluator based on Stacks)

25 de abril de 2018

Apresentação

O objetivo deste exercício de programação é utilizar as estrutura de dados *pilha* e *fila* no contexto de uma aplicação real. A aplicação a ser desenvolvida é um avaliador de expressões aritméticas simples.

Sumário

1 Introdução

O programa **BARES** (*Basic ARithmetic Expression Evaluator based on Stacks*) deve ser capaz de receber expressões aritméticas simples, formadas por:-

- *constantes numéricas inteiras* (-32.768 a 32.767);
- *operadores* ($+, -, /, *, \hat{}, \%$), com precedência descrita em Tabela 1; e
- *parênteses*.

Precedência	*Operadores*	*Associação*	*Descrição*
1	`^`	$\longleftarrow$	Potenciação ou Exponenciação
2	`* / %`	$\longrightarrow$	Multiplicação, divisão, resto da divisão inteira
3	`+ -`	$\longrightarrow$	Adição, subtração

Tabela 1: Precedência e ordem de associação de operadores em expressões *bares*.

Segue abaixo exemplos de expressões válidas para o *bares*:-

- `35 - 3 * (-2 + 5)^2`
- `54 / 3 ^ (12%5) * 2`
- `((2-3)*10 - (2^3*5))`
- `---3 + 4`

O fim de linha ('\n') será o indicador de fim de expressão, ou seja, o programa deverá receber uma expressão por linha de entrada de dados. Note que espaços em branco (código ascii 32) e tabulações (código ascii 9) podem aparecer antes da expressão, entre os termos da expressão ou após a expressão e devem ser ignorados pelo programa.

2 A Tarefa

Sua tarefa consiste em elaborar um programa em C++ denominado `bares.cpp` que deverá receber via arquivo uma ou mais expressões, uma por linha. O programa deverá, então, avaliar cada expressão e imprimir seu respectivo resultado na saída padrão, `std::cout`, ou em um arquivo texto de resultados informado pelo usuário. A forma geral de execução do programa é

```
$.\bares arquivo_entrada [arquivo_saida]
```

se você decidir trabalhar com abertura de arquivos com `stream`, ou

```
$.\bares < arquivo_entrada > [arquivo_saida]
```

se seu programa trabalhar com `std::cin` (entrada) e `std::cout` (saída).

Por exemplo, a resposta que o programa deveria oferecer para as expressões exemplo da Seção 1 seria: 8, 12 e -50, e 1 (uma resposta por linha). Os detalhes sobre tratamento de erros e formatos de saída do programa estão descritos na Seção 4.

3 Solucionando o Problema

De maneira básica, o problema que este trabalho tenta resolver é o seguinte. Considerando a expressão $2 + 3 \times 6$, qual o seu resultado?

$$\underbrace{\underbrace{2 + 3}_{5} \times 6}_{30} \quad ou \quad \underbrace{2 + \underbrace{3 \times 6}_{18}}_{20}$$

Sabemos que o correto é 20, visto que a multiplicação tem prioridade maior do que a adição. Por isso executamos a multiplicação primeiro e depois a adição.

No exemplo abaixo temos um expressão formada por operadores com mesma prioridade mas com associação da *direita para a esquerda*, ao invés do tradicional *esquerda para direita*. O resultado correto da expressão, portanto, é 256.

$$\underbrace{\underbrace{2 \hat{} 2}_{4} \hat{} 3}_{64} \quad ou \quad \underbrace{2 \hat{} \underbrace{2 \hat{} 3}_{8}}_{256}$$

Precisamos sistematicamente resolver estas ambiguidades por meio de um algoritmo. Como conseguir isso?

A estratégia de solução recomendada para este trabalho envolve mudança automatizada na forma de representar uma expressão aritmética. Esta mudança, descrita nas próximas seções, facilita o processamento (avaliação) de uma expressão na medida que elimina a ambiguidade inerente à representação infixa.

3.1 Notações de Expressões

Uma expressão para somar A e B pode ser descrita como "$A + B$". Essa representação é denominada de forma **infixa** da operação *binária* de soma. Além dessa existem outras duas representações, a **prefixa**, "$+\,A\,B$", e a **posfixa**, "$A\,B\,+$". Na notação prefixa[1] o operador binário (no caso o '+') é introduzido antes de seus dois operandos (A e B). Já na notação posfixa[2] o operador binário aparece logo após seus dois operandos.

As regras que devem ser consideradas durante o processo de conversão são: i) as operações com a precedência mais alta são convertidas em primeiro lugar (quando existe ambiguidade); ii) as operações com mesma predecedência são convertidas na ordem de chegada, a não ser que seja com associação da direta para esquerda, e; iii) uma expressão convertida para a forma posfixa deve ser tratada como se fosse um único operando. Veja na Tabela 2 alguns exemplos de conversão da forma infixa para a posfixa.

[1] Também conhecido como Notação Polonesa.
[2] Também conhecido como Notação Polonesa Reversa.

Forma Infixa	*Forma Posfixa*
$A + B$	$AB+$
$A + B - C$	$AB + C-$
$A + B * C - D$	$ABC * +D-$
$(A + B) * (C - D)$	$AB + CD - *$
$A\hat{}B * C - D + E/F/(G + H)$	$AB\hat{}C * D - EF/GH + /+$
$((A + B) * C - (D - E)) \hat{}(F + G)$	$AB + C * DE - -FG+\hat{}$
$A - B/(C * D \hat{}E)$	$ABCDE \hat{} */-$

Tabela 2: Exemplos de equivalências entre forma infixa e posfixa de expressões.

3.2 "Tokenização" e Parsing

Para realizar a conversão entre representações de expressões aritméticas, é preciso, primeiramente, separar seus componentes ou *tokens*. Este processo é denominado de *tokenização*

Assumindo que uma expressão de entrada é fornecida por meio de uma cadeia de caracteres (por exemplo, `std::string`), a *tokenização* consiste em percorrer esta cadeia, caractere por caractere, separando seu termos em uma lista de *tokens*. Neste processo, caracteres em branco (código ascii 32) ou tabulações (código ascii 9) são ignorados e o fim de linha ('\n') indica o fim da expressão.

No contexto deste trabalho, um *token* pode ser um *operando* (constantes inteiras), um *operador* (uma das operações aritméticas válidas para o trabalho) ou um delimitador de escopo (parênteses). Por exemplo, se a entrada for a expressão "$(2 + 3) * 10/ - 5$", a *tokenização* deve produzir uma lista sequencial de *tokens* da seguinte forma:

$$\{\ \text{“(”, “2”, “+”, “3”, “)”, “*”, “10”, “/”, “}-5\text{”}\ \}.$$

Muitas vezes o processo de *tokenização* é feito em conjunto com o processo de *parsing*. *Parsing* consiste em fazer uma *análise sintática* da expressão composta por *tokens* para saber se ela obedece a sintaxe que define uma expressão válida.

Durante o *parsing* vários error de formação de expressão devem ser detectados e sinalizados para o usuário. Existem diversas técnicas para realizar *parsing* de uma maneira eficiente. Neste trabalho sugere-se que seja adotada o uso de uma **gramática EBNF** (*Extended Backus-Naur Form*) para definir as expressões e um **analisador sintático descendente recursivo** para validar as expressões.

Uma gramática EBNF descreve uma linguagem formal por meio de notações especiais (ver Tabela 3). No contexto deste trabalho, estamos interessados em descrever uma expressão aritmética simples. Uma gramática EBNF é formado por **símbolos terminais** e **regras de produção** não terminais, que são restrições que determinam como os símbolos terminais podem ser combinados para gerar uma sequência legal. No caso do *bares*, os símbolos terminais serão dígitos e os caracteres das operações aritméticas.

A gramática para expressões *bares* é a seguinte:

Uso	*Notação*
definição	:=
concatenação	,
terminação	;
alternativa	|
opcional	**[**...**]**
repetição	{...}
agrupamento	**(**...**)**
símbolo terminal	"..." **ou** '...'
exceção	-
sequência especial	**?**...**?**

Tabela 3: Notação usada para descrever uma gramática EBNF.

Gramática EBNF para expressões bares

```
<expr>           := <term>,{ ("+"|"-"|"*"|"/"|"%"|"^"),<term> };
<term>           := "(",<expr>,")" | <integer>;
<integer>        := 0 | ["-"],<natural_number>;
<natural_number> := <digit_excl_zero>,{<digit>};
<digit_excl_zero> := "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
<digit>          := "0"| <digit_excl_zero>;
```

A regra de produção para `<natural_number>`, por exemplo, diz que uma sequência correspondente a um número natural *válido* deve ser iniciada por um dígito diferente de zero, podendo ser seguindo ou não de mais `<digits>`.

Uma vez definido a gramática, podemos utilizar um *analisador sintático descendente recursivo* para percorrer a cadeia de entrada, caractere por caractere, tentando identificar o termo principal que é uma `<expr>`. À medida que o parser caminha na cadeia original, ele valida seus componentes elementares, como operadores e números, a sua associação sintática, e pode separar seus componentes em *tokens*. Desta forma, qualquer erro sintático que for detectado pode ser prontamente identificado, inclusive com a posição dentro da cadeia de caracteres em que o problema foi encontrado.

Desta forma, a saída do processo de *parsing* seria uma lista (fila) de *tokens* que corresponde a uma expressão infixa **sintaticamente válida**. Como consequência, a fase seguinte de conversão entre notações fica bem mais fácil de ser realizada, já que ela recebe como entrada expressões bem formadas, sem espaços em branco e com *tokens* classificados (operando, operador, ou escopo) e separados.

3.3 Transformações Entre Notações

Como mencionado anteriormente, a estratégia de solução sugerida envolve a transformação da expressão original do formato *infixo*, o qual é mais natural para o usuário fornecer uma entrada,

para o formato *posfixo*, o qual é mais "natural" para um processamento sem ambiguidades. A transformação de um formato para outro apresenta duas vantagens:-

1. A expressão no formato posfixo não necessita da presença de delimitadores (parênteses) por ser uma representação *não-ambígua*;
2. O algoritmo para avaliar uma expressão posfixa é mais simples do que um algoritmo para avaliar uma expressão infixa.

3.4 Convertendo Expressões: de infixa para posfixa

Para realizar a avaliação da expressão (descrito na Seção 3.5), faz-se necessário, primeiramente, converter a expressão de infixa para posfixa. Essa conversão deve ser feita de tal forma a lidar de forma correta com, digamos, os casos "$A + B * C$" e "$(A + B) * C$"—produzindo, respectivamente, as expressões "$ABC * +$" e "$AB + C*$".

Ao analisar os casos acima percebe-se que o algoritmo de conversão deve possuir algum tipo de mecanismo para armazenar os operadores temporariamente de tal forma que a regra de precedência de operadores seja respeitada. Esse mecanismo de armazenamento também será uma estrutura de dados do tipo *pilha*.

O Algoritmo 1 apresenta uma forma de converter uma expressão (sem parênteses[3]) no formato infixo para o posfixo. Para tanto precisamos de uma função denominada `higher_precedence(op1,op2)` — onde `op1` e `op2` são operadores — que retorna `true` se `op1` tiver precedência sobre `op2`. Se ambos operadores tiverem a *mesma precedência*, a função deve retornar `true` se `op1` for operador de associação da esquerda para direita, ou `false` caso `op1` seja um operador de associação direita para esquerda (por exemplo, exponenciação). Por exemplo, `higher_precedence('*','+')` e `higher_precedence('-','+')` retornam `true`, enquanto `higher_precedence('+','*')` e `higher_precedence('^','^')` retornam `false`.

Entendendo a necessidade da pilha

Perceba no Algoritmo 1 que a pilha é utilizada para "segurar" operações binárias que ainda não podem ser efetivadas para a saída porque não se sabe qual é o segundo operando. Por exemplo, considerando a expressão "$A * B + C$", temos que o "A" é enviado para a saída e o "$*$" é empilhado, porque não se sabe qual é o segundo operando da multiplicação. O segundo operando poderia ser o "B" (de fato é), mas na verdade não é possível ter certeza disso apenas observando o próximo caractere que é o "B", pois não sabemos se existe alguma outra operação de maior precedência que a multiplicação após o "B". Por exemplo, se a expressão fosse "$A * B\hat{}C$", o segundo operando da multiplicação não seria o "B", mas sim o resultado da exponenciação "$B\hat{}C$".

É por este motivo que a operação de precedência é fundamental, pois ela é quem determina se um operador deve permanecer na pilha ou se ele pode ser "liberado" porque foi encontrado,

[3]Você deve adaptar o algoritmo para tratar o uso de parênteses.

na sequência, um outro operador de menor precedência.

Algoritmo 1 Conversão de expressão no formato infixo para posfixo.

Entrada: Fila de '`Símbolo`' representando uma expressão no formato infixo.
Saída: Fila de '`Símbolo`' representando uma expressão no formato posfixa equivalente.

```
1:  função Infx2Posfx(fila no formato infixo): fila no formato posfixo
2:      enquanto não chegar ao fim da fila de entrada faça
3:          remover símbolo da fila de entrada em symb
4:          se symb for operando então
5:              enviar symb diretamente para fila de saída
6:          senão
7:              enquanto Pilha não estiver vazia e símbolo do topo (topSymb) ≥ symb faça
8:                  remover topSym e enviar para fila de saída
9:              Empilhar symb      # depois que retirar operadores de precedência ≥, inserir symb
        # descarregar operadores remanescentes da pilha
10:     enquanto Pilha não estiver vazia faça
11:         remover símbolo da pilha e enviar para fila de saída
12:     retorna fila de saída na forma posfixa
```

3.5 Avaliando Uma Expressão Posfixa

Para realizar a avaliação de uma expressão na forma posfixa pode-se utilizar uma estrutura de dados do tipo *pilha*. Cada vez que um **operando** (i.e. uma constante inteira) é encontrado na expressão o mesmo deve ser introduzido na pilha. Quando um **operador** (i.e. $+$, $\hat{}$, $*$, etc.) é encontrado na expressão, os dois elementos no topo da pilha são seus operandos. Portanto, devemos retirar esses dois elementos da pilha, realizar a operação indicada pelo operador[4] e, a seguir, (re)introduzir o resultado de volta na pilha, tornando-o disponível para uso como operando do próximo operador. Confira o Algoritmo 2.

[4] Cuidado com a ordem dos operandos, pois a pilha tem comportamento *LIFO* (*Last In, First Out*).

Algoritmo 2 Avaliação de expressão no formato posfixo.

Entrada: Fila de 'Símbolo' representando uma expressão no formato posfixo.
Saída: Resultado da expressão avaliada.

```
1:  função AvalPosfixa(FPosfixa: FilaDeSímbolo): inteiro
2:      var symb: Símbolo                          # símbolo atual a ser analisado
3:      var OPn: PilhaDeInteiro                    # pilha de operandos
4:      var opnd1, opnd2: Operando                 # operandos auxiliares
5:      var resultado: inteiro                     # recebe o resultado de operação
6:      enquanto não FPosfixa.isEmpty() faça       # não chegar ao fim da fila...
7:          FPosfixa.dequeue(symb)
8:          se symb.ehOperando() então             # é operando?
9:              OPn.push(symb.getValue)            # empilha operandos
10:         senão
11:             OPn.pop(opnd1)                     # recupera 1º operando
12:             OPn.pop(opnd2)                     # recupera 2º operando
13:             resultado ← Aplicar symb à opnd1 e opnd2   # um ''caso'' para cada operador
14:             OPn.push(resultado)    # armazenar resultado; fila pode estar em processamento
15:     OPn.pop(resultado)                         # recuperar o valor final da pilha e...
16:     retorna resultado                          # ...retorna o valor inteiro do símbolo
```

3.6 Visão Geral do Processo

A Figura 1 apresenta uma visão geral do processo de avaliação de uma expressão aritmética, com cada componente anteriormente apresentado.

Note que a avaliação pode finalizar com erro em duas ocasiões: erro sintático e erro de execução. Os dois tipos de erros serão apresentados a seguir, na Seção 4. Por outro lado, se tudo correr bem, o resultdo final é o valor da expressão corretamente avaliada.

4 Tratamento de Erros

Note que em caso de haver algum problema com a expressão (e.g. escopo aberto, operador inválido, falta de operando, etc.) o programa deverá indicar qual o erro ocorrido e em que posição (coluna) da expressão.

De uma forma geral é possível separar os erros em categorias: **erros sintáticos** (e.g. números em ponto flutuante ou fora da faixa dos inteiros, falta de parênteses, símbolos não reconhecido, etc.) detectados durante a fase de *parsing* e os **erros em tempo de execução**, ou seja, quando a expressão no formato posfixa está sendo avaliada.

O conjunto de erros que devem ser tratados são os seguintes[5]:

1. **Integer constant out of range beginning at column (n)!**: O operando que se inicia

[5] O símbolo '␣' é utilizado para representar espaço em branco, não aparecendo de fato na expressão.

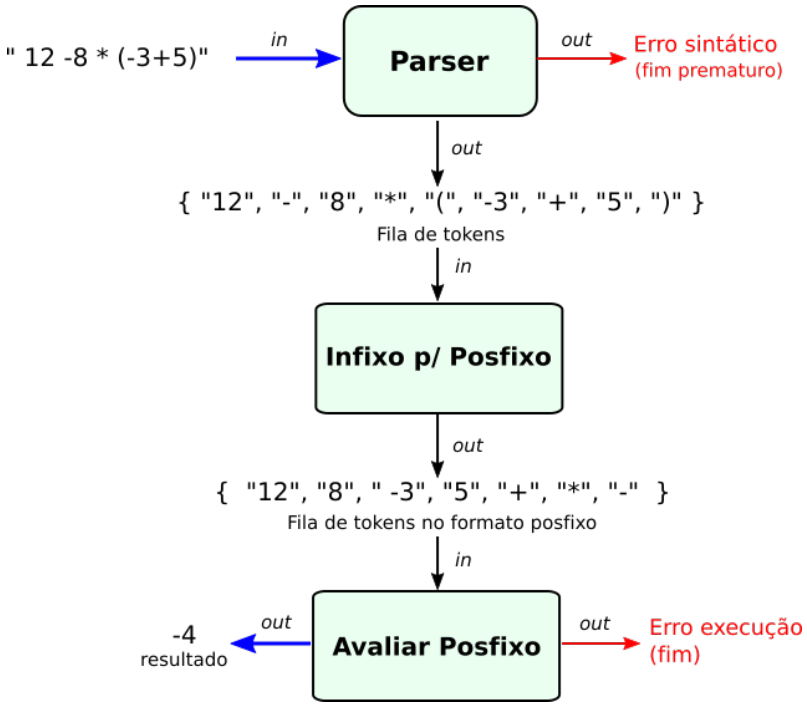

Figura 1: Visão geral do encadeamento de processos, com suas respectivas entradas e saídas possíveis. Na sequência de passos, entramos com uma string representando uma expressão; em seguida o Parser tokeniza a string e realiza a análise sintática, resultando em erro ou em uma fila de tokens; no passo seguinte, a fila de tokens é convertida do formato infixo para o posfixo; o passo final consiste em avaliar a expressão posfixa, o que pode gerar em erro de execução (por exemplo divisão por zero) ou resultar em um valor inteiro correspondente à avaliação da expressão aritmética de entrada.

na coluna n está fora da faixa permitida.
Ex.: $1000000 - 2$, coluna 1.

2. **Missing <term> at column (n)!**: A partir da coluna n está faltando o resto da expressão .
   Ex.: $2 +$, coluna 4.
3. **Extraneous symbol after valid expression found at column (n)!**: Existe um símbolo qualquer (válido ou não) que foi encontrado depois que uma expressão completa foi validada, na coluna n.
   Ex.: $2 = 3$, coluna 3; ou $2 + 3 4$, coluna 7.
4. **Ill formed integer at column (n)!**: Contante inteira iniciada na coluna n possui símbolo inválido em sua composição.
   Ex.: -5, coluna 1; ou $!3$, coluna 1; ou $)2 + 4$, coluna 1.
5. **Missing closing ")"at column (n)!**: Está faltando um parêntese de fechamento ')' para um parêntese de abertura '(' correspondente, na coluna n.
   Ex.: $((2 \% 3) * 8$, coluna 13.

6. **Unexpected end of expression at column (n)!**: Caso receba uma linha contendo apenas espaços, cujo final é encontrado na coluna n.
   Ex.: ␣␣␣, coluna 4 ou; ␣␣(, coluna 4 ou;
7. **Division by zero!**: Houve divisão cujo quociente é zero.
   Ex.: $3/(1-1)$; ou $10/(3*3\hat{}-2)$. Nestes casos não é preciso informar a coluna.
8. **Numeric overflow error!**: Acontece quando uma operação dentro da expressão ou a expressao inteira estoura o limite das constantes numéricas definidos na Seção 1.
   Ex.: $20*20000$. Nestes casos não é preciso informar a coluna.

Seu programa deve apresentar apenas o primeiro erro encontrado em uma expressão; isto é, os demais erros para a *mesma expressão* (se existir) devem ser ignorados. Contundo, entenda que você deve processar **todo** o arquivo de entrada, ou seja, se houver um erro em uma expressão você deve indicá-lo e seguir processando as demais expressões presentes no arquivo.

Para facilitar a correção automática, a saída do seu programa—seja em arquivo ascii ou na saída padrão `std::cout`—deve seguir **exatamente** as mensagens expostas acima, seguidas da coluna onde o erro foi encontrado, se for o caso.

Em linhas gerais para cada expressão de entrada (1 por linha) é possível gerar no arquivo de saída ou na saída padrão:-

- Uma linha com *apenas* com o resultado da expressão, se a expressão estiver sintaticamente correta; ou
- Uma linha contendo a indicação de erro e coluna correspondente (se for o caso).

As mensagens de erro devem ser **idênticas** às citadas acima, incluindo letras maiúsculas, minúsculas e espaços em branco, de maneira que o comando `diff` identifique apenas as linhas erradas entre a sua resposta e um arquivo com o gabarito.

Confira a seguir um exemplo de entrada de dados (lado esquerdo) e a saída correspondente (lado direito) que o programa deve gerar no arquivo de saída indicado via linha de comando.

```
10000000 - 2                    Integer constant out of range beginning at column (1)!
2+                              Missing <term> at column (3)!
3 * d                           Ill formed integer at column (5)!
2 = 3                           Extraneous symbol after valid expression found at column (3)!
2.3 + 4                         Extraneous symbol after valid expression found at column (2)!
2 * 3 4                         Extraneous symbol after valid expression found at column (7)!
2 ** 3                          Ill formed integer at column (4)!
%5 * 10                         Ill formed integer at column (1)!
*5 * 10                         Ill formed integer at column (1)!
(2+3)*/(1-4)                    Ill formed integer at column (7)!
(-3*4)(10*5)                    Extraneous symbol after valid expression found at column (7)!
2 - 4)                          Extraneous symbol after valid expression found at column (6)!
2) - 4                          Extraneous symbol after valid expression found at column (2)!
)2 - 4                          Ill formed integer at column (1)!
((2%3) * 8                      Missing closing ")" at column (11)!
 3/(1-1)                        Division by zero!
10/(3*3^-2)                     Division by zero!
20*20000                        Numeric overflow error!
```

```
   25    /   5 + 4 *    8          37
(2+3) * 8                          40
5 % 2 ^4                           5
 (5 % 3) ^4                        16
 -----3                            -3
```

5 Avaliação do Programa

Para a implementação deste projeto *é recomendado* a utilização das classes pilha, fila e lista sequencial que foram apresentadas em sala de aula.

O programa completo deverá ser entregue sem erros de compilação, testado e totalmente documentado. O projeto será avaliado sob os seguintes critérios:-

- Lê expressões de um arquivo ascii ou `std::cin` e cria corretamente uma lista de *tokens*, de acordo com a gramática da Seção 3.2 (20%);
- Converte corretamente expressões do formato infixo para posfixo (15%);
- Trata corretamente o uso de parênteses e '-' unário (5%);
- Avalia corretamente expressão no formato posfixo (15%);
- Detecta corretamente o conjunto de erros solicitados (20%);
- Gera a saída conforma solicitado (15%); e
- Código é organizado em classes (10%).
- **Extra**: se forem utilizados filas, pilhas e vector desenvolvidos para o projeto (10%).

A pontuação acima não é definitiva e imutável. Ela serve apenas como um guia de como o trabalho será avaliado em linhas gerais. É possível a realização de ajustes nas pontuações indicadas visando adequar a pontuação ao nível de dificuldade dos itens solicitados.

Os itens abaixo correspondem à descontos, ou seja, pontos que podem ser retirados da pontuação total obtida com os itens anteriores:-

- Presença de erros de compilação e/ou execução (até -20%)
- Falta de documentação do programa com Doxygen (até -10%)
- Vazamento de memória (até -10%)
- Falta de um arquivo texto `README` contendo, entre outras coisas, identificação da dupla de desenvolvedores; instruções de como compilar e executar o programa; lista dos erros que o programa trata; e limitações e/ou problemas que o programa possui/apresenta, se for o caso (até -20%).

Boas práticas de programação

Recomenda-se fortemente o uso das seguintes ferramentas:-

- Doxygen: para a documentação de código e das classes;
- Git: para o controle de versões e desenvolvimento colaborativo;
- Valgrind: para verificação de vazamento de memória;

- gdb: para depuração do código; e
- Makefile: para gerenciar o processo de compilação do projeto.
- Markdown: para escrever o arquivo `README.md`.

Recomenda-se também que sejam realizados testes unitários nas suas classes de maneira a garantir que elas foram implementadas corretamente. Procure organizar seu código em várias pastas, conforme vários exemplos apresentados em sala de aula, com pastas como `src` (arquivos `.cpp`), `include` (arquivos `.h`), `bin` (arquivos `.o` e executável) e `data` (arquivos de entrada e saída de dados).

6 Autoria e Política de Colaboração

O trabalho pode ser realizado **individualmente** ou em **duplas**, sendo que no último caso é importante, dentro do possível, dividir as tarefas igualmente entre os componentes. A divisão de trabalho deve estar refletida no *log* de *commits* associados ao repositório do projeto.

Qualquer equipe pode ser convocada para uma entrevista. O objetivo da entrevista é duplo: confirmar a autoria do trabalho e determinar a contribuição real de cada componente em relação ao trabalho. Durante a entrevista os membros da equipe devem ser capazes de explicar, com desenvoltura, qualquer trecho do trabalho, mesmo que o código tenha sido desenvolvido pelo outro membro da equipe. Portanto, é possível que, após a entrevista, ocorra redução da nota geral do trabalho e/ou ajustes nas notas individuais, de maneira a refletir a verdadeira contribuição de cada membro, conforme determinado na entrevista.

O trabalho em cooperação entre alunos da turma é estimulado. É aceitável a discussão de ideias e estratégias. Note, contudo, que esta interação **não** deve ser entendida como permissão para utilização de código ou parte de código de outras equipes, o que pode caracterizar a situação de plágio. Em resumo, tenha o cuidado de escrever seus próprios programas.

Trabalhos plagiados receberão nota **zero** automaticamente, independente de quem seja o verdadeiro autor dos trabalhos infratores. Fazer uso de qualquer assistência sem reconhecer os créditos apropriados é considerado **plagiarismo**. Quando submeter seu trabalho, forneça a citação e reconhecimentos necessários. Isso pode ser feito pontualmente nos comentários no início do código, ou, de maneira mais abrangente, no arquivo texto `README`. Além disso, no caso de receber assistência, certifique-se de que ela lhe é dada de maneira genérica, ou seja, de forma que não envolva alguém tendo que escrever código por você.

7 Entrega

Você deve submeter um único arquivo com a compactação da pasta do seu projeto. Indique também o link GitLab para o seu projeto. O arquivo compactado deve ser enviado **apenas** através da opção Tarefas da turma Virtual do Sigaa, em data divulgada no sistema.

◄ FIM ►