Calculadora Polonesa Reversa

Guilherme Mendel de Almeida Nascimento

Universidade de Brasília, DF, Brasil

Abstract

Esse relatório se refere ao primeiro trabalho da matéria Estrutura de Dados (prof. Marcos Caetano) do primeiro semestre de 2018, UnB. Nele, será descrita a arquitetura do sistema desenvolvido e a explicação de todas as funcionalidades do programa, além de mencionar como foram feitos os *smoke tests* e conter um link para a documentação gerada pelo doxygen do programa.

Keywords: Calculadora, Notação posfixa, Notação polonesa reversa

Doxygen do Arquivo

1. Introdução

Em 1929, o matemático polonês Jan Lukasiewicz demonstrou que os parênteses não são necessários para se representar a ordem das operações numa expressão, ao propor uma nova notação de expressão: a notação Polonesa. Ao serem representados à direita dos operandos, em vez de entre eles, os operadores agem como delimitadores entre as diferentes operações da expressão. No entanto, não foi até o ano de 1950 que o cientista da computação e filósofo australiano Charles Hamblin introduziu a notação Polonesa Reversa, em que os operandos precedem as operações. Essa nova maneira de representar expressões se mostrou muito compatível com a natureza sequencial dos algoritmos computacionais — sobretudo quando da sua implementação por meio de listas LIFO (*Last In, First Out*), ou pilhas.

Nesse trabalho, a proposta era desenvolver um programa capaz de tanto ler uma expressão infixa, transforma-la em posfixa e encontrar sua solução, quanto executar, pelo modo *calculadora*, operações em tempo real, apresentando uma pilha na tela, na qual o usuário pode inserir novos números e aplicar operações.

2. Arquitetura do Sistema Desenvolvido

Primeiramente, é importante ressaltar que o foco da treinamento com esse trabalho foi na área de alocação dinâmica de memória e na implementação de listas LIFO, ou pilhas. Essas pilhas seguem um padrão tal que o último elemento a ela adicionado será o primeiro elemento a ser removido, tal como uma pilha de pratos.

2.1. Das Pilhas

No arquivo *pile.c* se encontram as definições das diversas funções referentes à implementação das pilhas.

Nessa implementação, como pode ser observado a partir do *doxygen* do programa, foram desenvolvidas duas structs — a Pile e a Pelm. A struct Pile (Pilha) serve meramente como uma estrutura de referência, em que são armazenados o endereço do último elemento da pilha (o último deve ser entendido pelo último a ser empilhado) e a quantidade de elementos nela. Já a struct Pelm (Pile Element – Elemento da Pilha) vai armazenar um valor *double* e o endereço do próximo elemento na pilha. Ao serem inicializadas pelas suas respectivas funções *genPile* e *genPelm* (*gen* de generate – gerar), seus ponteiros recebem *NULL*.

```
--(pile.c.pdisplay)--> Atual configuracao da pilha:
        1: 5.00
        2: 7.00
        3: 46.00
        4: 2.00
        5: 1.00
```

Figure 1: Exemplo de pilha

Sempre que um elemento é alocado, o endereço armazenado na struct Pile correspondente é atualizado, assim como seu valor quantidade, e o novo elemento vai apontar para o antigo último elemento.

Como as struct Pile's não são alocadas dinamicamente, não é necessário liberar memória para ela na destituição de uma pilha, porém seus elementos (struct Pelm) são dinâmicos, o que levou à implementação da função *poust(Pile *p)* (oust pile – destituir pilha), que libera o espaço alocado elemento por elemento da pilha alvo, e deve ser chamada sempre que uma pilha não vai mais ser utilizada.

As pilhas servirão de grande importância no desenvolvimento do restante do programa, e tomarão participação nos dois modos da calculadora.

2.2. Da Main

Ao longo da implementação do programa, a fim de exercitar o uso de arquivos headers e objetos, foi definido que nenhuma função seria delcarada no mesmo arquivo que a main. Por esse motivo, o arquivo *main.c* somente tem a função de iniciar o programa e pedir ao usuário que escolha entre o modo "Resolução de expressão" e "Calculadora".

```
Calculadora Polonesa
Bem vindo!

Selecione um modo:
        1 - Resolucao de Expressao
        2 - Calculadora
        0 - Sair
-> _
```

Figure 2: Menu principal

2.3. Do Modo Resolução de Expressão

Nesse modo, o programa vai receber uma expressão matemática no formato infixo, formada por operandos reais e operações de soma, subtração, divisão e multiplicação, a converter para o formato posfixo e só então solucionala. Ele é capaz de detectar diversos erros, como múltiplas vírgulas, operadores utilizados incorretamente e até mesmo detecta divisões por zero. É importante mencionar que o programa aceita o uso tanto do carácter '.' quanto ',' para representar a vírgula do operando e não suporta números negativos.

Primeiramente, é solicitado do usuário uma expressão válida, que passa pela função *validate(char *exp)*, onde ela validada quanto ao uso correto dos parênteses, da sintaxe operador–operando, dos operandos em si (por exemplo, o numero 2.1.3 é inválido), entre outros. A validação dos parênteses é feita com a utilização das pilhas.

```
Modo Resolucao de Expressao

Digite uma expressao valida, ou q para voltar:
-> 2*[3-4)
A expressao fornecida possui erro(s) na sintaxe dos '('
```

Figure 3: Exemplo de expressão rejeitada

Após ser validada com sucesso, a solução da expressão é calulcada pela função *solve(char *exp)*, que é responsável não só por calcular o resultado mas também por chamar outras funções, tal qual a função *dump(Pile *n)*, que utiliza a lógica das pilhas para montar um único número *double* a partir de carácteres de números e vírgulas, que foram anteriormente empilhados na Pile 'n'.

Por fim, o resultado é impresso na tela, a menos que haja alguma divisão por 0 — caso em que o programa alerta o usuário do acontecimento, apesar de ainda assim retornar o resultado como "0.00".

```
Digite uma expressao valida, ou q para voltar:
-> 5-2*(4+3/[1+1]-7)/1.2
A expressao fornecida e valida.
A expressao no formato posfixo e: 5 2 4 3 1 1 + / + 7 - * - 1.2 /
Seu resultado e 6.67
```

Figure 4: Exemplo de expressão aceita

2.4. Do Modo Calculadora

Nesse modo, uma pilha é inicializada, e o usuário tem a opção de colocar nela quaisquer operandos ele gostar, menos o 0, naturalmente. Ao digitar uma operação básica, tal qual o +, -, * ou / — adição, subtração, multiplicação ou divisão — o programa realiza e operação no formato

$$a(o)b \tag{1}$$

onde a é o último elemento, b o penúltimo e *(o)* a operação em questão. Existem dois operadores especiais, a saber:

- **'!'** – Quando antecedido por um operador básico, indica que essa mesma operação deve ser realizada sobre o último elemento da pilha por todos os outros, sobrando na pilha após sua realização apenas o resultado.
- **'c'** – Empilha repetidamente o penúltimo elemento da pilha uma quantidade de vezes igual ao último elemento da pilha.

O funcionamento desse modo é bem intuitivo, a única parte mais complexa é validar a expressão de entrada. Porém é óbvio que se o primeiro caráctere for um número/vírgula, tambémdeverão o ser os demais. Partindo

```
Modo Calculadora

Atual configuracao da pilha:
        1: 10.00
        2: 5.65
        3: 2.40

Digite um valor valido, ou q para voltar
->_
```

Figure 5: Estruturação do modo calculadora

desse princípio, surge a função *process_Celement(char *c, double *num)* (Processar Elemento Calculadora), que faz a validação da expressão lida e, caso esta seja um número, já converte esse número em *string* para seu valor correspondente em *double*.

É interessante notar que sempre que algum elemento da pilha zera — caso o usuário faça 1-1, por exemplo — esse elemento é retirado da pilha. Dessa maneira, nunca haverá divisões, nem mesmo multiplicações, por zero.

```
Modo Calculadora
A pilha necessita ter no minimo 2 elementos para realizar essa funcao!

Atual configuracao da pilha:
        1: 2.00

Digite um valor valido, ou q para voltar
->_
```

Figure 6: Caso o usuário tente realizar operações com um número insuficiente de operandos, ele será alertado com uma mensagem de erro.

3. Smoke Test

No português, *Smoke Test* significa Teste de Fumaça, e consiste em realizar um teste prático de um circuito ou de uma aplicação pela primeira vez ou após um reparo/atualização. Comumente esse teste é realizado todo dia ao início do desenvolvimento, e uma planilha é utilizada para manter o controle dos resultados de cada teste.

No desenvolvimento deste programa, foram realizados alguns Smoke Tests a pedido do professor e também para manter a estabilidade ao longo da

implementação de novas funções e mudanças nas preexistentes. A função *dump(Pile *n)*, por exemplo, em sua primeira versão, recebia também o endereço de uma segunda pilha e não retornava nenhum valor. Ela já colocava o resultado obtido por meio da montagem dos números da pilha n direto numa segunda pilha p — daí o seu nome *dump*, no português despejar/descarregar.

3.1. Suite de Pré-testes: Pilhas

- Declaração de Pilhas

  Função genPile

  Arquivo pile.c

  Ação inicializa a quantidade da pilha com 0 e seu ponteiro com NULL.

  Testes Sucesso observado nos testes dos dias 12/05, 13/05 e 15/05.

- Declaração de Elementos

  Função genPelm

  Arquivo pile.c

  Ação aloca espaço para um elemento dinamicamente e faz seu ponteiro apontar para NULL.

  Testes Sucesso observado nos testes dos dias 12/05, 13/05 e 15/05.

- Adição de elementos às pilhas

  Função push

  Arquivo pile.c

  Ação adiciona um dado valor a uma pilha.

  Testes Sucesso observado nos testes dos dias 12/05, 13/05 e 15/05.

- Remoção de elementos das pilhas

  Função pop

  Arquivo pile.c

  Ação remove o último elemento de uma pilha e libera seu espaço.

  Testes Sucesso observado nos testes dos dias 12/05, 13/05 e 15/05.

- Desmantelamento de uma pilha

Função poust

Arquivo pile.c

Ação remove todos os elementos de uma pilha e libera seus espaços.

Testes Sucesso observado nos testes dos dias 12/05, 13/05 e 15/05.

- Receber o valor do último elemnto da pilha

  Função pfirst

  Arquivo pile.c

  Ação retorna o valor do último elemento da pilha.

  Testes Sucesso observado nos testes dos dias 13/05 e 15/05.

- Imprimir na tela a pilha

  Função pdisplay

  Arquivo pile.c

  Ação imprime na tela elemento por elemento de uma pilha.

  Testes Sucesso observado nos testes dos dias 13/05 e 15/05.

3.2. Suite de Testes 1: Menu

- Apresentação do menu principal

  Função main

  Arquivo main.c

  Ação inicia o programa e apresenta a tela de menu principal que pode ser navegado.

  Testes foi um sucesso em todas as suas execuções (Testada isoladamente em 14/05 e 19/05).

- Apresentação e funcionamento do modo Resolução de Expressão

  Função m_expression

  Arquivo calc.c

  Ação solicita do usuário uma expressão válida, a transforma no formato posfixo e a resolve. Alerta usuário em caso de erro na expressão.

Testes foi um sucesso em todas as suas execuções (Testada isoladamente em 16/05, 19/05, 21/05)

- Apresentação e funcionamento do modo Calculadora

  Função m_calculator

  Arquivo calc.c

  Ação solicita do usuário um valor válido, o empilha e realiza operações solicitadas. Avisa o usuário caso as operações não possam ser realizadas.

  Testes foi um sucesso em todas as suas execuções (Testada isoladamente em 18/05, 19/05, 21/05)

3.3. Suite de Testes 2: Funcionalidades Gerais

- Conversão de *char* para *double*

  Função char_number

  Arquivo calc.c

  Ação transforma um único carácter *char* para o seu valor correspondente em *double*, caso seja um número, mas apenas retorna seu valor na tabela ASCII caso seja uma vírgula ou ponto.

  Testes Sucesso observado nos testes dos dias 14/05 e 19/05.

- Montar um número a partir de uma pilha

  Função dump

  Arquivo calc.c

  Ação retorna o valor em *double* equivalente à montagem feita pelos números dispostos numa pilha.

  Testes Sucesso observado nos testes dos dias 14/05, 19/05 e 21/05.

3.4. Suite de Testes 3: Modo Resolução de Expressão

- Reconhecimento de um carácter

  Função tdigit

  Arquivo calc.c

Ação analisa um dado carácter e o define como operando, operador, parêntese ou outro.

Testes Sucesso observado nos dias 14/05 e 19/05.

- Validação de expressão

  Função validate

  Arquivo calc.c

  Ação verifica se a expressão possui algum carácter inválido e se ela possui a quantidade correta de parenteses de abertura e fechamento.

  Testes Sucesso observado nos dias 14/05 e 19/05.

- Conversão de notação

  Função intopost

  Arquivo calc.c

  Ação transforma uma expressao infixa para o formato posfixo.

  Testes Sucesso observado nos dias 15/05 e 19/05.

- Resolução de expressão

  Função solve

  Arquivo calc.c

  Ação encontra a solução de uma expressão no formato posfixo.

  Testes Sucesso observado nos dias 17/05 e 19/05.

- Obter uma potência de 10

  Função texp

  Arquivo calc.c

  Ação encontra a potência especificada de 10

  Testes Sucesso observado nos dias 17/05 e 19/05.

3.5. Suite de Testes 4: Modo Calculadora

- Imprimir na tela a pilha da calculadora

  Função calc_pdisplay

  Arquivo calc.c

  Ação imprime na tela elemento por elemento da pilha da calculadora.

  Testes Sucesso observado nos dias 18/05 e 21/05.

- Executar um comando sobre a pilha

  Função execute

  Arquivo calc.c

  Ação realiza o comando desejado sobre a pilha da calculadora.

  Testes Sucesso observado nos dias 20/05 e 21/05.

- Validar um comando

  Função process_Celement

  Arquivo calc.c

  Ação valida uma *string* e decide se ela é um valor a ser empilhado ou um comando a ser executado.

  Testes Sucesso observado nos dias 20/05 e 21/05.