Trabalho

Thiago Santos Matos Versão 1.0 Quarta, 25 de Outubro de 2017

Índice

Table of contents

Índice das estruturas de dados

Estruturas de dados

Lista das estruturas de dados com uma breve descrição:	
element (Estrutura de tipo elemento)	
stack (Estrutura de tipo pilha)	4

Índice dos ficheiros

Lista de ficheiros

Lista de todos os ficheiros com uma breve descrição:

calculator.c	6
expression.c	14
main.c	
stack.c	
stdtask.h (Este arquivo contém as declarações de todas as funções implementadas)	

Documentação da classe

Referência à estrutura element

Estrutura de tipo elemento. #include <stdtask.h>

Campos de Dados

- float data
- struct element * next

Descrição detalhada

Estrutura de tipo elemento.

Nesta estrutura é armazenado um dado no formato float e um ponteiro para o próximo elemento.

Documentação dos campos e atributos

float data

struct element* next

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

• stdtask.h

Referência à estrutura stack

Estrutura de tipo pilha. #include <stdtask.h>

Campos de Dados

• T_element * top

Descrição detalhada

Estrutura de tipo pilha.

Nesta estrutura é armazenado o endereço do elemento que está no topo da pilha.

Documentação dos campos e atributos

T_element* top

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

• stdtask.h

Documentação do ficheiro

Referência ao ficheiro calculator.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stdtask.h"
```

Funções

• void calc ()

Recebe operandos e operadores.

• void sum (T_stack *stack)
Soma os 2 primeiros dados da pilha.

void sumr (T_stack *stack)
 Soma todos os dados da pilha.

void sub (T_stack *stack)
 Subtrai os 2 primeiros dados da pilha.

• void **subr** (**T_stack** ***stack**)
Subtrai todos os dados da pilha.

• void mul (T_stack *stack)

Multiplica os 2 primeiros dados da pilha.

• void mulr (T_stack *stack)

Multiplica todos os dados da pilha.

• void divs (T_stack *stack)

Divide os 2 primeiros dados da pilha.

void divsr (T_stack *stack)
 Divide todos os dados da pilha.

• void copy (T_stack *stack)

Copia o primeiro elemento da pilha.

void ctod (char *S)
 Transforma vírgula em ponto.

Documentação das funções

void calc ()

Recebe operandos e operadores.

Recebe os operandos e operadores do usuário e executa as funções necessárias para solucionar as operações.

Parâmetros:

void não recebe nenhum argumento

Retorna:

```
12
13
     system("clear");
14
15
     printf("\tCalculadora\t(para sair da calculadora pressione s)\n\n");
16
17
     printstack(stack);
18
19
     printf("\n-> ");
     scanf("%s", S);
20
21
     ctod(S);
22
23
     freebuffer();
24
25
     switch (S[0]) {
  case '+':
   if(S[1] == '!') {
26
27
28
           sumr(stack);
29
         } else {
30
           sum(stack);
31
        break;
32
33
       case '-':
        if(S[1] >= '0' && S[1] <= '9') {
34
35
          F = atof(S);
36
           push(stack, F);
        } else {
  if(S[1] == '!') {
37
38
39
             subr(stack);
40
           } else {
41
             sub(stack);
42
           }
43
44
         break;
       case '*':
45
46
        if(S[1] == '!') {
47
          mulr(stack);
48
        } else {
49
          mul(stack);
50
51
        break;
      case '/':
52
        if(S[1] == '!') {
53
54
           divsr(stack);
55
        } else {
56
           divs(stack);
57
        break;
58
59
      case 'C':
         copy(stack);
60
        break;
61
       case 'c':
62
63
        copy(stack);
64
        break;
       case 'S':
65
66
        break;
67
      case 's':
68
         break;
69
       default:
70
        F = atof(S);
71
         push(stack, F);
72
         break;
73
     }
74
75
     while(S[0] != 'S' \&\& S[0] != 's') {
76
      system("clear");
77
      printf("\tCalculadora\t(para sair da calculadora pressione s)\n\n");
78
79
80
      printstack(stack);
81
       printf("\n-> ");
82
       scanf("%s", S);
8.3
84
       ctod(S);
85
86
       freebuffer();
87
88
      switch (S[0]) {
```

```
case '+':
 89
         if(S[1] == '!') {
 90
 91
             sumr(stack);
 92
           } else {
 93
             sum(stack);
 94
 9.5
          break;
 96
         case '-':
 97
          if(S[1] >= '0' && S[1] <= '9') {
 98
           F = atof(S);
99
             push(stack, F);
         } else {
100
101
            if(S[1] == '!') {
              subr(stack);
102
103
             } else {
104
               sub(stack);
105
106
         break;
107
       case '*':
108
         if(S[1] == '!') {
109
110
            mulr(stack);
111
           } else {
         mul(stack);
}
break;
112
113
114
        case '/':
115
         if(S[1] == '!') {
116
117
            divsr(stack);
118
           } else {
119
            divs(stack);
120
121
          break;
122
        case 'C':
        copy(stack);
123
124
          break;
        case 'c':
125
         copy(stack);
break;
126
127
       case 'S':
128
129
          break;
130
        case 's':
131
          break;
132
         default:
133
         F = atof(S);
134
           push(stack, F);
135
           break;
136
      }
137
     }
     printf("\n(pressione a tecla Enter para retornar ao menu)\n");
138
139
    freestack(stack);
140 }
```

void copy (T_stack * stack)

Copia o primeiro elemento da pilha.

Desempilha o primeiro e o segundo elemento da pilha, o número armazenado no primeiro elemento desempilhado dirá quantas vezes o segundo elemento deverá ser repetidamente empilhado.

Parâmetros:

T_stack* stack: pilha na qual operandos e operadores são em	pilhados
---	----------

Retorna:

```
311 {
    312    int F, i;
    313    float A;
    314    315    if(emptystack(stack)) {
        316         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter para retornar ao menu)\n");
```

```
317
           getchar();
  319 if(stack->top->next == NULL) {
320 printf("\novertine")
             printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
              getchar();
  321
            } else {
  322
          F = (int) pop(stack);

A = pop(stack);

for(i = 0; i < F; i++) {

   push(stack, A);
  323
  324
  325
  326
  327
  328
            }
  329 }
  330 }
```

void ctod (char * S)

Transforma vírgula em ponto.

Percorre a string recebida do usário transformando todos os caracteres ',' em '.'.

Parâmetros:

-		
	char*	S: armazena uma string recebida do usuário
- 1	Citai	b. armazena ama sariig recesiaa ao asaario

Retorna:

void: não retorna nenhum valor

```
333 {
334 int i;
335
336 for(i = 0; S[i] != '\0'; i++) {
337 if(S[i] == ',') {
338 S[i] = '.';
339 }
340 }
341 }
```

void divs (T_stack * stack)

Divide os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de divisão entre eles e empilha o resultado da operação na pilha.

Parâmetros:

T_stack*	stack: pilha na qual operandos e operadores são empilhados
----------	--

Retorna:

```
269
  270
       float F, A;
  271
  272 if(emptystack(stack)) {
         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
  273
para retornar ao menu) \n");
  274 getchar();
  275
        } else {
       if(stack->top->next == NULL) {
   printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
  276
  277
Enter para retornar ao menu) \n");
  278 getchar();
  279
         } else {
         F = pop(stack);
A = pop(stack);
F = F / A;
  280
  281
282
           push(stack, F);
  283
  284
          }
  285 }
  286 }
```

void divsr (T_stack * stack)

Divide todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de divisão entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

```
T_stack* stack: pilha na qual operandos e operadores são empilhados
```

Retorna:

void: não retorna nenhum valor

```
289
 290
       float F, A;
 291
 292
      if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 293
para retornar ao menu) \n");
 294
        getchar();
       } else {
  295
       if(stack->top->next == NULL) {
 296
 2.97
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
           getchar();
  299
         } else {
          F = pop(stack);
  300
  301
          do {
            A = pop(stack);
F = F / A;
  302
  303
  304
           } while(!emptystack(stack));
  305
           push(stack, F);
  306
  307
       }
 308 }
```

void mul (T_stack * stack)

Multiplica os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de multiplicação entre eles e empilha o resultado da operação na pilha.

Parâmetros:

T stack*	stack: pilha na qual operandos e operadores são empilhados	ı

Retorna:

```
227
  228
       float F, A;
 229
 230 if(emptystack(stack)) {
  231
         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
        getchar();
 2.32
  233
       } else {
       if(stack->top->next == NULL) {
 234
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
  235
Enter para retornar ao menu) \n");
 236
          getchar();
  237
         } else {
         F = pop(stack);
  238
          A = pop(stack);
F = F * A;
  239
  240
  241
           push(stack, F);
  242
  243 }
 244 }
```

void mulr (T_stack * stack)

Multiplica todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de multiplicação entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

T_stack*	stack: pilha na qual operandos e operadores são empilhados

Retorna:

void: não retorna nenhum valor

```
248
        float F, A;
 249
 250
       if(emptystack(stack)) {
 251
         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 252
         getchar();
  253
        } else {
 254 if(stack->top->next == NULL) {
255 printf("\nonext') }
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 256
            getchar();
  257
         } else {
  258
            F = pop(stack);
  259
           do {
           A = pop(stack);
F = F * A;
  260
  261
  262
           } while(!emptystack(stack));
  263
           push(stack, F);
  264
  265 }
  266 }
```

void sub (T_stack * stack)

Subtrai os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de subtração entre eles e empilha o resultado da operação na pilha.

Parâmetros:

T stasly*	stock milho no quel anguandos a anguadanas são ammilhodos
T stack*	stack: pilha na qual operandos e operadores são empilhados

Retorna:

```
185
 186
       float F, A;
 187
 188
      if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 189
para retornar ao menu) \n");
 190
        getchar();
  191
       } else {
       if(stack->top->next == NULL) {
 192
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
 193
Enter para retornar ao menu) \n");
 194
          getchar();
  195
         } else {
 196
          F = pop(stack);
 197
          A = pop(stack);
  198
           F = F - A;
 199
           push(stack, F);
 200
         }
      }
  2.01
 202 }
```

void subr (T_stack * stack)

Subtrai todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de subtração entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

```
T_stack* stack: pilha na qual operandos e operadores são empilhados
```

Retorna:

void: não retorna nenhum valor

```
205
 206
       float F, A;
  207
 208
      if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 209
para retornar ao menu) \n");
 210
        getchar();
       } else {
  211
       if(stack->top->next == NULL) {
 212
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
 213
Enter para retornar ao menu) \n");
          getchar();
  215
         } else {
          F = pop(stack);
 216
  217
          do {
            A = pop(stack);
F = F - A;
  218
  219
          } while(!emptystack(stack));
  220
  221
           push(stack, F);
  222
  223
       }
 224 }
```

void sum (T_stack * stack)

Soma os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de soma entre eles e empilha o resultado da operação na pilha.

Parâmetros:

T_stack* stack: pilha r

Retorna:

```
void: não retorna nenhum valor
  143
 144
       float F, A;
 145
 146 if(emptystack(stack)) {
         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 147
para retornar ao menu) \n");
        getchar();
 148
 149
       } else {
       if(stack->top->next == NULL) {
 150
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
 151
Enter para retornar ao menu) \n");
 152
          getchar();
 153
         } else {
         F = pop(stack);
 154
          A = pop(stack);

F = F + A;
 155
 156
 157
           push(stack, F);
  158
 159 }
 160 }
```

void sumr (T_stack * stack)

Soma todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de soma entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

```
T_stack* stack: pilha na qual operandos e operadores são empilhados
```

Retorna:

```
163
       float F, A;
 164
 165
 166
      if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 167
para retornar ao menu) \n");
 168
        getchar();
 169
       } else {
       if(stack->top->next == NULL) {
 170
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
 171
Enter para retornar ao menu) \n");
 172
          getchar();
         } else {
 173
 174
          F = pop(stack);
          do {
 175
            A = pop(stack);
F = F + A;
 176
 177
 178
           } while(!emptystack(stack));
 179
           push(stack, F);
 180
 181
      }
 182 }
```

Referência ao ficheiro expression.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stdtask.h"
```

Funções

• void **resofex** ()
Recebe e soluciona a expressão.

• int validexpression (T_stack *stack, char *IE) Valida a expressão.

- void **convertexpression** (**T_stack** ***stack**, char *IE, float *PE, int *SP, int N) *Converte a expressão de infica para pósfixa*.
- void **printexpression** (float *PE, int *SP) Exibe a expressao na forma pósfixa.
- float **calcexpression** (**T_stack** ***stack**, float *PE, int *SP) *Resolve a expressão*.
- void freebuffer ()Limpa o buffer.

Documentação das funções

float calcexpression (T_stack * stack, float * PE, int * SP)

Resolve a expressão.

Recebe a expressão na forma pósfixa armazenada no vetor PE, e com auxílio do vetor SP, que indica as posições no vetor PE que devem ser tratadas como operadores realiza as operações necessárias afim de solucionar a expressão, retornando o resultado.

Parâmetros:

T_stack*	stack: pilha onde os operandos são empilhados
float*	PE: armazena a expressão na forma pósfixa
int*	SP: o primeiro elemento do vetor SP armazena a quantidade de elementos no
	vetor PE e a partir do segundo elemento armazena as posições no vetor PE
	cujo os elementos devem ser tratados como operadores

Retorna:

float: resultado obtido a partir da resolução da expressão pósfixa

```
288
      freeelements(stack);
289
290
     float A, B, R;
291
     char O;
int i, j;
292
293
     for(i = 0, j = 1; i < SP[0]; i++) {
294
      if(i == SP[j]) {
295
        O = (char) PE[i];
296
         B = pop(stack);
297
298
        A = pop(stack);
        switch (0) {
    case '*':
299
300
             R = A * B;
301
302
              push(stack, R);
303
              break;
          case '/':
R = A / B;
304
305
```

```
push(stack, R);
306
307
             break;
308
           case '+':
           R = A + B;
309
            push(stack, R);
310
311
             break;
           case '-':
R = A - B;
312
313
314
             push(stack, R);
315
             break;
316
317
         j++;
318
      push(stack, PE[i]);
}
319
320
321
322 R = pop(stack);
323
     return R;
324 }
```

void convertexpression (T_stack * stack, char * IE, float * PE, int * SP, int N)

Converte a expressão de infica para pósfixa.

Recebe a expressão na forma infixa (que está no vetor IE) e converte para forma pósfixa (armazenando no vetor PE).

Parâmetros:

T_stack*	stack: pilha na qual os operadores são empilhados
char*	IE: armazena a expressão infixa recebida do usuário
float*	PE: armazena a expressão na forma pósfixa
int*	SP: o primeiro elemento do vetor SP armazena a quantidade de elementos no
	vetor PE e a partir do segundo elemento armazena as posições no vetor PE
	cujo os elementos devem ser tratados como operadores
int	N: quantidade de caracteres recebidos pelo usuário

Retorna:

```
91
 92
      freeelements(stack);
 93
 94
      float C, A;
 95
      char 0;
 96
 97
      int i, j, k, P;
 98
      for(i = 0, j = 0, k = 1; IE[i] != '\0' && i < N; i++) {
  if(j == 0 && emptystack(stack)) {</pre>
 99
100
101
          switch(IE[i]) {
102
            case ' ':
103
              break;
104
             case '{':
105
              C = (float) IE[i];
             push(stack, C);
break;
106
107
           case '[':
C = (float) IE[i];
108
109
110
              push(stack, C);
111
               break;
            case '(':
C = (float) IE[i];
112
113
114
               push(stack, C);
115
               break;
116
            default:
               C = atof(\&IE[i]);
117
118
               PE[j] = C;
119
120
               while(IE[i] >= '0' && IE[i] <= '9') {
121
                 i++;
                 if(IE[i] == '.') {
122
```

```
123
                    i++;
 124
                 }
 125
 126
                i--;
 127
                break;
 128
            } else {
 129
 130
              switch(IE[i]) {
 131
               case ' ':
 132
                 break;
                case '{':
C = (float) IE[i];
 133
 134
                 push(stack, C);
break;
 135
 136
                case '[':
C = (float) IE[i];
 137
 138
 139
                  push(stack, C);
 140
                  break;
                case '(':
C = (float) IE[i];
 141
 142
 143
                  push(stack, C);
 144
                  break;
                case '}':
0 = (char) pop(stack);
 145
 146
 147
                  while(O != '{') {
 148
                   C = (float) O;
                     PE[j] = C;
 149
 150
                    SP[k] = j;
 151
                     j++;
 152
                    k++;
 153
                    0 = (char) pop(stack);
 154
 155
                  break;
                case ']':
    O = (char) pop(stack);
 156
 157
                  while(O != '[') {
 158
 159
                    C = (float) O;
 160
                     PE[j] = C;
 161
                     SP[k] = j;
 162
                    j++;
                    k++;
 163
 164
                    0 = (char) pop(stack);
 165
 166
                  break;
                case ')':
    O = (char) pop(stack);
    while(O != '(') {
 167
 168
 169
 170
                   C = (float) O;
                    PE[j] = C;
SP[k] = j;
 171
 172
 173
                    j++;
 174
                     k++;
 175
                    0 = (char) pop(stack);
 176
 177
                  break;
 178
                case '*':
 179
                case '/':
                  if(emptystack(stack)) {
 180
 181
                    C = (float) IE[i];
 182
                    push(stack, C);
 183
                  } else {
 184
                    P = 1;
 185
                     while(!emptystack(stack) && P) {
                      C = (float) IE[i];
O = (char) pop(stack);
 186
 187
                       switch (O) {
 188
                        case '*':
 189
 190
                           PE[j] = (float) 0;
 191
                            SP[k] = j;
 192
 193
                            j++;
                           k++;
 194
 195
                           break;
 196
                          case '+':
 197
                          P = 0;
                         case '-':
 198
                           P = 0;
 199
```

```
200
                      default:
201
                       A = (float) O;
202
                        push(stack, A);
203
                        P = 0;
204
                        break;
205
206
207
                 push(stack, C);
208
209
               break;
             case '+':
210
             case '-':
211
212
               if(emptystack(stack)) {
213
                C = (float) IE[i];
                 push(stack, C);
214
215
                } else {
216
                 P = 1;
217
                  while(!emptystack(stack) && P) {
                   C = (float) IE[i];
218
                   O = (char) pop(stack);
219
220
                   switch (0) {
                    case '*':
221
222
                     case '/':
                     case '+':
223
                     case '-':
224
225
                       PE[j] = (float) O;
226
                       SP[k] = j;
227
                        j++;
228
                        k++;
229
                       break;
230
                      default:
                       A = (float) O;
231
232
                        push(stack, A);
233
                        P = 0;
234
                        break;
235
                   }
236
237
                 push(stack, C);
238
239
               break;
240
              default:
241
               C = atof(\&IE[i]);
242
               PE[j] = C;
243
                j++;
244
                while(IE[i] >= '0' && IE[i] <= '9') {
                 i++;
245
                  if(IE[i] == '.') {
246
247
                   i++;
248
249
250
                i--;
251
                break;
252
253
         }
254
255
256
     while(!emptystack(stack)) {
      PE[j] = pop(stack);
257
258
       SP[k] = j;
259
        j++;
260
       k++;
261
     }
262
     SP[0] = j;
263
264 }
```

void freebuffer ()

Limpa o buffer.

Executa a limpeza do buffer do teclado.

Parâmetros:

void	não recebe nenhum argumento

Retorna:

void: não retorna nenhum valor

```
327 {
328 char C;
329
330 while((C = getchar()) != '\n' && C != EOF);
331 }
```

void printexpression (float * PE, int * SP)

Exibe a expressao na forma pósfixa.

Exibe a expressao na forma pósfixa armazenada no vetor PE com auxílio do vetor SP, que indica as posições no vetor PE que devem ser tratadas como operadores.

Parâmetros:

float*	PE: armazena a na forma expressão pósfixa
int*	SP: o primeiro elemento do vetor SP armazena a quantidade de elementos no
	vetor PE e a partir do segundo elemento armazena as posições no vetor PE
	cujo os elementos devem ser tratados como operadores

Retorna:

void: não retorna nenhum valor

```
267
268
      char O;
269
     int i, j;
270
     printf("\nForma Posfixa: ");
272
     for(i = 0, j = 1; i < SP[0]; i++) {
  if(i == SP[j]) {</pre>
273
274
275
         O = (char) PE[i];
        printf(" %c ", 0);
276
277
          j++;
      } else {
278
          printf(" %.2f ", PE[i]);
279
280
281
282
     printf("\n");
283
284 }
```

void resofex ()

Recebe e soluciona a expressão.

Recebe a expressão do usuário e executa as funções necessárias para solucionar a expressão.

Parâmetros:

void	não recebe nenhum argumento	

Retorna:

```
system("clear");
    T stack* stack = allocstack();
 8
 9
10
    int N;
11
    float R;
12
    char IE[200];
13
14
    printf("\tResolucao da Expressao\n\nDigite a Expressao: ");
15
    scanf("%[^\n]s", IE);
16
    N = 0;
17
18
```

```
while(IE[N] != '\0') {
  19
  20
        N++;
  21
  22
  23
       if(!validexpression(stack, IE)) {
        printf("\n\tExpressao Invalida\n\n(pressione a tecla Enter para retornar ao
  2.4
menu)\n");
  25
       } else {
  26
         float PE[100] = {};
  27
         int SP[50] = {};
  28
  29
         printf("\n\tExpressao Valida\n");
   31
         ct.od(TE):
        convertexpression(stack, IE, PE, SP, N);
   32
   33
         printexpression(PE, SP);
        R = calcexpression(stack, PE, SP);
  34
        printf("\nResultado: %.2f\n", R);
   35
         printf("\n(pressione a tecla Enter para retornar ao menu)\n");
   36
   37
   38
       freebuffer();
   39
       freestack(stack);
  40 }
```

int validexpression (T_stack * stack, char * IE)

Valida a expressão.

Analisa os delimitadores da expressão e retorna 1 se a expressão é valida e 0 se é invalida.

Parâmetros:

T_stack*	stack: pilha na qual os delimitadores são empilhados
char*	IE: armazena a expressão recebida do usuário

Retorna:

int: identifica se a expressão é válida

```
43
44
     char C;
45
46
     int i = 0, V = 1;
47
     while(IE[i] != '\0') {
48
49
50
       if(IE[i] == '(' ||
          IE[i] == '[' ||
IE[i] == '{') {
51
52
53
             push(stack, (int) IE[i]);
54
55
      if(IE[i] == ')' ||
IE[i] == ']' ||
56
57
          IE[i] == '}') {
58
59
             if(emptystack(stack)) {
60
              V = 0;
61
            } else {
62
               C = (char) pop(stack);
63
64
               switch(IE[i]) {
                 case ')':
65
                   if(C != '(')
66
67
                      V = 0;
68
                    break;
69
                 case ']':
                    if(C != '[')
70
71
                      V = 0;
72
                    break;
73
                 case '}':
74
75
                    if(C != '{')
                      V = 0;
76
                    break;
77
               }
78
```

Referência ao ficheiro main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stdtask.h"
```

Funções

• int main ()

Documentação das funções

int main ()

```
6
       system("clear");
    8
    9
   10
       printf("\tMenu\n\n1. Resolucao da Expressao\n2. Calculadora\n3. Sair\n");
   11
   12
       printf("\nSelecione a Opcao: ");
  13
       0 = getchar();
   14
  15
       freebuffer();
  16
       if(0 == '1') {
  17
  18
         resofex();
       } else {
  19
         if(0 == '2') {
   20
  21
           calc();
         } else {
  if(0 == '3') {
   22
  23
           system("clear");
   24
   25
             exit(0);
   26
          } else {
            printf("\nOpcao Invalida\n\n(pressione a tecla Enter para retornar ao
  27
menu)\n");
  28
   29
         }
   30
   31
       while(0 != '3') {
   32
   33
        getchar();
         system("clear");
   34
   3.5
   36
        printf("\tMenu\n\n1. Resolucao da Expressao\n2. Calculadora\n3. Sair\n");
   37
   38
         printf("\nSelecione a Opcao: ");
   39
         0 = getchar();
   40
   41
         freebuffer();
   42
        if(0 == '1') {
   43
   44
           resofex();
         } else {
  if(0 == '2') {
   45
   46
   47
             calc();
   48
            } else {
             if(0 == '3') {
   49
               system("clear");
   50
   51
               exit(0);
             } else
   52
               printf("\nOpcao Invalida\n\n(pressione a tecla Enter para retornar ao
   53
menu) \n");
  54
   55
  56
         }
  57
   58
  59 return 0;
```

Referência ao ficheiro stack.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stdtask.h"
```

Funções

• T_stack * allocstack ()

Aloca de forma dinâmica uma estrutura de tipo pilha.

• **T_element** * **allocelement** (float D)

Aloca de forma dinâmica uma estrutura de tipo elemento.

• int emptystack (T_stack *stack)

Verifica se a pilha está vazia.

• void **push** (**T_stack** ***stack**, float D)

Empilha um elemto na pilha.

 $\bullet \quad \text{float pop } (T_stack *stack)$

Remove um elemento da pilha.

• void **printstack** (**T_stack** ***stack**) *Exibe a pilha*.

• void freeelements (T_stack *stack)

Libera os elementos da pilha.

• void freestack (T_stack *stack)

Libera a pilha e seus elementos.

Documentação das funções

T_element* allocelement (float D)

Aloca de forma dinâmica uma estrutura de tipo elemento.

Aloca de forma dinâmica uma estrutura de tipo elemento e configura o dado armazenado nela como D, e o ponteiro para o próximo como NULL.

Parâmetros:

float	D: dado à ser armazenado na estrutura de tipo elemento

Retorna:

T_element*: ponteiro para a estrutura de tipo elemento alocada

```
14
15  T_element* element = (T_element*) malloc(sizeof(T_element));
16
17  element->data = D;
18  element->next = NULL;
19
20  return element;
21 }
```

T_stack* allocstack ()

Aloca de forma dinâmica uma estrutura de tipo pilha.

Aloca de forma dinâmica uma estrutura de tipo pilha e configura o ponteiro para o elemento no todo da pilha como NULL.

Parâmetros:

void	não recebe nenhum parâmetro
voiu	nao recebe nemiam parametro

Retorna:

T_stack*: ponteiro para a estrutura de tipo pilha alocada

int emptystack (T_stack * stack)

Verifica se a pilha está vazia.

Retorna 1 se o topo da pilha aponta para NULL e 0 caso contrário.

Parâmetros:

CT Lab	1 1 111 (10) 1
T stack*	stack: estrutura de tipo pilha que será verificada
1_Stack	stack. Caratara de tipo pinia que sera verificada

Retorna:

```
int: identifica se a pilha está vazia
```

```
24 {
25 return (stack->top == NULL);
26 }
```

void freeelements (T_stack * stack)

Libera os elementos da pilha.

Libera apenas os elementos da pilha, mantendo a estrutura de pilha previamente alocada.

Parâmetros:

T_stack*	stack: pilha da qual os elementos serão liberados
----------	---

Retorna:

void: não retorna nenhum valor

```
if(!emptystack(stack)) {
      T element* now = stack->top;
77
78
     while(now != NULL) {
79
       stack->top = now->next;
81
        free (now);
82
        now = stack->top;
83
     }
84
85
    stack->top = NULL;
86 }
```

void freestack (T_stack * stack)

Libera a pilha e seus elementos.

Libera os elementos da pilha e a pilha.

Parâmetros:

T_stack*	stack: pilha que será liberada

Retorna:

```
89
90 if(!emptystack(stack)) {
91  T element* now = stack->top;
92
```

float pop (T_stack * stack)

Remove um elemento da pilha.

Retira o elemento que está no topo da pilha, apontando o topo da pilha para o próximo elemento, e retorna o dado armazenado no elemento retirado.

Parâmetros:

T_stack*	stack: pilha da qual o elemento será retirado

Retorna:

float: dado armazenado no elemento retirado

```
42
    float aux;
43
44
    if(!emptystack(stack)) {
45
     T element* element = stack->top;
46
47
     aux = element->data;
48
49
      stack->top = element->next;
50
      free(element);
51
52
53
    return aux;
54 }
```

void printstack (T_stack * stack)

Exibe a pilha.

Exibe todods os dados armazenados nos elementos da pilha.

Parâmetros:

```
T_stack* stack: pilha à ser exibida
```

Retorna:

void: não retorna nenhum valor

```
58
    if(emptystack(stack)) {
59
     printf("Pilha Vazia\n");
60
    } else {
61
      T element* now = stack->top;
62
     int i = 1;
63
64
     while(now != NULL) {
65
66
       printf("%d. %.2f\n", i, now->data);
67
68
        i++;
69
        now = now->next;
70
      }
71
    }
72 }
```

void push (T_stack * stack, float D)

Empilha um elemto na pilha.

Executa a função allocelement, configura o ponteiro para o próximo do elemento alocado como o elemento no topo da pilha e aponta o topo da pilha para o elemento alocado.

Parâmetros:

T_stack*	stack: estrutura de tipo pilha na qual o elemento será empilhado
float	D: dado que será armazenado na estrutura de tipo elemento

Retorna:

```
29
30  T_element* element = allocelement(D);
31
32  if(emptystack(stack)) {
33    stack->top = element;
34  } else {
35    element->next = stack->top;
36    stack->top = element;
37  }
38 }
```

Referência ao ficheiro stdtask.h

Este arquivo contém as declarações de todas as funções implementadas.

Estruturas de Dados

- struct element
- Estrutura de tipo elemento. struct stack

Estrutura de tipo pilha. Definições de tipos

• typedef struct **element T_element**

Estrutura de tipo elemento.

• typedef struct stack T_stack

Estrutura de tipo pilha.

Funções

• T_stack * allocstack ()

Aloca de forma dinâmica uma estrutura de tipo pilha.

• **T_element** * **allocelement** (float D)

Aloca de forma dinâmica uma estrutura de tipo elemento.

• int emptystack (T_stack *stack)

Verifica se a pilha está vazia.

• void **push** (**T_stack** ***stack**, float D)

Empilha um elemto na pilha.

• float pop (T_stack *stack)

Remove um elemento da pilha.

• void printstack (T_stack *stack)

Exibe a pilha.

• void freeelements (T stack *stack)

Libera os elementos da pilha.

• void freestack (T_stack *stack)

Libera a pilha e seus elementos.

• void resofex ()

Recebe e soluciona a expressão.

• int validexpression (T_stack *stack, char *IE)

Valida a expressão.

• void convertexpression (T_stack *stack, char *IE, float *PE, int *SP, int N)

Converte a expressão de infica para pósfixa.

• void **printexpression** (float *PE, int *SP)

Exibe a expressao na forma pósfixa.

• float calcexpression (T_stack *stack, float *PE, int *SP)

Resolve a expressão.

• void freebuffer ()

Limpa o buffer.

void calc ()

Recebe operandos e operadores.

• void sum (T_stack *stack)

Soma os 2 primeiros dados da pilha.

• void sumr (T_stack *stack)

Soma todos os dados da pilha.

• void **sub** (**T_stack** ***stack**)
Subtrai os 2 primeiros dados da pilha.

• void **subr** (**T_stack** ***stack**)
Subtrai todos os dados da pilha.

• void mul (T_stack *stack)

Multiplica os 2 primeiros dados da pilha.

void mulr (T_stack *stack)
 Multiplica todos os dados da pilha.

• void divs (T_stack *stack)

Divide os 2 primeiros dados da pilha.

• void divsr (T_stack *stack)

Divide todos os dados da pilha.

• void copy (T_stack *stack)

Copia o primeiro elemento da pilha.

void ctod (char *S)
 Transforma vírgula em ponto.

Descrição detalhada

Este arquivo contém as declarações de todas as funções implementadas.

Autor:

Thiago Santos Matos

Data:

25 de outubro de 2017

Todas as funções implementadas nos arquivos "stack.c", "expression.c" e "calculator.c" estão declaradas neste arquivo.

Documentação dos tipos

typedef struct element T_element

Estrutura de tipo elemento.

Nesta estrutura é armazenado um dado no formato float e um ponteiro para o próximo elemento.

typedef struct stack T_stack

Estrutura de tipo pilha.

Nesta estrutura é armazenado o endereço do elemento que está no topo da pilha.

Documentação das funções

T_element* allocelement (float D)

Aloca de forma dinâmica uma estrutura de tipo elemento.

Aloca de forma dinâmica uma estrutura de tipo elemento e configura o dado armazenado nela como D, e o ponteiro para o próximo como NULL.

Parâmetros:

float	D: dado à ser armazenado na estrutura de tipo elemento
-------	--

Retorna:

T_element*: ponteiro para a estrutura de tipo elemento alocada

```
14
15  T_element* element = (T_element*) malloc(sizeof(T_element));
16
17  element->data = D;
18  element->next = NULL;
19
20  return element;
21 }
```

T_stack* allocstack ()

Aloca de forma dinâmica uma estrutura de tipo pilha.

Aloca de forma dinâmica uma estrutura de tipo pilha e configura o ponteiro para o elemento no todo da pilha como NULL.

Parâmetros:

void	não recebe nenhum parâmetro
------	-----------------------------

Retorna:

T_stack*: ponteiro para a estrutura de tipo pilha alocada

void calc ()

Recebe operandos e operadores.

Recebe os operandos e operadores do usuário e executa as funções necessárias para solucionar as operações.

Parâmetros:

• 1	
void	não recebe nenhum argumento

Retorna:

```
system("clear");
    T stack* stack = allocstack();
10
    char S[50];
11
    float F;
12
13
    system("clear");
14
15
    printf("\tCalculadora\t(para sair da calculadora pressione s)\n\n");
16
17
    printstack(stack);
18
    printf("\n-> ");
19
20
    scanf("%s", S);
21
    ctod(S);
22
```

```
23
    freebuffer();
24
25
    switch (S[0]) {
26
     case '+':
27
        if(S[1] == '!') {
28
          sumr(stack);
       sum(stack);
}
       } else {
29
30
31
32
        break;
      case '-':
33
        if(S[1] >= '0' && S[1] <= '9') {
34
35
         F = atof(S);
          push(stack, F);
36
       } else {
37
         if(S[1] == '!') {
38
39
            subr(stack);
40
          } else {
41
            sub(stack);
         }
42
43
44
        break;
      case '*':
45
        if(S[1] == '!') {
46
47
         mulr(stack);
48
        } else {
49
          mul(stack);
50
        break;
51
52
     case '/':
       if(S[1] == '!') {
53
54
          divsr(stack);
55
       divs(stack);
}
        } else {
56
57
58
        break;
      case 'C':
59
       copy(stack);
break;
60
61
      case 'c':
62
       copy(stack);
63
64
        break;
65
      case 'S':
66
        break;
67
      case 's':
68
        break;
69
      default:
70
       F = atof(S);
71
         push(stack, F);
72
        break;
73
74
75
    while(S[0] != 'S' && S[0] != 's') {
76
     system("clear");
77
78
     printf("\tCalculadora\t(para sair da calculadora pressione s)\n\n");
79
80
      printstack(stack);
81
82
      printf("\n-> ");
83
      scanf("%s", S);
84
      ctod(S);
85
86
      freebuffer();
87
88
      switch (S[0]) {
        case '+':
if(S[1] == '!') {
89
90
91
            sumr(stack);
92
           } else {
93
            sum(stack);
          }
94
          break;
95
96
         case '-':
         if(S[1] >= '0' && S[1] <= '9') {
97
            F = atof(S);
98
99
            push(stack, F);
```

```
100
           } else {
             if(S[1] == '!') {
101
102
              subr(stack);
103
             } else {
104
              sub(stack);
105
106
         break;
107
108
        case '*':
         if(S[1] == '!') {
109
            mulr(stack);
110
         } else {
111
         mul(stack);
112
113
          break;
114
       case '/':
115
         if(S[1] == '!') {
116
117
            divsr(stack);
118
          } else {
119
            divs(stack);
120
         break;
121
        case 'C':
122
        copy(stack);
break;
123
124
       case 'c':
125
       copy(stack);
break;
case 'S':
126
127
128
129
          break;
130
       case 's':
131
          break;
132
        default:
        F = atof(S);
133
134
          push(stack, F);
135
          break;
      }
136
137
138
     printf("\n(pressione a tecla Enter para retornar ao menu)\n");
139
     freestack(stack);
```

float calcexpression (T_stack * stack, float * PE, int * SP)

Resolve a expressão.

Recebe a expressão na forma pósfixa armazenada no vetor PE, e com auxílio do vetor SP, que indica as posições no vetor PE que devem ser tratadas como operadores realiza as operações necessárias afim de solucionar a expressão, retornando o resultado.

Parâmetros:

T_stack*	stack: pilha onde os operandos são empilhados
float*	PE: armazena a expressão na forma pósfixa
int*	SP: o primeiro elemento do vetor SP armazena a quantidade de elementos no
	vetor PE e a partir do segundo elemento armazena as posições no vetor PE
	cujo os elementos devem ser tratados como operadores

Retorna:

float: resultado obtido a partir da resolução da expressão pósfixa

```
287
288
     freeelements(stack);
289
290
     float A, B, R;
291
     char O;
    int i, j;
292
293
294
     for (i = 0, j = 1; i < SP[0]; i++) {
295
      if(i == SP[j]) {
       O = (char) PE[i];
296
         B = pop(stack);
297
        A = pop(stack);
298
299
        switch (0) {
```

```
case '*':
300
            R = A * B;
301
302
             push(stack, R);
             break;
303
           case '/':
R = A / B;
304
305
            push(stack, R);
break;
306
307
308
          case '+':
            R = A + B;
309
            push(stack, R);
break;
310
311
312
           case '-':
313
             R = A - B;
              push(stack, R);
314
315
              break;
316
      j++;
} else {
317
318
      push(stack, PE[i]);
}
319
320
321 }
     R = pop(stack);
return R;
322
323
324 }
```

void convertexpression (T_stack * stack, char * IE, float * PE, int * SP, int N)

Converte a expressão de infica para pósfixa.

Recebe a expressão na forma infixa (que está no vetor IE) e converte para forma pósfixa (armazenando no vetor PE).

Parâmetros:

T_stack*	stack: pilha na qual os operadores são empilhados
char*	IE: armazena a expressão infixa recebida do usuário
float*	PE: armazena a expressão na forma pósfixa
int*	SP: o primeiro elemento do vetor SP armazena a quantidade de elementos no
	vetor PE e a partir do segundo elemento armazena as posições no vetor PE
	cujo os elementos devem ser tratados como operadores
int	N: quantidade de caracteres recebidos pelo usuário

Retorna:

```
91
 92
     freeelements(stack);
 93
 94
     float C, A;
 95
      char 0;
 96
 97
     int i, j, k, P;
 98
 99
     for(i = 0, j = 0, k = 1; IE[i] != '\0' \&\& i < N; i++) {
      if(j == 0 && emptystack(stack)) {
100
         switch(IE[i]) {
101
           case ' ':
102
103
              break;
           case '{':
   C = (float) IE[i];
104
105
106
             push(stack, C);
107
             break;
          case '[':
    C = (float) IE[i];
108
109
            push(stack, C);
break;
110
111
112
          case '(':
113
             C = (float) IE[i];
114
              push(stack, C);
115
              break;
     default:
116
```

```
C = atof(\&IE[i]);
117
118
               PE[j] = C;
119
               j++;
120
               while(IE[i] >= '0' && IE[i] <= '9') {
121
                 i++;
                 if(IE[i] == '.') {
122
123
                   i++;
124
125
126
               i--;
127
               break;
128
129
           } else {
130
             switch(IE[i]) {
               case ' ':
131
132
                 break;
133
               case '{':
                C = (float) IE[i];
134
135
                 push(stack, C);
136
                 break;
               case '[':
    C = (float) IE[i];
137
138
139
                 push(stack, C);
140
                 break;
141
               case '(':
142
                 C = (float) IE[i];
143
                 push(stack, C);
144
                 break;
               Dreak;
case '}':
    0 = (char) pop(stack);
    while(0 != '{') {
145
146
147
148
                   C = (float) O;
                    PE[j] = C;
149
                   SP[k] = j;
150
151
                   j++;
152
                   k++;
153
                   0 = (char) pop(stack);
154
                  }
155
                 break;
156
               case ']':
                 0 = (char) pop(stack);
while(0 != '[') {
157
158
159
                  C = (float) O;
160
                    PE[j] = C;
161
                   SP[k] = j;
162
                   j++;
163
                   k++;
164
                   0 = (char) pop(stack);
165
                  }
166
                 break;
               case ')':
167
168
                 0 = (char) pop(stack);
                 while(O != '(') {
169
                  C = (float) 0;
PE[j] = C;
170
171
                   SP[k] = j;
172
173
                    j++;
                   k++;
174
175
                   0 = (char) pop(stack);
176
177
                 break;
               case '*':
178
179
                 if(emptystack(stack)) {
180
181
                   C = (float) IE[i];
182
                   push(stack, C);
183
                 } else {
184
                   P = 1;
185
                    while(!emptystack(stack) && P) {
186
                     C = (float) IE[i];
                      O = (char) pop(stack);
187
                      switch (0) {
  case '*':
188
189
                        case '/':
190
191
                          PE[j] = (float) O;
                          SP[k] = j;
192
193
                          j++;
```

```
194
                          k++;
 195
                          break;
                        case '+':
 196
                         P = 0;
 197
 198
                        case '-':
                          P = 0;
 199
 200
                        default:
 201
                         A = (float) O;
 202
                          push(stack, A);
 203
                          P = 0;
 204
                          break;
 205
 206
                 push(stack, C);
}
 207
 208
 209
                 break;
               case '+':
case '-':
 210
 211
 212
                 if(emptystack(stack)) {
 213
                  C = (float) IE[i];
 214
                    push(stack, C);
 215
                 } else {
 216
                   P = 1;
 217
                    while(!emptystack(stack) && P) {
                     C = (float) IE[i];
 218
                     0 = (char) pop(stack);
switch (0) {
  case '*':
   case '/':
 219
 220
 221
 222
                        case '+':
 223
 224
                        case '-':
                         PE[j] = (float) 0;
 225
                          SP[k] = j;
 226
 227
                          j++;
                         k++;
 228
 229
                          break;
 230
                        default:
 231
                         A = (float) O;
 232
                          push(stack, A);
 233
                          P = 0;
 234
                          break;
 235
                     }
 236
 237
                   push(stack, C);
 238
 239
                 break;
 240
               default:
 241
                C = atof(\&IE[i]);
 242
                 PE[j] = C;
 243
                  j++;
                  while(IE[i] >= '0' && IE[i] <= '9') {
 244
 245
                   i++;
 246
                   if(IE[i] == '.') {
                     i++;
 247
 248
 249
 250
 251
                 break;
 252
            }
 253
 254
 255
       while(!emptystack(stack)) {
 256
       PE[j] = pop(stack);
 257
 258
         SP[k] = j;
        j++;
k++;
 259
 260
 261
 262
 263
       SP[0] = j;
264 }
```

void copy (T_stack * stack)

Copia o primeiro elemento da pilha.

Desempilha o primeiro e o segundo elemento da pilha, o número armazenado no primeiro elemento desempilhado dirá quantas vezes o segundo elemento deverá ser repetidamente empilhado.

Parâmetros:

T_stack*	stack: pilha na qual operandos e operadores são empilhados
----------	--

Retorna:

void: não retorna nenhum valor

```
311
  312
       int F, i;
 313
       float A;
  314
      if(emptystack(stack)) {
 315
 316
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 317 getchar();
 319 if(stack->top->next == NULL) {
320 printf("\nonext")
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 321
           getchar();
  322
         } else {
         F = (int) pop(stack);
A = pop(stack);
 323
  324
  325
          for(i = 0; i < F; i++) {
  326
             push(stack, A);
  327
  328
         }
  329 }
 330 }
```

void ctod (char * S)

Transforma vírgula em ponto.

Percorre a string recebida do usário transformando todos os caracteres ',' em '.'.

Parâmetros:

char*	S: armazena uma string recebida do usuário	
-------	--	--

Retorna:

void: não retorna nenhum valor

```
333 {
334 int i;
335
336 for(i = 0; S[i] != '\0'; i++) {
337    if(S[i] == ',') {
338       S[i] = '.';
339    }
340 }
341 }
```

void divs (T_stack * stack)

Divide os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de divisão entre eles e empilha o resultado da operação na pilha.

Parâmetros:

<i>T_stack*</i> stack: pilha na qual operandos e operadores são empilhados
--

Retorna:

```
void: não retorna nenhum valor

269 {
270 float F, A;
```

```
271
  2.72
       if(emptystack(stack)) {
  273
         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 274
        getchar();
 275
       } else {
 276 if (stack->top->next == NULL) {
 277
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 278
          getchar();
  279
         } else {
 280
          F = pop(stack);
          A = pop(stack);

F = F / A;
  281
  282
  283
          push(stack, F);
  284
  285 }
 286 }
```

void divsr (T_stack * stack)

Divide todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de divisão entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

T stack*	stack: pilha na qual operandos e operadores são empilhados

Retorna:

void: não retorna nenhum valor

```
289
 290
      float F, A;
 291
 292
      if(emptystack(stack)) {
 293
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 294 getchar();
 295
      } else {
       if(stack->top->next == NULL) {
 296
 297
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 298
           getchar();
        } else {
 299
 300
           F = pop(stack);
 301
          do {
 302
            A = pop(stack);
             F = F / A;
 303
 304
           } while(!emptystack(stack));
  305
           push(stack, F);
 306
 307
       }
 308 }
```

int emptystack (T_stack * stack)

Verifica se a pilha está vazia.

Retorna 1 se o topo da pilha aponta para NULL e 0 caso contrário.

Parâmetros:

```
T_stack* stack: estrutura de tipo pilha que será verificada
```

Retorna:

```
int: identifica se a pilha está vazia
24
25    return (stack->top == NULL);
26 }
```

void freebuffer ()

Limpa o buffer.

Executa a limpeza do buffer do teclado.

Parâmetros:

void	não recebe nenhum argumento
voiu	nao recebe nemiam argumento

Retorna:

void: não retorna nenhum valor

```
327 {
328 char C;
329
330 while((C = getchar()) != '\n' && C != EOF);
331 }
```

void freeelements (T_stack * stack)

Libera os elementos da pilha.

Libera apenas os elementos da pilha, mantendo a estrutura de pilha previamente alocada.

Parâmetros:

T_stack*	stack: pilha da qual os elementos serão liberados
----------	---

Retorna:

void: não retorna nenhum valor

```
76
    if(!emptystack(stack)) {
77
      T element* now = stack->top;
78
     while(now != NULL) {
79
80
      stack->top = now->next;
81
        free (now);
82
        now = stack->top;
83
84
    }
    stack->top = NULL;
85
86 }
```

void freestack (T_stack * stack)

Libera a pilha e seus elementos.

Libera os elementos da pilha e a pilha.

Parâmetros:

```
T_stack* stack: pilha que será liberada
```

Retorna:

void: não retorna nenhum valor

```
89
     if(!emptystack(stack)) {
 91
       T element* now = stack->top;
 92
 93
      while(now != NULL) {
 94
       stack->top = now->next;
 95
         free (now);
 96
         now = stack->top;
97
98
99
100
     free (stack);
101 }
```

void mul (T_stack * stack)

Multiplica os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de multiplicação entre eles e empilha o resultado da operação na pilha.

Parâmetros:

```
T_stack* stack: pilha na qual operandos e operadores são empilhados
```

Retorna:

void: não retorna nenhum valor

```
228
       float F, A;
 229
 230
      if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 231
para retornar ao menu) \n");
 232
        getchar();
       } else {
  233
       if(stack->top->next == NULL) {
 234
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
 2.35
Enter para retornar ao menu) \n");
           getchar();
  237
          } else {
        F = pop(stack);
A = pop(stack);
F = F * A;
 238
  239
  240
           push(stack, F);
  241
 242
         }
      }
 243
 244 }
```

void mulr (T_stack * stack)

Multiplica todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de multiplicação entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

Retorna:

void: não retorna nenhum valor

```
248
       float F, A;
 249
 250
      if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 252
        getchar();
  253
       } else {
      if(stack->top->next == NULL) {
 254
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 256
          getchar();
  257
         } else {
  258
           F = pop(stack);
  259
           do {
            A = pop(stack);
  260
  261
            F = F * A;
           } while(!emptystack(stack));
  262
  263
           push(stack, F);
 2.64
         }
      }
  265
 266 }
```

float pop (T_stack * stack)

Remove um elemento da pilha.

Retira o elemento que está no topo da pilha, apontando o topo da pilha para o próximo elemento, e retorna o dado armazenado no elemento retirado.

Parâmetros:

T_stack*	stack: pilha da qual o elemento será retirado

Retorna:

float: dado armazenado no elemento retirado

```
{
   float aux;
42
43
44
   if(!emptystack(stack)) {
45
     T_element* element = stack->top;
46
47
    aux = element->data;
48
49
     stack->top = element->next;
50
    free(element);
51
52
53
    return aux;
54 }
```

void printexpression (float * PE, int * SP)

Exibe a expressao na forma pósfixa.

Exibe a expressao na forma pósfixa armazenada no vetor PE com auxílio do vetor SP, que indica as posições no vetor PE que devem ser tratadas como operadores.

Parâmetros:

float*	PE: armazena a na forma expressão pósfixa
int*	SP: o primeiro elemento do vetor SP armazena a quantidade de elementos no
	vetor PE e a partir do segundo elemento armazena as posições no vetor PE
	cujo os elementos devem ser tratados como operadores

Retorna:

void: não retorna nenhum valor

```
267
268
      char 0;
269
     int i, j;
270
271 printf("\nForma Posfixa: ");
272
273 for(i = 0, j = 1; i < SP[0]; i++) {
274 if(i == SP[j]) {
         O = (char) PE[i];
275
276
        printf(" %c ", 0);
277
      } else {
278
         printf(" %.2f ", PE[i]);
279
280
281
282
    printf("\n");
283
284 }
```

void printstack (T_stack * stack)

Exibe a pilha.

Exibe todods os dados armazenados nos elementos da pilha.

Parâmetros:

T ctack*	stack: pilha à ser exibida	
I Stack*	Stack, pilita a sci catolida	

Retorna:

void: não retorna nenhum valor

```
58
    if(emptystack(stack)) {
     printf("Pilha Vazia\n");
60
    } else {
61
      T element* now = stack->top;
62
63
      int i = 1;
64
     while(now != NULL) {
65
       printf("%d. %.2f\n", i, now->data);
66
67
68
        i++;
        now = now->next;
69
70
71
72 }
```

void push (T_stack * stack, float D)

Empilha um elemto na pilha.

Executa a função allocelement, configura o ponteiro para o próximo do elemento alocado como o elemento no topo da pilha e aponta o topo da pilha para o elemento alocado.

Parâmetros:

T_stack*	stack: estrutura de tipo pilha na qual o elemento será empilhado
float	D: dado que será armazenado na estrutura de tipo elemento

Retorna:

void: não retorna nenhum valor

```
29
30  T element* element = allocelement(D);
31
32  if(emptystack(stack)) {
33    stack->top = element;
34  } else {
35    element->next = stack->top;
36    stack->top = element;
37  }
38 }
```

void resofex ()

Recebe e soluciona a expressão.

Recebe a expressão do usuário e executa as funções necessárias para solucionar a expressão.

Parâmetros:

void	não recebe nenhum argumento

Retorna:

void: não retorna nenhum valor

```
16
       N = 0:
   17
   18
  19
       while(IE[N] != '\0') {
   20
         N++;
   2.1
   22
   23
       if(!validexpression(stack, IE)) {
   24
        printf("\n\tExpressao Invalida\n\n(pressione a tecla Enter para retornar ao
menu) \n");
  25
       } else {
   26
         float PE[100] = {};
   27
         int SP[50] = {};
   28
   29
        printf("\n\tExpressao Valida\n");
   30
   31
        convertexpression(stack, IE, PE, SP, N);
   32
   33
         printexpression(PE, SP);
        R = calcexpression(stack, PE, SP);
   34
        printf("\nResultado: %.2f\n", R);
   35
   36
         printf("\n(pressione a tecla Enter para retornar ao menu)\n");
   37
   38
       freebuffer();
   39
       freestack(stack);
   40 }
```

void sub (T_stack * stack)

Subtrai os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de subtração entre eles e empilha o resultado da operação na pilha.

Parâmetros:

T_stack*	stack: pilha na qual operandos e operadores são empilhados
----------	--

Retorna:

void: não retorna nenhum valor

```
185
 186
       float F, A;
  187
  188
       if(emptystack(stack)) {
  189
         printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 190 getchar();
  191
 192  if(stack->top->next == NULL) {
193    printf("\nonext');
       } else {
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 194
           getchar();
          } else {
  195
         F = pop(stack);
A = pop(stack);
  196
  197
  198
           F = F - A;
  199
            push(stack, F);
  200
         }
  201
       }
  202 }
```

void subr (T_stack * stack)

Subtrai todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de subtração entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

|--|

Retorna:

void: não retorna nenhum valor

```
206
        float F, A;
 207
 208
       if(emptystack(stack)) {
 209
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 210
        getchar();
 212 if(stack->top->next == NULL) {
213 printf("\nonext') }
      } else {
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 214
           getchar();
         } else {
  215
  216
           F = pop(stack);
           do {
  217
  218
           A = pop(stack);
              F = F - A;
  219
  220
          } while(!emptystack(stack));
           push(stack, F);
  221
  222
      }
  223
  224 }
```

void sum (T stack * stack)

Soma os 2 primeiros dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de soma entre eles e empilha o resultado da operação na pilha.

Parâmetros:

l	
T_stack*	stack: pilha na qual operandos e operadores são empilhados
I SIUCK	I SIACK. DITTA HA UUAI ODELAHUOS E ODELAUOTES SAO EITIDITTAUOS

Retorna:

void: não retorna nenhum valor

```
143
 144
       float F, A;
 145
 146 if(emptystack(stack)) {
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
 147
para retornar ao menu) \n");
 148
         getchar();
  149
       } else {
       if(stack->top->next == NULL) {
 150
 151
           printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
 152
           getchar();
 153
         } else {
 154
        F = pop(stack);
A = pop(stack);
F = F + A;
 155
  156
           push(stack, F);
 157
 158
         }
 159
 160 }
```

void sumr (T stack * stack)

Soma todos os dados da pilha.

Desempilha o primeiro e o segundo elemento da pilha, realiza a operação de soma entre eles e empilha o resultado da operação na pilha, enqunto a pilha não estiver vazia.

Parâmetros:

Retorna:

void: não retorna nenhum valor

```
164
        float F, A;
 165
 166
       if(emptystack(stack)) {
 167
        printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu) \n");
 168 getchar();
 170 if(stack->top->next == NULL) {
171 printf("\nOvertile") {
 169
       } else {
          printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu) \n");
           getchar();
 172
         } else {
 173
 174
           F = pop(stack);
 175
           do {
           A = pop(stack);
F = F + A;
 176
 177
 178
           } while(!emptystack(stack));
  179
           push(stack, F);
 180
 181 }
 182 }
```

int validexpression (T_stack * stack, char * IE)

Valida a expressão.

Analisa os delimitadores da expressão e retorna 1 se a expressão é valida e 0 se é invalida.

Parâmetros:

T_stack*	stack: pilha na qual os delimitadores são empilhados
char*	IE: armazena a expressão recebida do usuário

Retorna:

int: identifica se a expressão é válida

```
43
44
     char C;
45
    int i = 0, V = 1;
46
47
48
     while(IE[i] != '\0') {
49
     if(IE[i] == '(' ||
IE[i] == '[' ||
50
51
          IE[i] == '{') {
52
53
            push(stack, (int) IE[i]);
54
55
     if(IE[i] == ')' ||
IE[i] == ']' ||
56
57
          IE[i] == '}') {
58
59
            if(emptystack(stack)) {
60
              V = 0;
61
             } else {
              C = (char) pop(stack);
62
63
              switch(IE[i]) {
64
65
                case ')':
                   if(C != '(')
66
                     V = 0;
67
68
                    break;
69
                 case ']':
70
                   if(C != '[')
71
                     V = 0;
72
                   break;
73
                 case '}':
74
                 if(C != '{')
75
                     V = 0;
76
                    break;
```

Índice

INDEX