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:

Índice dos ficheiros

Lista de ficheiros

Lista de todos os ficheiros com uma breve descrição:

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:

void: não retorna nenhum valor

```
 5            {
 6   system("clear");
 7
 8   T stack* stack = allocstack();
 9
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
19  printf("\n-> ");
20  scanf("%s", S);
21  ctod(S);
22
23  freebuffer();
24
25  switch (S[0]) {
26    case '+':
27      if(S[1] == '!') {
28        sumr(stack);
29      } else {
30        sum(stack);
31      }
32      break;
33    case '-':
34      if(S[1] >= '0' && S[1] <= '9') {
35        F = atof(S);
36        push(stack, F);
37      } else {
38        if(S[1] == '!') {
39          subr(stack);
40        } else {
41          sub(stack);
42        }
43      }
44      break;
45    case '*':
46      if(S[1] == '!') {
47        mulr(stack);
48      } else {
49        mul(stack);
50      }
51      break;
52    case '/':
53      if(S[1] == '!') {
54        divsr(stack);
55      } else {
56        divs(stack);
57      }
58      break;
59    case 'C':
60      copy(stack);
61      break;
62    case 'c':
63      copy(stack);
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]) {
```

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

Retorna:

void: não retorna nenhum valor

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

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

Retorna:

void: não retorna nenhum valor

```
  269                           {
  270   float F, A;
  271
  272   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);
  281       A = pop(stack);
  282       F = F / A;
  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 {
  296     if(stack->top->next == NULL) {
  297       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  298       getchar();
  299     } else {
  300       F = pop(stack);
  301       do {
  302         A = pop(stack);
  303         F = F / A;
  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

Retorna:

void: não retorna nenhum valor

```
  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");
  232     getchar();
  233   } else {
  234     if(stack->top->next == NULL) {
  235       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  236       getchar();
  237     } else {
  238       F = pop(stack);
  239       A = pop(stack);
  240       F = F * A;
  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

```
  247                            {
  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("\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 {
  260         A = pop(stack);
  261         F = F * A;
  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_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   } else {
  192     if(stack->top->next == NULL) {
  193       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
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     }
  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:

*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)) {
  209     printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu)\n");
  210     getchar();
  211   } else {
  212     if(stack->top->next == NULL) {
  213       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  214       getchar();
  215     } else {
  216       F = pop(stack);
  217       do {
  218         A = pop(stack);
  219         F = F - A;
  220       } while(!emptystack(stack));
  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 na qual operandos e operadores são empilhados

Retorna:

void: não retorna nenhum valor

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

void: não retorna nenhum valor

```
  163                             {
  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();
  169   } else {
  170     if(stack->top->next == NULL) {
  171       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  172       getchar();
  173     } else {
  174       F = pop(stack);
  175       do {
  176         A = pop(stack);
  177         F = F + A;
  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

```
287                                                          {
288   freeelements(stack);
289
290   float A, B, R;
291   char O;
292   int i, j;
293
294   for(i = 0, j = 1; i < SP[0]; i++) {
295     if(i == SP[j]) {
296       O = (char) PE[i];
297       B = pop(stack);
298       A = pop(stack);
299       switch (O) {
300         case '*':
301           R = A * B;
302           push(stack, R);
303           break;
304         case '/':
305           R = A / B;
```

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

void: não retorna nenhum valor

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

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

```
200                   default:
201                     A = (float) O;
202                     push(stack, A);
203                     P = 0;
204                     break;
205                   }
206               }
207               push(stack, C);
208             }
209             break;
210           case '+':
211           case '-':
212             if(emptystack(stack)) {
213               C = (float) IE[i];
214               push(stack, C);
215             } else {
216               P = 1;
217               while(!emptystack(stack) && P) {
218                 C = (float) IE[i];
219                 O = (char) pop(stack);
220                 switch (O) {
221                   case '*':
222                   case '/':
223                   case '+':
224                   case '-':
225                     PE[j] = (float) O;
226                     SP[k] = j;
227                     j++;
228                     k++;
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++;
244             while(IE[i] >= '0' && IE[i] <= '9') {
245               i++;
246               if(IE[i] == '.') {
247                 i++;
248               }
249             }
250             i--;
251             break;
252         }
253       }
254   }
255
256   while(!emptystack(stack)) {
257     PE[j] = pop(stack);
258     SP[k] = j;
259     j++;
260     k++;
261   }
262
263   SP[0] = j;
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
271   printf("\nForma Posfixa: ");
272
273   for(i = 0, j = 1; i < SP[0]; i++) {
274     if(i == SP[j]) {
275       O = (char) PE[i];
276       printf(" %c ", O);
277       j++;
278     } else {
279       printf(" %.2f ", PE[i]);
280     }
281   }
282
283   printf("\n");
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:

void: não retorna nenhum valor

```
  5                 {
  6   system("clear");
  7
  8   T stack* stack = allocstack();
  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
 17   N = 0;
 18
```

```
   19   while(IE[N] != '\0') {
   20     N++;
   21   }
   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     ctod(IE);
   32     convertexpression(stack, IE, PE, SP, N);
   33     printexpression(PE, SP);
   34     R = calcexpression(stack, PE, SP);
   35     printf("\nResultado: %.2f\n", R);
   36     printf("\n(pressione a tecla Enter para retornar ao menu)\n");
   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
   48   while(IE[i] != '\0') {
   49
   50     if(IE[i] == '(' ||
   51        IE[i] == '[' ||
   52        IE[i] == '{') {
   53          push(stack, (int) IE[i]);
   54        }
   55
   56     if(IE[i] == ')' ||
   57        IE[i] == ']' ||
   58        IE[i] == '}') {
   59          if(emptystack(stack)) {
   60            V = 0;
   61          } else {
   62            C = (char) pop(stack);
   63
   64            switch(IE[i]) {
   65              case ')':
   66                 if(C != '(')
   67                   V = 0;
   68                 break;
   69              case ']':
   70                 if(C != '[')
   71                   V = 0;
   72                 break;
   73              case '}':
   74                 if(C != '{')
   75                   V = 0;
   76                 break;
   77            }
   78          }
```

```
79          }
80          i++;
81      }
82
83      if(!emptystack(stack)) {
84        V = 0;
85      }
86
87      return V;
88  }
```

Referência ao ficheiro main.c

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

Funções

- int **main** ()

Documentação das funções

int main ()

```
    5           {
    6   system("clear");
    7
    8   char O;
    9
   10   printf("\tMenu\n\n1. Resolucao da Expressao\n2. Calculadora\n3. Sair\n");
   11
   12   printf("\nSelecione a Opcao: ");
   13   O = getchar();
   14
   15   freebuffer();
   16
   17   if(O == '1') {
   18     resofex();
   19   } else {
   20     if(O == '2') {
   21       calc();
   22     } else {
   23       if(O == '3') {
   24         system("clear");
   25         exit(0);
   26       } else {
   27         printf("\nOpcao Invalida\n\n(pressione a tecla Enter para retornar ao
menu)\n");
   28       }
   29     }
   30   }
   31
   32   while(O != '3') {
   33     getchar();
   34     system("clear");
   35
   36     printf("\tMenu\n\n1. Resolucao da Expressao\n2. Calculadora\n3. Sair\n");
   37
   38     printf("\nSelecione a Opcao: ");
   39     O = getchar();
   40
   41     freebuffer();
   42
   43     if(O == '1') {
   44       resofex();
   45     } else {
   46       if(O == '2') {
   47         calc();
   48       } else {
   49         if(O == '3') {
   50           system("clear");
   51           exit(0);
   52         } else {
   53           printf("\nOpcao Invalida\n\n(pressione a tecla Enter para retornar ao
menu)\n");
   54         }
   55       }
   56     }
   57   }
   58
   59   return 0;
```

```
60 }
```

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.
- 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

Retorna:

T_stack*: ponteiro para a estrutura de tipo pilha alocada

```
 5                       {
 6   T_stack* stack = (T_stack*) malloc(sizeof(T_stack));
 7
 8   stack->top = NULL;
 9
10   return stack;
11 }
```

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 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

```
75                                    {
76   if(!emptystack(stack)) {
77     T_element* now = stack->top;
78
79     while(now != NULL) {
80       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:

void: não retorna nenhum valor

```
89                                  {
90   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  }
```

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

```
41                              {
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

```
57                                  {
58    if(emptystack(stack)) {
59      printf("Pilha Vazia\n");
60    } else {
61      T_element* now = stack->top;
62
63      int i = 1;
64
65      while(now != NULL) {
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:

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 }
```

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

```
 5                      {
 6   T_stack* stack = (T_stack*) malloc(sizeof(T_stack));
 7
 8   stack->top = NULL;
 9
10   return stack;
11 }
```

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:

void: não retorna nenhum valor

```
 5              {
 6   system("clear");
 7
 8   T_stack* stack = allocstack();
 9
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
19   printf("\n-> ");
20   scanf("%s", S);
21   ctod(S);
22
```

```
23   freebuffer();
24
25   switch (S[0]) {
26     case '+':
27       if(S[1] == '!') {
28         sumr(stack);
29       } else {
30         sum(stack);
31       }
32       break;
33     case '-':
34       if(S[1] >= '0' && S[1] <= '9') {
35         F = atof(S);
36         push(stack, F);
37       } else {
38         if(S[1] == '!') {
39           subr(stack);
40         } else {
41           sub(stack);
42         }
43       }
44       break;
45     case '*':
46       if(S[1] == '!') {
47         mulr(stack);
48       } else {
49         mul(stack);
50       }
51       break;
52     case '/':
53       if(S[1] == '!') {
54         divsr(stack);
55       } else {
56         divs(stack);
57       }
58       break;
59     case 'C':
60       copy(stack);
61       break;
62     case 'c':
63       copy(stack);
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]) {
89       case '+':
90         if(S[1] == '!') {
91           sumr(stack);
92         } else {
93           sum(stack);
94         }
95         break;
96       case '-':
97         if(S[1] >= '0' && S[1] <= '9') {
98           F = atof(S);
99           push(stack, F);
```

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

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

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

void: não retorna nenhum valor

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

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

```
194                     k++;
195                     break;
196                   case '+':
197                     P = 0;
198                   case '-':
199                     P = 0;
200                   default:
201                     A = (float) O;
202                     push(stack, A);
203                     P = 0;
204                     break;
205                   }
206               }
207               push(stack, C);
208             }
209             break;
210           case '+':
211           case '-':
212             if(emptystack(stack)) {
213               C = (float) IE[i];
214               push(stack, C);
215             } else {
216               P = 1;
217               while(!emptystack(stack) && P) {
218                 C = (float) IE[i];
219                 O = (char) pop(stack);
220                 switch (O) {
221                   case '*':
222                   case '/':
223                   case '+':
224                   case '-':
225                     PE[j] = (float) O;
226                     SP[k] = j;
227                     j++;
228                     k++;
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++;
244             while(IE[i] >= '0' && IE[i] <= '9') {
245               i++;
246               if(IE[i] == '.') {
247                 i++;
248               }
249             }
250             i--;
251             break;
252         }
253       }
254   }
255
256   while(!emptystack(stack)) {
257     PE[j] = pop(stack);
258     SP[k] = j;
259     j++;
260     k++;
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
  315   if(emptystack(stack)) {
  316     printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu)\n");
  317     getchar();
  318   } else {
  319     if(stack->top->next == NULL) {
  320       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  321       getchar();
  322     } else {
  323       F = (int) pop(stack);
  324       A = pop(stack);
  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:

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

Retorna:

void: não retorna nenhum valor

```
  269                             {
  270   float F, A;
```

```
  271
  272   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);
  281       A = pop(stack);
  282       F = F / A;
  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 {
  296     if(stack->top->next == NULL) {
  297       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  298       getchar();
  299     } else {
  300       F = pop(stack);
  301       do {
  302         A = pop(stack);
  303         F = F / A;
  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

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

```
 75                                      {
 76   if(!emptystack(stack)) {
 77     T_element* now = stack->top;
 78
 79     while(now != NULL) {
 80       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:

void: não retorna nenhum valor

```
 89                                    {
 90   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

```
  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");
  232     getchar();
  233   } else {
  234     if(stack->top->next == NULL) {
  235       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  236       getchar();
  237     } else {
  238       F = pop(stack);
  239       A = pop(stack);
  240       F = F * A;
  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

```
  247                           {
  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("\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 {
  260         A = pop(stack);
  261         F = F * A;
  262       } while(!emptystack(stack));
  263       push(stack, F);
  264     }
  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

```
41                              {
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 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
271   printf("\nForma Posfixa: ");
272
273   for(i = 0, j = 1; i < SP[0]; i++) {
274     if(i == SP[j]) {
275       O = (char) PE[i];
276       printf(" %c ", O);
277       j++;
278     } else {
279       printf(" %.2f ", PE[i]);
280     }
281   }
282
283   printf("\n");
284 }
```

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

```
57                                  {
58   if(emptystack(stack)) {
59     printf("Pilha Vazia\n");
60   } else {
61     T element* now = stack->top;
62
63     int i = 1;
64
65     while(now != NULL) {
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:

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

```
 5                  {
 6   system("clear");
 7
 8   T stack* stack = allocstack();
 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
  17   N = 0;
  18
  19   while(IE[N] != '\0') {
  20     N++;
  21   }
  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     ctod(IE);
  32     convertexpression(stack, IE, PE, SP, N);
  33     printexpression(PE, SP);
  34     R = calcexpression(stack, PE, SP);
  35     printf("\nResultado: %.2f\n", R);
  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   } else {
 192     if(stack->top->next == NULL) {
 193       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
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     }
 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:

*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)) {
  209     printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla Enter
para retornar ao menu)\n");
  210     getchar();
  211   } else {
  212     if(stack->top->next == NULL) {
  213       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  214       getchar();
  215     } else {
  216       F = pop(stack);
  217       do {
  218         A = pop(stack);
  219         F = F - A;
  220       } while(!emptystack(stack));
  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 na qual operandos e operadores são empilhados

Retorna:

void: não retorna nenhum valor

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

void: não retorna nenhum valor

```
  163                             {
  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();
  169   } else {
  170     if(stack->top->next == NULL) {
  171       printf("\nQuantidade de operandos insuficiente\n\n(pressione a tecla
Enter para retornar ao menu)\n");
  172       getchar();
  173     } else {
  174       F = pop(stack);
  175       do {
  176         A = pop(stack);
  177         F = F + A;
  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
   46   int i = 0, V = 1;
   47
   48   while(IE[i] != '\0') {
   49
   50     if(IE[i] == '(' ||
   51        IE[i] == '[' ||
   52        IE[i] == '{') {
   53          push(stack, (int) IE[i]);
   54        }
   55
   56     if(IE[i] == ')' ||
   57        IE[i] == ']' ||
   58        IE[i] == '}') {
   59          if(emptystack(stack)) {
   60            V = 0;
   61          } else {
   62            C = (char) pop(stack);
   63
   64            switch(IE[i]) {
   65              case ')':
   66                 if(C != '(')
   67                   V = 0;
   68                 break;
   69              case ']':
   70                 if(C != '[')
   71                   V = 0;
   72                 break;
   73              case '}':
   74                 if(C != '{')
   75                   V = 0;
   76                 break;
```

```
77              }
78            }
79          }
80          i++;
81    }
82
83    if(!emptystack(stack)) {
84      V = 0;
85    }
86
87    return V;
88  }
```

Índice

INDEX