

# BrickOP

## Proyecto final

Joshua S. González Torres      *Elementos de Ciencias de Computación*

17 de diciembre de 2021

### 1. Propuesta

La propuesta original de este proyecto consistía en la creación de un programa que leyera un archivo de texto que contuviera una colección de operaciones normalmente utilizadas para enseñar jerarquía de operaciones (expresiones meramente numéricas, sin álgebra) e imprimiera en un archivo nuevo estas mismas líneas pero con sus respectivos resultados al final. Esta última parte de la idea cambió durante la programación del mismo: ahora el programa imprimiría en el nuevo archivo todo el procedimiento realizado para obtener el resultado.

La propuesta fue aceptada y el producto final funciona tal y como se planteó. El código fuente completo, al igual que la forma en que se ha construido, es consultable en mi [GitHub](#).

### 2. Funcionamiento general

El archivo ejecutable (.exe) debe colocarse en un mismo directorio que un archivo de texto llamado "operaciones.txt". El archivo de texto es el que contiene las operaciones a realizar; éste debe cumplir lo siguiente para el funcionamiento óptimo del programa:

- Cada operación debe ocupar exactamente una línea.
- Los operadores (+, -, \*, /, ^)<sup>1</sup> deben estar separados de los números y de los paréntesis por exactamente un espacio:  $1 + 2 * 3 * (5 + (10 ^ 2))$ . Nótese que los paréntesis pueden encontrarse directamente junto a los números y otros paréntesis, por lo que espaciarlos es innecesario.
- Al final del archivo, debe haber una línea conteniendo únicamente un espacio.

Luego, puede ejecutarse el programa y después de que este haya concluido, puede esperar ver un archivo de texto de nombre "resultados.txt" con los procedimientos y resultados de sus operaciones.

Como demostración, se ha escrito el archivo "operaciones.txt" como sigue:

---

<sup>1</sup>No hay actualmente operador para denotar raíces, se deberán usar potencias fraccionarias.

---

`operaciones.txt`

---

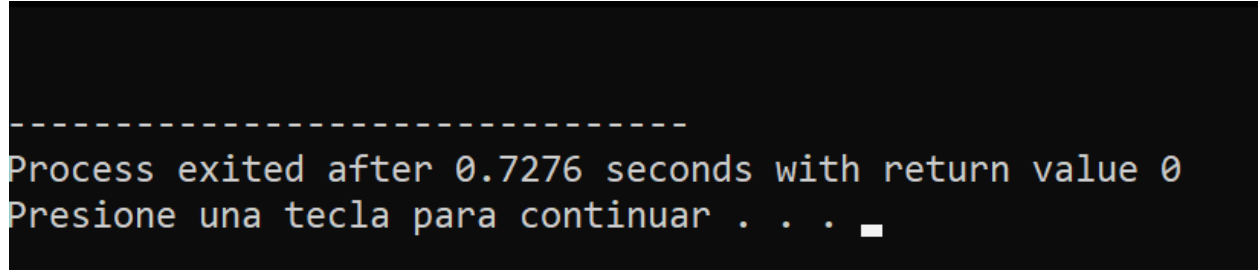
```
1 + 2 * 3
(3 + 1) * 2 + 2 * (3 - 1)
12 / 4 * 6 / 3
4 / 2 ^ 2 - 4
3 + 4 / 2 - 4
1 + 2 - 3 * 3 + 4 ^ 2
1 - 2 * 3 * 4
7 - 1 * 0 + 3 / 3
2 * (4 * (7 + 4 * 6) - 3 * 32)
3 * (5 - 1) ^ 2 - 14 / 2
27 + 3 * 5 - 16
27 + 3 - 45 / 5 + 16
(2 * 4 + 12) * (6 - 4)
3 * 9 + (6 + 5 - 3) - 12 / 4
2 + 5 * (2 * 3) ^ 3
440 - (30 + 6 * (19 - 12))
2 * (4 * (7 + 4 * (5 * 3 - 9)) - 3 * (40 - 8))
```

---

Luego, se ha dejado correr el programa:

`main.c`

```
1  #include "pemd.h"
2  #include <locale.h>
3
4  int main(int argc, char *argv[]) {
5      setlocale(LC_ALL, "");
6
7      solveLineByLine("operaciones.txt");
8
9      getchar();
10     return 0;
11 }
```



-----  
Process exited after 0.7276 seconds with return value 0  
Presione una tecla para continuar . . .

Y ha devuelto el archivo "resultados.txt":

---

`resultados.txt`

---

```
1 + 2 * 3
```

$1.000000 + 6.000000$   
 $7.000000$

$(3 + 1) * 2 + 2 * (3 - 1)$   
 $(3 + 1) * 2 + 2 * (3.000000 + -1.000000)$   
 $(3 + 1) * 2 + 2 * (2.000000)$   
 $(3 + 1) * 2 + 2 * 2.000000$   
 $(4.000000) * 2 + 2 * 2.000000$   
 $4.000000 * 2 + 2 * 2.000000$   
 $8.000000 + 4.000000$   
 $12.000000$

$12 / 4 * 6 / 3$   
 $12.000000 * 0.250000 * 6.000000 * 0.333333$   
 $6.000000$

$4 / 2 ^ 2 - 4$   
 $4.000000 * 0.500000 ^ 2.000000 + -4.000000$   
 $4.000000 * 0.250000 + -4.000000$   
 $1.000000 + -4.000000$   
 $-3.000000$

$3 + 4 / 2 - 4$   
 $3.000000 + 4.000000 * 0.500000 + -4.000000$   
 $3.000000 + 2.000000 + -4.000000$   
 $1.000000$

$1 + 2 - 3 * 3 + 4 ^ 2$   
 $1.000000 + 2.000000 + -3.000000 * 3.000000 + 4.000000 ^ 2.000000$   
 $1.000000 + 2.000000 + -3.000000 * 3.000000 + 16.000000$   
 $1.000000 + 2.000000 + -9.000000 + 16.000000$   
 $10.000000$

$1 - 2 * 3 * 4$   
 $1.000000 + -2.000000 * 3.000000 * 4.000000$   
 $1.000000 + -24.000000$   
 $-23.000000$

$7 - 1 * 0 + 3 / 3$   
 $7.000000 + -1.000000 * 0.000000 + 3.000000 * 0.333333$   
 $7.000000 + 1.000000$   
 $8.000000$

$2 * (4 * (7 + 4 * 6) - 3 * 32)$   
 $2 * (4 * (7.000000 + 24.000000) - 3 * 32)$   
 $2 * (4 * (31.000000) - 3 * 32)$

$2 * (4 * 31.000000 - 3 * 32)$   
 $2 * (4.000000 * 31.000000 + -3.000000 * 32.000000)$   
 $2 * (124.000000 + -96.000000)$   
 $2 * (28.000000)$   
 $2 * 28.000000$   
 $56.000000$

$3 * (5 - 1) ^ 2 - 14 / 2$   
 $3 * (5.000000 + -1.000000) ^ 2 - 14 / 2$   
 $3 * (4.000000) ^ 2 - 14 / 2$   
 $3 * 4.000000 ^ 2 - 14 / 2$   
 $3.000000 * 4.000000 ^ 2.000000 + -14.000000 * 0.500000$   
 $3.000000 * 16.000000 + -14.000000 * 0.500000$   
 $48.000000 + -7.000000$   
 $41.000000$

$27 + 3 * 5 - 16$   
 $27.000000 + 3.000000 * 5.000000 + -16.000000$   
 $27.000000 + 15.000000 + -16.000000$   
 $26.000000$

$27 + 3 - 45 / 5 + 16$   
 $27.000000 + 3.000000 + -45.000000 * 0.200000 + 16.000000$   
 $27.000000 + 3.000000 + -9.000000 + 16.000000$   
 $37.000000$

$(2 * 4 + 12) * (6 - 4)$   
 $(2 * 4 + 12) * (6.000000 + -4.000000)$   
 $(2 * 4 + 12) * (2.000000)$   
 $(2 * 4 + 12) * 2.000000$   
 $(8.000000 + 12.000000) * 2.000000$   
 $(20.000000) * 2.000000$   
 $20.000000 * 2.000000$   
 $40.000000$

$3 * 9 + (6 + 5 - 3) - 12 / 4$   
 $3 * 9 + (6.000000 + 5.000000 + -3.000000) - 12 / 4$   
 $3 * 9 + (8.000000) - 12 / 4$   
 $3 * 9 + 8.000000 - 12 / 4$   
 $3.000000 * 9.000000 + 8.000000 + -12.000000 * 0.250000$   
 $27.000000 + 8.000000 + -3.000000$   
 $32.000000$

$2 + 5 * (2 * 3) ^ 3$   
 $2 + 5 * (6.000000) ^ 3$   
 $2 + 5 * 6.000000 ^ 3$

```
2.000000 + 5.000000 * 216.000000
2.000000 + 1080.000000
1082.000000
```

```
440 - (30 + 6 * (19 - 12))
440 - (30 + 6 * (19.000000 + -12.000000))
440 - (30 + 6 * (7.000000))
440 - (30 + 6 * 7.000000)
440 - (30.000000 + 42.000000)
440 - (72.000000)
440 - 72.000000
440.000000 + -72.000000
368.000000
```

```
2 * (4 * (7 + 4 * (5 * 3 - 9)) - 3 * (40 - 8))
2 * (4 * (7 + 4 * (5 * 3 - 9)) - 3 * (40.000000 + -8.000000))
2 * (4 * (7 + 4 * (5 * 3 - 9)) - 3 * (32.000000))
2 * (4 * (7 + 4 * (5 * 3 - 9)) - 3 * 32.000000)
2 * (4 * (7 + 4 * (5.000000 * 3.000000 + -9.000000)) - 3 * 32.000000)
2 * (4 * (7 + 4 * (15.000000 + -9.000000)) - 3 * 32.000000)
2 * (4 * (7 + 4 * (6.000000)) - 3 * 32.000000)
2 * (4 * (7 + 4 * 6.000000) - 3 * 32.000000)
2 * (4 * (7.000000 + 24.000000) - 3 * 32.000000)
2 * (4 * (31.000000) - 3 * 32.000000)
2 * (4 * 31.000000 - 3 * 32.000000)
2 * (4.000000 * 31.000000 + -3.000000 * 32.000000)
2 * (124.000000 + -96.000000)
2 * (28.000000)
2 * 28.000000
56.000000
```

---

Se motiva al lector a probar el programa por sí mismo con operaciones tan variadas como le parezca.

### 3. Funcionamiento interno

El programa consiste de dos librerías: `pemdas.h` y `printer.h`. Ambas librerías se describen a continuación.

### 3.1. La librería pmdas.h

La librería principal del programa, contiene las funciones que hacen el trabajo numérico. Contiene en total 5 funciones.

pmdas.c

```
179 int countNumbers(FILE* stream){
180     int counter = 0;
181     double aux;
182     while(!feof(stream)){
183         counter++;
184         fscanf(stream, "%lf %c ", &aux);
185     }
186     rewind(stream);
187     return counter;
188 }
```

Entre las funciones más básicas, se encuentra la que calcula la cantidad de números que hay en un archivo que contenga una operación sin paréntesis. Esto resulta útil para asignar la memoria dinámica de los vectores `double* numbers` y `char* operators` de la función `pmdasSolve()`.

pmdas.c

```
118 double pmdasSolve(FILE* stream, FILE* destination){
119     int numNumbers = countNumbers(stream);
120     double* numbers = malloc(numNumbers * sizeof(double)), final;
121     char* operators = malloc(numNumbers * sizeof(char)), done = 0;
122     int i;
123     for(i=0; i<numNumbers; i++){
124         fscanf(stream, "%lf %c", &numbers[i], &operators[i]);
125     }
126
127     for(i=0; i<numNumbers; i++){
128         if(operators[i] == '-') {
129             numbers[i+1] *= -1;
130             operators[i] = '+';
131             done = 1;
132         }
133         if(operators[i] == '/') {
134             numbers[i+1] = 1/numbers[i+1];
135             operators[i] = '*';
136             done = 1;
137         }
138     }
139     if(done){
140         printCompleteLineToFile(numbers, operators, numNumbers, destination);
141     }
```

```
142     done = 0;
143
144     for(i=0; i<numNumbers; i++){
145         if(operators[i] == '^'){
146             numbers[i+1] = pow(numbers[i], numbers[i+1]);
147             operators[i] = '*';
148             numbers[i] = 1;
149             done = 1;
150         }
151     }
152     if(done){
153         printStepToFile(numbers, operators, numNumbers, destination);
154     }
155     done = 0;
156
157     for(i=0; i<numNumbers; i++){
158         if(operators[i] == '*'){
159             numbers[i+1] *= numbers[i];
160             operators[i] = '+';
161             numbers[i] = 0;
162             done = 1;
163         }
164     }
165     if(done){
166         printStepToFile(numbers, operators, numNumbers, destination);
167     }
168
169     final = sumaVector(numbers, numNumbers);
170     if(final != numbers[numNumbers - 1]){
171         fprintf(destination, "%lf\n", final);
172     }
173     free(numbers);
174     free(operators);
175
176     return final;
177 }
```

Después de contar cuántos números se van a considerar en la operación sin paréntesis, se asigna memoria dinámicamente a dos arreglos: el de los operadores y el de los números. Se itera a través del archivo que contiene la operación a realizar, guardando los números y los operadores en sus vectores respectivos. Luego, se itera a través de ambos arreglos para cambiar las restas por sumas con inversos aditivos y las divisiones como productos con inversos multiplicativos; esto esclarece los siguientes pasos. Si alguno de estos procesos se realizó, entonces se imprime la línea al archivo de destino. Se itera una segunda vez, ahora para realizar las potencias; para conservar la simplicidad de las iteraciones, se cambian las potencias por multiplicaciones por 1. Si este proceso se realizó,

entonces se imprime el paso al archivo de destino. Una tercera vez, se itera; las multiplicaciones se realizan donde es debido y, de nuevo para conservar la simplicidad, se cambian por sumando 0. Una vez más, el paso se imprime si es que se realizó alguna multiplicación. Basta ahora con sumar todos los elementos del vector para obtener el resultado de la operación. La función utilizada para sumar es una función recursiva bastante sencilla (vista en clase) que puede omitirse durante este reporte. Sucede a veces que la suma es efectivamente el último elemento del vector `numbers`, por lo que se comprueba si sí es así para no imprimir dos veces un mismo número en el archivo destino. Finalmente, se libera la memoria solicitada y el resultado de la operación se devuelve por la izquierda.

pemdas.c

```
3 void clearParen(char filename[MAX_FILENAME_SIZE], FILE* destination){
4     FILE* stream = fopen(filename, "rb+");
5     char* str = malloc(MAX_OPERATION_SIZE * sizeof(char));
6     fgets(str, MAX_OPERATION_SIZE, stream);
7     fclose(stream);
8     int i = 0, parenFirstPos = -1;
9     while(str[i]){
10         if(str[i] == '('){
11             parenFirstPos = i;
12         }
13         i++;
14     }
15
16     if(parenFirstPos < 0){
17         free(str);
18         fclose(stream);
19         return;
20     }
21
22     int parenLastPos = -1;
23     i = parenFirstPos + 2;
24     while(str[i] != ')'){
25         i++;
26     }
27     parenLastPos = i;
28
29     char* substr = malloc((parenLastPos-parenFirstPos+1)*sizeof(char));
30     for(i=0; i<parenLastPos-parenFirstPos-1; i++){
31         substr[i] = str[i+parenFirstPos+1];
32     }
33     substr[parenLastPos-parenFirstPos-1] = '\n';
34     substr[parenLastPos-parenFirstPos] = 0;
35     FILE* subOperation = fopen("subOp.bin", "wb+");
36     fputs(substr, subOperation);
37     free(substr);
```



```
38  fclose(subOperation);
39  subOperation = fopen("subOp.bin", "rb+");
40  double result = pmdasSolve(subOperation, subOperation);
41  rewind(subOperation);
42
43  char* head = malloc(MAX_OPERATION_SIZE * sizeof(char));
44  char* subStep = malloc(MAX_OPERATION_SIZE * sizeof(char));
45  char* tail = malloc(MAX_OPERATION_SIZE * sizeof(char));
46
47  for(i=0; i<parenFirstPos; i++){
48      head[i] = str[i];
49  }
50  head[i] = 0;
51
52  i = 0;
53  while(1){
54      tail[i] = str[i+parenLastPos+1];
55      if(tail[i] == 0){
56          break;
57      }
58      i++;
59  }
60  fgets(subStep, MAX_OPERATION_SIZE, subOperation);
61  while(fgets(subStep, MAX_OPERATION_SIZE, subOperation) != NULL){
62      fputs(head, destination);
63      fprintf(destination, "(");
64
65      fputs(subStep, destination);
66      fseek(destination, -2, SEEK_END);
67      fprintf(destination, ")");
68
69      fputs(tail, destination);
70  }
71  fclose(subOperation);
72  remove("subOp.bin");
73
74  fputs(head, destination);
75  fprintf(destination, "%lf", result);
76  fputs(tail, destination);
77
78  stream = fopen(filename, "wb");
79  fputs(head, stream);
80  fprintf(stream, "%lf", result);
81  fputs(tail, stream);
82  fclose(stream);
```

```
83     free(str); free(subStep);
84     free(head); free(tail);
85
86     clearParen(filename, destination);
87 }
```

La función `clearParen()` es posiblemente la más complicada de entre las funciones que componen a esta librería.

Primero, se abre el archivo de donde se quieren limpiar los paréntesis, se obtiene su primera línea y se revisa iterativamente si alguno de los caracteres extraídos es un paréntesis. Si no se encuentra alguno, entonces la función retorna sin mayor problema. Si se encuentra alguno, entonces se guarda el de la última posición donde se encontró uno y se busca iterativamente su par. Ahora, en una nueva cadena, se guarda lo que está contenido entre los paréntesis encontrados. Luego, se crea un nuevo archivo "subOp.bin" donde esta cadena (la cual no contiene paréntesis alguno) se guarda y se resuelve usando la función `pemdasSolve()`. Los pasos se imprimen en el mismo archivo de donde se extrajo la operación.

Se generan ahora tres cadenas nuevas: `head` que guarda todos los caracteres antes del paréntesis que se encontró al inicio, `subStep` que guardará los pasos que se guardaron en el archivo `subOp.bin`, y `tail`, que guardará todos los caracteres después del paréntesis que se encontró al inicio. Comenzando ahora por el segundo paso (el primero es solamente la operación sin realizar), se colocan en el archivo de destino `head`, `subStep` y `tail` en ese orden para mostrar todos los pasos realizados dentro del paréntesis. Finalmente, se escribe en el archivo cuyo nombre se recibió la operación sin el par de paréntesis que se detectaron al inicio, se libera memoria y se llama nuevamente a la función; esto último con el objetivo de limpiar recursivamente al archivo recibido de todos los pares de paréntesis que hay en él.

pemdas.c

```
89 void solveLineByLine(char filename[MAX_FILENAME_SIZE]){
90     FILE* whole = fopen(filename, "r");
91     if(!whole){
92         printf("Error de archivo."); return;
93     }
94     FILE* results = fopen("resultados.txt", "w");
95     if(!results){
96         printf("Error en creacion de archivo de resultados"); return;
97     }
98
99     while(!feof(whole)){
100         char* str = malloc(MAX_OPERATION_SIZE * sizeof(char));
101         fgets(str, MAX_OPERATION_SIZE, whole);
102         FILE* oneLine = fopen("online.bin", "wb");
103         fputs(str, oneLine);
104         fclose(oneLine);
105         fputs(str, results);
106         free(str);
```

```

107     clearParen("oneline.bin", results);
108     oneLine = fopen("oneline.bin", "rb+");
109     pmdasSolve(oneLine, results);
110     fclose(oneLine);
111     fprintf(results, "\n");
112     remove("oneline.bin");
113 }
114 fclose(results);
115 fclose(whole);
116 }

```

Esta última función unifica las anteriores y permite que el programa realice lo establecido en la propuesta. Recibe el nombre del archivo con todas las operaciones a realizar y se crea el archivo de resultados. Luego, se toma línea a línea del archivo y se guardan en un auxiliar al cual se le limpian los paréntesis y se resuelve con las funciones `clearParen()` y `pmdasSolve()` respectivamente. Los pasos de ambas funciones se imprimen en el archivo resultados y se elimina el archivo auxiliar.

### 3.2. La librería `printer.h`

Esta funciona como librería auxiliar y contiene únicamente dos funciones que se utilizan durante la ejecución del programa (las otras fueron creadas con el propósito de probar el programa durante sus fases preliminares).

`printer.c`

```

3 void printCompleteLineToFile(double* numbers, char* operators, int size, FILE* destination)
4 {
5     for(i=0; i<size-1; i++){
6         fprintf(destination, "%lf %c ", numbers[i], operators[i]);
7     }
8     fprintf(destination, "%lf\n", numbers[i]);
9 }

```

Como su nombre lo indica, esta función imprime todo el contenido de los arreglos de números y operadores que conforman la línea tomada desde algún archivo y los imprime en el archivo de destino.

`printer.c`

```

11 void printStepToFile(double* numbers, char* operators, int size, FILE* destination){
12     int i;
13     for(i=0; i<size-1; i++){
14         if((numbers[i] == 1 && operators[i] == '*') || (numbers[i] == 0 && operators[i] == '+'))
15             continue;
16     }
17     fprintf(destination, "%lf %c ", numbers[i], operators[i]);

```

```
18     }  
19     fprintf(destination, "%lf\n", numbers[i]);  
20 }
```

A diferencia de la función anterior, ésta contiene una restricción para las operaciones redundantes: multiplicar por 1 y sumar 0, omitiéndolas para que la línea que se imprime sea lo más concisa posible.