Universidad de Buenos Aires

Facultad de Ingeniería

# Algoritmos y Programación I (95.11)

Curso: 01-Ing. Cardozo

## Caso de estudio n.º 2 - Altas, bajas y modificaciones

**Alumnos:**

| | | |
|---|---|---|
| OCHAGAVIA, Lara | 100637 | lari.ochagavia13@gmail.com |
| PINTOS, Gastón Maximiliano | 99711 | massipintos@gmail.com |


Fecha de entrega: 30 de mayo de 2019

Fecha de primer reentrega: 13 de junio de 2019

# Índice

# 1. Introducción

En este informe se presenta el desarrollo de aplicativos de consola con comandos de línea de órdenes y escritos en lenguaje C, que permiten implementar las operaciones de altas, bajas y modificaciones de un inventario almacenado en un archivo.

A lo largo del desarrollo del informe se explicitan las consideraciones que se tuvieron en cuenta para la elaboración del trabajo práctico. El tipo de archivo elegido fue de tipo CSV (*comma-separated-values*) y, a partir de dicho archivo, se desarrollaron los programas individuales para generar un nuevo archivo actualizado según se invoque el programa que realiza altas, bajas o modifique los datos originales.

Para la realización de las acciones previamente enumeradas, cada operación cuenta con un archivo conteniendo el código de implementación de las función y un homónimo como archivo de inventario. El archivo de inventario se conforma por registros con los campos de información: ID, la descripción de un recurso y el tiempo de uso o tráfico del recurso. Los registros se presentan ordenados de forma ascendente por el campo ID. Los archivos individuales correspondientes a las operaciones a realizar siguen, salvo alteraciones particulares, el mismo formato.

En los apéndices del informe se adjuntan los códigos y el contenido de los archivos de encabezado correspondientes a cada programa. Estos códigos están adecuadamente modularizados con el fin de reutilizar componentes de software que los programas tienen en común.

Por último, en la conclusión del trabajo, se especifican las dificultades que se presentaron al momento de la elaboración del código, los errores que se cometieron y una descripción del resultado final del trabajo realizado.

# 2. Desarrollo

Para comenzar con la elaboración del trabajo, se partió de la base del archivo de inventario original. El inventario consiste en registros compuestos por campos que se presentan separados por el delimitador barra vertical. Todos los campos del inventario contienen datos según se corresponda. Lo mismo ocurre en el archivo de altas, el cual tiene el mismo formato que el inventario y tiene registros con todos los campos completos con los datos que correspondan a cada posición. No ocurre lo mismo con los archivos de bajas y modificaciones. El primero de los mencionados presenta únicamente el primer campo completo, mientras que los últimos dos campos no existen. El archivo de modificaciones contiene sólo los datos del ID y del tiempo de uso.

Lo primero que se realizó como común para todos los programas fue la apertura de los archivos y la creación de un archivo de salida temporal que reemplazará al archivo de inventario original. Todos los archivos se abrieron en modo lectura, salvo el último mencionado, sobre el cual se desea escribir, por lo que fue abierto y creado en modo escritura.

La lectura de los archivos consistió en identificar el fin de cada línea, es decir, el fin de cada registro. Cada registro contiene, separado por los delimitadores, los tres campos que conformarán la estructura del inventario. Cada linea es convertida a una cadena de caracteres, para posteriormente, ser transformada en cadenas de caracteres más cortas, determinadas por los delimitadores. Estas cadenas más pequeñas se denominan *tokens*. En resumen, cada linea de cada archivo, se convierte en tres cadenas de caracteres.

A medida que se crea cada *token*, se cargan las cadenas a un arreglo, que pasan los datos a la estructura de inventario una vez que las cadenas fueron convertidas al tipo de dato correspondiente a cada campo: el identificador se conforma por números naturales (*size_t*) y el tiempo de uso por números reales (*float*). El campo de la descripción del recurso es definido como una cadena de caracteres, por lo que no hace falta modificar el tipo de dato que se recibe.

Para el caso del programa de altas, la comparación a realizar entre los identificadores de la estructura proveniente del inventario original y del archivo de altas consiste en analizar cuál de los dos identificadoras es menor, e imprimir el que lo sea en el archivo de salida temporal. El resultado es un archivo de texto que contiene

los registros del inventario original, mas los registros del archivo de altas, ordenados según su identificador ascendente, con los campos separados por el delimitador presente en los archivos originales.

Con el programa de bajas, la comparación a realizar entre los identificadores de la estructura proveniente del inventario original y del archivo de bajas consiste en encontrar un dato repetido. El archivo de bajas no cuenta con la información de la descripción del recurso ni el tiempo de use ya que es información irrelevante a los fines de eliminar un registro. Mientras que los identificadores a comparar sean diferentes, se copia el registro completo del archivo de inventario en el archivo temporal de salida.

Por último, para el programa de modificaciones, se comparan los identificadores de la estructura proveniente del inventario original y del archivo de modificaciones buscando coincidencia entre ellos. Cuando se produce la igualdad, se escribe en el campo del tiempo de uso de la estructura del inventario el valor original sumado al valor presente en el campo del tiempo de uso en la estructura del inventario correspondiente al archivo de modificaciones. De esta forma, en el archivo de salida, se imprime el registro del archivo de inventario, cuando este fue modificado. Por lo tanto, contendrá una lista en principio similar a la del inventario original, pero con el tiempo de uso de los registros presentes en el archivo de modificaciones alterados.

Cada función implementada para la realización de cada programa se validó al momento de ser invocada. Las funciones cuentan con validaciones internas para asegurar su correcto funcionamiento, adaptadas a las necesidades de cada programa. Si un error se detecta, se informa al usuario cuál es el problema, mediante la implementación de un diccionario de errores.

Al estar todas las funciones validadas, el siguiente paso consiste en cerrar los archivos abiertos, con el detalle de que se elimina el archivo de inventario original, y el archivo de salida temporal recibe ahora el nombre del reciente eliminado.

# 3. Conclusión

Los archivos de texto en formato CSV son eficientes en el espacio que ocupan los campos y verificables al conocer como están separados los campos. Sin embargo, son ineficientes en el uso de memoria dinámica ya que la misma puede generar errores en su implementación y precisan entonces de un sistema operativo. Por otro lado, sus campos son de longitud variable por lo que dificulta separarlos y solo pueden reconocerse utilizando su delimitador. En el caso de utilizar campos delimitados el código debe ser alterado.

En cuanto a los archivos de texto con formato de ancho fijo, este formato es eficiente para la detección de errores de archivos corruptos y su separación por campos se facilita con funciones de manejo de memoria (*memcpy*) al conocer la longitud de cada campo. No obstante, suelen ser archivos extensos en donde los campos se completan para lograr la longitud deseada haciendo que su manejo sea ineficiente.

Por ultimo, los archivos binarios son eficiente en extensión y se pueden detectar fallas de archivos corruptos. Para el manejo de archivos se pueden utilizar las funciones *fread* y *fwrite* lo que hace su procesamiento más simple. Aunque para poder visualizar su contenido se debe realizar un programa de conversión lo que significa tiempo de procesamiento.

Los archivos de textos son portables, mientras que los binarios no lo son debido al manejo de estructuras y al formato *Big Endian*.
Se eligieron archivos de texto para trabajar a lo largo del trabajo debido a su simplicidad frente a la visibilización de su contenido por parte de cualquier usuario. En los archivos de texto, los resultados del programa son verificables sin la necesidad de extender el código. Además, el uso de archivos de tipo CSV es una práctica común, por lo que se consideró adecuada su elección para familiarizarse con su manejo.

Las principales dificultades se presentaron con el manejo de estructuras, al momento de realizar las comparaciones y de guardar los datos a utilizar en memoria. Las comparaciones de los campos de las estructuras, al variar en cada programa lo comparado, requerían de código diferente para cada operación. La identificación de los cambios a realizar respecto de un código a otro consumió gran parte del tiempo de elaboración.

Al creer finalizado el trabajo, se observó que se cometió el error de haber cargado a memoria la totalidad de los registros, lo cual atenta con la efectividad y la robustez del programa. El correcto funcionamiento del programa dependería de la escasa cantidad de registros presentes en cada archivo. Se logró solucionar dicho problema mediante las comparaciones descriptas en el desarrollo de manera secuencial y se carga un registro a la vez.

En un principio, se desarrolló el mismo para la implementación del programa de altas. Para desarrollar el programa de bajas, y con el de modificaciones, se reutilizó el código en su gran mayoría, salvando las diferencias que se presentan en la comparación a realizar por cada programa, lo cual corresponde a una función propia para cada programa. Esto se asumió como una ventaja del uso de la de modularización, con el propósito de que la reutilización del código ahorre recursos de software y tiempo de implementación.

Con este trabajo se profundizó el conocimiento de punteros, archivos y estructuras, ya que en cada función implementada se requiere suficiente abstracción para entender cómo esta función interactúa con las otras. Se necesitó la concentración suficiente para relacionar las funciones, sin confundir los niveles de punteros en cada argumento y sin olvidar de qué manera se manejó cada parámetro.

# 4. Apéndice I: Código

## 4.1. Altas, bajas y modificaciones

### 4.1.1. main_altas.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "main.h"
#include "types.h"
#include "error.h"
#include "record.h"
#include "abm.h"


int main (int argc, char * argv[])
{
        FILE * stock_file;
        FILE * crud_file;
        FILE * output_file;

        status_t status;

        if((status = validate_args( argv, argc)) != OK)
        {
                print_error(status);
                return status;
        }

        if((stock_file = fopen (argv[CMD_ARG_STOCK_FILE_POSITION ], "rt")) == NULL)
        {
                status=ERROR_CORRUPT_FILE;
                print_error(status);
                return status;
        }

        if((crud_file = fopen (argv[CMD_ARG_CRUD_FILE_POSITION], "rt")) == NULL)
        {
                fclose(stock_file);
                status=ERROR_CORRUPT_FILE;
                print_error(status);
                return status;
        }

        if((output_file = fopen("stocklist_update.txt", "wt")) == NULL)
        {
                fclose(stock_file);
                fclose(crud_file);
                status=ERROR_CORRUPT_FILE;
```

```
46                  print_error(status);
47                  return status;
48          }
49
50          if((status = compare_records_uploads(stock_file,crud_file,output_file)) != OK
                )
51          {
52                  fclose(stock_file);
53                  fclose(crud_file);
54                  fclose(output_file);
55
56                  print_error(status);
57                  return status;
58          }
59
60          fclose(stock_file);
61          fclose(crud_file);
62
63          if(fclose(output_file) == EOF){
64                  fclose(output_file);
65                  status=ERROR_WRITING_FILE;
66                  print_error(status);
67                  return status;
68          }
69
70          remove(argv[CMD_ARG_STOCK_FILE_POSITION ]);
71
72          rename(output_file, "stocklist_update.txt");
73
74          return OK;
75  }
```

#### 4.1.2. main_bajas.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include "main.h"
6   #include "types.h"
7   #include "error.h"
8   #include "record.h"
9   #include "abm.h"
10
11  int main (int argc, char * argv[])
12  {
13          FILE * stock_file;
14          FILE * crud_file;
15          FILE * output_file;
```

```
16
17          status_t status;
18
19          if((status = validate_args( argv, argc)) != OK)
20          {
21                  print_error(status);
22                  return status;
23          }
24
25          if((stock_file = fopen (argv[CMD_ARG_STOCK_FILE_POSITION ], "rt")) == NULL)
26          {
27                  status=ERROR_CORRUPT_FILE;
28                  print_error(status);
29                  return status;
30          }
31
32          if((crud_file = fopen (argv[CMD_ARG_CRUD_FILE_POSITION], "rt")) == NULL)
33          {
34                  fclose(stock_file);
35                  status=ERROR_CORRUPT_FILE;
36                  print_error(status);
37                  return status;
38          }
39
40          if((output_file = fopen("stocklist_update.txt", "wt")) == NULL)
41          {
42                  fclose(stock_file);
43                  fclose(crud_file);
44                  status=ERROR_CORRUPT_FILE;
45                  print_error(status);
46                  return status;
47          }
48
49          if((status = compare_records_deletions(stock_file,crud_file,output_file)) !=
                OK)
50          {
51                  fclose(stock_file);
52                  fclose(crud_file);
53                  fclose(output_file);
54
55                  print_error(status);
56                  return status;
57          }
58
59          fclose(stock_file);
60          fclose(crud_file);
61
62          if(fclose(output_file) == EOF){
63                  fclose(output_file);
64                  status=ERROR_WRITING_FILE;
65                  print_error(status);
```

```
66                  return status;
67          }
68
69          remove(argv[CMD_ARG_STOCK_FILE_POSITION ]);
70
71          rename(output_file, "stocklist_update.txt");
72
73          return OK;
74  }
```

#### 4.1.3. main_modificaciones.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include "main.h"
6   #include "types.h"
7   #include "error.h"
8   #include "record.h"
9   #include "abm.h"
10
11  int main (int argc, char * argv[])
12  {
13          FILE * stock_file;
14          FILE * crud_file;
15          FILE * output_file;
16
17          status_t status;
18
19          if((status = validate_args( argv, argc)) != OK)
20          {
21                  print_error(status);
22                  return status;
23          }
24
25          if((stock_file = fopen (argv[CMD_ARG_STOCK_FILE_POSITION ], "rt")) == NULL)
26          {
27                  status=ERROR_CORRUPT_FILE;
28                  print_error(status);
29                  return status;
30          }
31
32          if((crud_file = fopen (argv[CMD_ARG_CRUD_FILE_POSITION], "rt")) == NULL)
33          {
34                  fclose(stock_file);
35                  status=ERROR_CORRUPT_FILE;
36                  print_error(status);
37                  return status;
```

```
38          }
39
40          if((output_file = fopen("stocklist_update.txt", "wt")) == NULL)
41          {
42                  fclose(stock_file);
43                  fclose(crud_file);
44                  status=ERROR_CORRUPT_FILE;
45                  print_error(status);
46                  return status;
47          }
48
49          if((status = compare_records_modif(stock_file,crud_file,output_file)) != OK)
50          {
51                  fclose(stock_file);
52                  fclose(crud_file);
53                  fclose(output_file);
54
55                  print_error(status);
56                  return status;
57          }
58
59          fclose(stock_file);
60          fclose(crud_file);
61
62          if(fclose(output_file) == EOF){
63                  fclose(output_file);
64                  status=ERROR_WRITING_FILE;
65                  print_error(status);
66                  return status;
67          }
68
69          remove(argv[CMD_ARG_STOCK_FILE_POSITION ]);
70
71          rename(output_file, "stocklist_update.txt");
72
73          return OK;
74  }
```

## 4.2. Implementación

### 4.2.1. abm.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include "abm.h"
6   #include "record.h"
7
```

```
8  status_t compare_records_modif(FILE * stock_file,FILE * crud_file, FILE * output_file
       )
9  {
10
11         record_t stock, modif, aux_stock, stock_backup;
12         status_t status;
13         bool_t eof_stock, eof_crud, equal_record;
14
15         if(stock_file == NULL || crud_file == NULL || output_file == NULL)
16                 return ERROR_NULL_POINTER;
17
18         if((status=read_record(stock_file,&stock, &eof_stock))!=OK)
19                 return status;
20
21         if((status=read_record(crud_file,&modif, &eof_crud))!=OK)
22                 return status;
23
24         while(eof_crud==FALSE && eof_stock==FALSE)
25         {
26                         if (stock.id == modif.id)
27                         {
28                                 equal_record=TRUE;
29
30                                 aux_stock = modif;
31                                 stock_backup = modif;
32
33                                 while(eof_crud==FALSE && aux_stock.id ==modif.id)
34                                 {
35                                         aux_stock = modif;
36
37                                         if((status=read_record(crud_file, &modif, &
                                               eof_crud))!=OK)
38                                                 return status;
39                                 }
40
41                                 stock.used_time = (stock.used_time + aux_stock.
                                       used_time + stock_backup.used_time);
42
43                                 if((status=save_record(stock, output_file,
                                       CSV_DELIMITER))!=OK)
44                                         return status;
45
46                                 if((status=read_record(stock_file, &stock, &eof_stock
                                       ))!=OK)
47                                         return status;
48                         }
49
50                         else
51                         {
52                                 if((status=save_record(stock, output_file,
                                       CSV_DELIMITER))!=OK)
```

```
53                                      return status;
54
55                                  if((status=read_record(stock_file, &stock, &eof_stock
                                        ))!=OK)
56                                      return status;
57                          }
58
59          }
60
61          if(equal_record==FALSE)
62                  return ERROR_MISSING_KEY;
63
64
65          while(eof_stock==FALSE)
66          {
67                  if((status=save_record(stock, output_file, CSV_DELIMITER))!=OK)
68                          return status;
69
70                  if((status=read_record(stock_file, &stock, &eof_stock))!=OK)
71                          return status;
72
73          }
74
75          return OK;
76  }
77
78  status_t compare_records_deletions(FILE * stock_file,FILE * crud_file, FILE *
        output_file)
79  {
80          record_t stock, down;
81          status_t status;
82          bool_t eof_stock, eof_crud, equal_record;
83
84          if(stock_file == NULL || crud_file == NULL || output_file == NULL)
85                  return ERROR_NULL_POINTER;
86
87          if((status=read_record(stock_file,&stock, &eof_stock))!=OK)
88                  return status;
89
90          if((status=read_record(crud_file,&down, &eof_crud))!=OK)
91                  return status;
92
93          while(eof_crud==FALSE)
94          {
95                          if(stock.id == down.id)
96                          {
97                                  equal_record=TRUE;
98
99                                  if((status=read_record(crud_file,&down, &eof_crud))!=
                                        OK)
100                                     return status;
```

```
101                         }
102                         else{
103
104                                 save_record(stock, output_file, CSV_DELIMITER);
105
106                                 if((status=read_record(stock_file,&stock, &eof_stock)
                                        )!=OK)
107                                         return status;
108                         }
109
110         }
111
112
113         if(equal_record==FALSE)
114                 return ERROR_MISSING_KEY;
115
116         while(eof_stock==FALSE)
117         {
118                 save_record(stock,output_file, CSV_DELIMITER);
119
120                 if((status=read_record(stock_file,&stock, &eof_stock))!=OK)
121                         return status;
122         }
123
124         return OK;
125 }
126
127 status_t compare_records_uploads(FILE * stock_file,FILE * crud_file, FILE *
        output_file)
128 {
129         record_t stock, update;
130         status_t status;
131         bool_t eof_stock, eof_crud;
132
133         if(stock_file == NULL || crud_file == NULL || output_file == NULL)
134                 return ERROR_NULL_POINTER;
135
136         if((status=read_record(stock_file,&stock, &eof_stock))!=OK)
137                 return status;
138
139         if((status=read_record(crud_file,&update, &eof_crud))!=OK)
140                 return status;
141
142         while(eof_crud==FALSE && eof_stock==FALSE)
143         {
144                         if (stock.id == update.id)
145                                 return ERROR_DUPLICATED_KEY;
146
147                         if (stock.id < update.id){
148                                 if((status=save_record(stock,output_file,
                                        CSV_DELIMITER))!=OK)
```

```
149                                     return status;
150                             if((status=read_record(stock_file,&stock, &eof_stock)
                                    )!=OK)
151                                     return status;
152                     }
153                     else{
154                             if((status=save_record(update,output_file,
                                    CSV_DELIMITER))!=OK)
155                                     return status;
156                             if((status=read_record(crud_file,&update, &eof_crud))
                                    !=OK)
157                                     return status;
158                     }
159         }
160
161         while(eof_stock==FALSE)
162         {
163                 save_record(stock,output_file, CSV_DELIMITER);
164
165                 if((status=read_record(stock_file,&stock, &eof_stock))!=OK)
166                         return status;
167         }
168
169         while(eof_crud==FALSE)
170         {
171                 save_record(update,output_file, CSV_DELIMITER);
172
173                 if((status=read_record(crud_file,&update, &eof_crud))!=OK)
174                         return status;
175         }
176
177         return OK;
178 }
```

#### 4.2.2. abm.h

```
1  #ifndef ABM__H
2  #define ABM__H
3
4  #include "types.h"
5
6  status_t compare_records_modif(FILE * stock_file,FILE * crud_file, FILE * output_file
       );
7
8  status_t compare_records_deletions(FILE * stock_file,FILE * crud_file, FILE *
       output_file);
9
10 status_t compare_records_uploads(FILE * stock_file,FILE * crud_file, FILE *
       output_file);
```

```
11
12  #endif
```

### 4.2.3. record.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include "record.h"
6   #include "utilities.h"
7
8   status_t read_record(FILE * fi, record_t * stocklist, bool_t *eof)
9   {
10          char * file_line;
11          char **string_array;
12          size_t amount_fields;
13
14          if(fi == NULL || stocklist == NULL)
15                  return ERROR_NULL_POINTER;
16
17          if(read_line(fi, &file_line,eof) == OK && (strlen(file_line))!=0)
18          {
19
20                          if(split(file_line, CSV_DELIMITER, &amount_fields, &
                                string_array) != OK)
21                          {
22                                  return ERROR_CREATING_RECORD;
23                          }
24
25                          if(create_record(string_array, stocklist) != OK)
26                          {
27                                  destroy_strings(&string_array, &amount_fields);
28                                  return ERROR_CREATING_RECORD;
29                          }
30
31          }
32          return OK;
33  }
34  status_t create_record(char * string_array[], record_t *stocklist)
35  {
36          char *endp;
37
38          if(string_array == NULL || stocklist == NULL)
39                  return ERROR_NULL_POINTER;
40          (stocklist)->id = strtoul((string_array[ID_POSITION]), &endp, 10);
41
42          if(!*endp)
43          {
```

```
44                  return ERROR_INVALID_ID;
45          }
46
47          if(string_array[RESOURCE_DESCRIPTION_POSITION] == NULL)
48                  strcpy((stocklist)->resources_description , "");
49
50          else
51                  strcpy( (stocklist)->resources_description , string_array[
                        RESOURCE_DESCRIPTION_POSITION]);
52
53          if( string_array[USED_TIME_POSITION] == NULL)
54          {
55                  (stocklist)->used_time = 0.0;
56          }
57          else
58          {
59                  (stocklist)->used_time = strtod((string_array[USED_TIME_POSITION]), &
                        endp);
60
61                       if(*endp)
62                       {
63                               return ERROR_INVALID_USED_TIME;
64                       }
65          }
66          return OK;
67  }
68  status_t save_record(record_t record, FILE * fo, char delimiter)
69  {
70          if(fo == NULL)
71                  return ERROR_NULL_POINTER;
72
73          fprintf(fo,"%lu",record.id);
74
75          fprintf(fo,"%c",delimiter);
76
77          fprintf(fo,"%s",record.resources_description);
78
79          fprintf(fo,"%c",delimiter);
80
81          fprintf(fo,"%f",record.used_time);
82
83          fprintf(fo, "\n");
84          return OK;
85  }
```

#### 4.2.4. record.h

```
1  #ifndef RECORD__H
2  #define RECORD__H
```

```
3
4  #include "types.h"
5
6  #define ID_POSITION 0
7  #define RESOURCE_DESCRIPTION_POSITION 1
8  #define USED_TIME_POSITION 2
9
10 #define MAX_RESOURCES_DESCRIPTION 128
11
12 #define CSV_DELIMITER '|'
13
14 status_t read_record(FILE * fi, record_t * stocklist, bool_t *eof );
15 status_t create_record(char * string_array[], record_t * stocklist );
16 status_t save_record(record_t structure, FILE * fo, char delimiter );
17
18 #endif
```

#### 4.2.5. utilities.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "utilities.h"
6  #include "record.h"
7
8  status_t split (const char * s, char del, size_t * l , char *** string_array)
9  {
10         char *str, *q, *p;
11         char delims[2];
12         size_t i;
13
14         if ( s == NULL || l == NULL || string_array == NULL)
15                 return ERROR_NULL_POINTER;
16
17         delims[0] = del;
18
19         delims[1] = '\0';
20
21         if(strdupl (s,&str) != OK )
22         {
23                 *l = 0;
24                 return ERROR_MEMORY;
25         }
26
27         for(i = 0, *l = 0; str[i]; i++)
28         {
29                 if(str[i] == del)
30                         (*l)++;
```

```
31
32          }
33          (*l)++;
34
35          if((* string_array = (char **)malloc((*l) * sizeof(char *))) == NULL)
36          {
37                  free(str);
38                  *l=0;
39                  return ERROR_MEMORY;
40          }
41
42          for( i=0, q=str; (p = strtok(q, delims))!= NULL; q=NULL, i++)
43          {
44                  if((strdupl(p,&(* string_array)[i])) != OK)
45                  {
46                          free(str);
47                          destroy_strings(string_array, l);
48                          *l=0;
49                          return ERROR_MEMORY;
50                  }
51          }
52          free(str);
53
54          if((*string_array)[USED_TIME_POSITION]==NULL)
55          {
56                  (*string_array)[USED_TIME_POSITION]=(*string_array)[
                        RESOURCE_DESCRIPTION_POSITION];
57
58                  (*string_array)[RESOURCE_DESCRIPTION_POSITION] = "";
59          }
60          return OK;
61
62  }
63
64  status_t strdupl(const char *s, char **t)
65  {
66          size_t i;
67
68          if( s == NULL || t == NULL)
69                  return ERROR_NULL_POINTER;
70
71          if((*t =(char *)malloc((strlen(s)+sizeof(char))*sizeof(char))) == NULL)
72                  return ERROR_MEMORY;
73
74          for( i=0; ((*t)[i] = s[i]) ; i++);
75
76          return OK;
77  }
78
79  status_t read_line(FILE * fi, char ** created_string, bool_t *eof)
80  {
```

```
81          char c;
82          size_t alloc_size, used_size;
83          char * aux;
84
85          if (created_string == NULL || fi == NULL)
86                  return ERROR_NULL_POINTER;
87
88          if((*created_string=(char *)malloc(INIT_SIZE*sizeof(char)))== NULL)
89                  return ERROR_MEMORY;
90
91          alloc_size = INIT_SIZE;
92          used_size = 0;
93
94          while((c = fgetc(fi)) != '\n' && c != EOF)
95          {
96                  if(used_size == alloc_size-1)
97                  {
98                          if((aux=(char*)realloc(*created_string,alloc_size + INIT_SIZE
                                ))) == NULL)
99                          {
100                                 free(*created_string);
101                                 return ERROR_MEMORY;
102                         }
103                         *created_string = aux;
104                         alloc_size += INIT_SIZE;
105                 }
106                 (*created_string)[used_size ++] = c;
107         }
108
109         *eof=(c==EOF)?TRUE:FALSE;
110
111         (*created_string)[used_size]='\0';
112         return OK;
113 }
114
115
116 status_t destroy_strings(char *** string_array, size_t *l)
117 {
118         size_t i;
119
120         if(string_array == NULL)
121                 return ERROR_NULL_POINTER;
122
123         for(i=0; i < *l; i++)
124         {
125                 free(*string_array[i]);
126                 (*string_array)[i]=NULL;
127         }
128
129         free(*string_array);
130
```

```
131         *string_array = NULL;
132
133         *l=0;
134
135         return OK;
136 }
```

#### 4.2.6. utilities.h

```
1  #ifndef UTILITIES__H
2  #define UTILITIES__H
3
4  #include "types.h"
5
6  #define INIT_SIZE 5
7  #define GROWTH_FACTOR 2
8
9
10 status_t split (const char * s, char del, size_t * amount_fields , char ***
       string_array);
11 status_t strdupl(const char *s, char **t);
12 status_t read_line(FILE * fi, char ** created_string, bool_t *eof);
13 status_t destroy_strings(char *** string_array, size_t *l);
14
15
16 #endif
```

### 4.3. Errores

#### 4.3.1. error.c

```
1  #include <stdio.h>
2  #include "error.h"
3
4  status_t print_error(status_t status)
5  {
6           char* errors_dictionary[MAX_ERRORS]=
7                   {
8                           "Null pointer.",
9                           "Insufficient memory.",
10                          "Duplicated ID.",
11                          "Missing ID.",
12                          "Incorrect program invocation.",
13                          "Unable to open file.",
14                          "Incorrect ID.",
15                          "Incorrect USED TIME.",
16                          "Unable to write in file",
17                  };
```

```
18
19          fprintf(stderr, "%s\n", errors_dictionary[status]);
20
21          return OK;
22  }
```

#### 4.3.2. error.h

```
1   #ifndef ERRORS__H
2   #define ERRORS__H
3
4   #include "types.h"
5
6   #define MAX_ERRORS 10
7
8   status_t print_error(status_t status);
9
10  #endif
```

### 4.4. Ejecución

#### 4.4.1. main.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include "main.h"
6   #include "types.h"
7   #include "error.h"
8
9   status_t validate_args(char * argv[], size_t l)
10  {
11          if (argv == NULL)
12                  return ERROR_NULL_POINTER;
13
14          if( l != MAX_CMD_ARGS)
15                  return ERROR_PROG_INVOCATION;
16
17          return OK;
18  }
```

#### 4.4.2. main.h

```
1   #ifndef MAIN_H
2   #define MAIN_H
```

```
3
4  #include "types.h"
5
6  #define MAX_CMD_ARGS 3
7  #define CMD_ARG_STOCK_FILE_POSITION 1
8  #define CMD_ARG_CRUD_FILE_POSITION 2
9
10 status_t validate_args(char * argv[], size_t l);
11
12 #endif
```

#### 4.4.3. types.h

```
1  #ifndef TYPES__H
2  #define TYPES__H
3
4  #define MAX_RESOURCES_DESCRIPTION 128
5
6  typedef enum
7  {
8          OK,
9          ERROR_NULL_POINTER,
10         ERROR_MEMORY,
11         ERROR_DUPLICATED_KEY,
12         ERROR_MISSING_KEY,
13         ERROR_PROG_INVOCATION,
14         ERROR_CORRUPT_FILE,
15         ERROR_INVALID_ID,
16         ERROR_INVALID_USED_TIME,
17         ERROR_CREATING_RECORD,
18         ERROR_WRITING_FILE
19
20 }status_t;
21
22 typedef struct
23 {
24         size_t id;
25         char resources_description[MAX_RESOURCES_DESCRIPTION];
26         float used_time;
27
28 }record_t;
29
30 typedef enum{
31         FALSE,
32         TRUE
33 }bool_t;
34
35 #endif
```