```
/***   main.c  ****/

# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include "errors.h"
# include "types.h"
# include "main.h"
# include "setup.h"
# include "mp3.h"
# include "vector.h"
# include "track.h"

int main (int argc, char * argv [])
{
	status_t st;
	FILE * file_track_list;
	FILE * file_mp3;
	track_list_format_t track_list_format;
	track_sort_type_t track_sort_type;
	destructor_t destructor;
	clone_t clone;
	ADT_Vector_t * ADT_Vector;
	ADT_Track_t ADT_Track;
	size_t mp3_file_index;
	size_t mp3_files_quantity;
	context_t context;
	printer_t printers [NUMBER_OF_PRINTERS_FUNCTIONS] =
	{
		ADT_Track_export_as_csv,
		ADT_Track_export_as_xml,
	};
	comparer_t comparers [NUMBER_OF_COMPARATORS_FUNCTIONS ] =
	{
		ADT_Track_compare_by_name,
		ADT_Track_compare_by_artist,
		ADT_Track_compare_by_genre,
	};


	clone = ADT_Track_clone;
	destructor = ADT_Track_destroy;
	if ((st = validate_arguments (argc, argv, &track_list_format, &track_sort_type, &mp3_files_quantity
)) != OK)
	{
		print_error_msg (st);
		return st;
	}
	if ((st = set_context (&context, mp3_files_quantity)) != OK)
	{
		print_error_msg (st);
		return st;
	}
	if ((file_track_list = fopen (argv [CMD_ARG_POSITION_OUTPUT_FILE], "wt")) == NULL)
	{
		print_error_msg (ERROR_OUTPUT_FILE);
		return ERROR_OUTPUT_FILE;
	}
	if ((st = ADT_Vector_new (&ADT_Vector)) != OK)
	{
		print_error_msg (st);
		fclose (file_track_list);
		return st;
	}
	for (mp3_file_index = 0; mp3_file_index < mp3_files_quantity; mp3_file_index ++)
	{
		if ((file_mp3 = fopen (argv [mp3_file_index + CMD_ARG_POSITION_FIRST_MP3_FILE], "rb"))
== NULL)
		{
```

```
                        ADT_Vector_destroy (&ADT_Vector, destructor);
                        print_error_msg (ERROR_INPUT_MP3_FILE);
                        fclose (file_track_list);
                        return ERROR_INPUT_MP3_FILE;
                }
                if ((st = ADT_Track_new_from_file (&ADT_Track, file_mp3)) != OK)
                {
                        ADT_Vector_destroy (&ADT_Vector, destructor);
                        print_error_msg (st);
                        fclose (file_track_list);
                        fclose (file_mp3);
                        return st;
                }
                if ((st = ADT_Vector_set_element (&ADT_Vector, clone, &ADT_Track, mp3_file_index)) !=
OK)
                {
                        ADT_Vector_destroy (&ADT_Vector, destructor);
                        print_error_msg (st);
                        fclose (file_track_list);
                        fclose (file_mp3);
                        return st;
                }
                fclose (file_mp3);
        }
        if ((st = ADT_Vector_sort (&ADT_Vector, comparers [track_sort_type])) != OK)
        {
                ADT_Vector_destroy (&ADT_Vector, destructor);
                print_error_msg (st);
                fclose (file_track_list);
                return st;
        }

        if ((st = ADT_Vector_export (ADT_Vector, &context, file_track_list, printers [track_list_format])) !=
OK)
        {
                ADT_Vector_destroy (&ADT_Vector, destructor);
                print_error_msg (st);
                fclose (file_track_list);
                return st;
        }
        if ((st = ADT_Vector_destroy (&ADT_Vector, destructor)) != OK)
        {
                print_error_msg (st);
                fclose (file_track_list);
                return st;
        }
        fclose (file_track_list);
        return OK;
}

status_t validate_arguments (int argc, char * argv [], track_list_format_t * track_list_format,
        track_sort_type_t * track_sort_type, size_t * mp3_files_quantity )
{
        status_t st;

        if (argv == NULL || track_list_format == NULL || track_sort_type == NULL || mp3_files_quantity
== NULL)
                return ERROR_NULL_POINTER;
        if ((st = validate_format_argument (argv, track_list_format)) != OK)
                return st;
        if ((st = validate_sort_argument (argv, track_sort_type)) != OK)
                return st;
        if (strcmp (argv [CMD_ARG_POSITION_FLAG_OUTPUT_FILE], CMD_ARG_FLAG_OUTPUT_FILE))
                return ERROR_PROG_INVOCATION;
        if (argc < CMD_ARG_POSITION_FIRST_MP3_FILE)
                return ERROR_PROG_INVOCATION;
        *mp3_files_quantity = argc - CMD_ARG_POSITION_FIRST_MP3_FILE;
        return OK;
}
```

```
status_t validate_format_argument (char * argv [], track_list_format_t * track_list_format)
{
	if (argv == NULL || track_list_format == NULL)
		return ERROR_NULL_POINTER;
	if (strcmp (argv [CMD_ARG_POSITION_FLAG_FORMAT], CMD_ARG_FLAG_FORMAT))
		return ERROR_PROG_INVOCATION;
	if (!strcmp (argv [CMD_ARG_POSITION_FORMAT], CMD_ARG_CSV_FORMAT))
		*track_list_format = TRACK_LIST_FORMAT_CSV;
	if (!strcmp (argv [CMD_ARG_POSITION_FORMAT], CMD_ARG_XML_FORMAT))
		*track_list_format = TRACK_LIST_FORMAT_XML;
	if (*track_list_format != TRACK_LIST_FORMAT_XML && *track_list_format !=
TRACK_LIST_FORMAT_CSV)
		return ERROR_PROG_INVOCATION;
	return OK;
}

status_t validate_sort_argument (char * argv [], track_sort_type_t * track_sort_type)
{
	if (argv == NULL || track_sort_type == NULL)
		return ERROR_NULL_POINTER;
	if (strcmp (argv [CMD_ARG_POSITION_FLAG_SORT], CMD_ARG_FLAG_SORT))
		return ERROR_PROG_INVOCATION;
	if (!strcmp (argv [CMD_ARG_POSITION_SORT], CMD_ARG_NAME_SORT))
		*track_sort_type = TRACK_SORT_BY_NAME;
	if (!strcmp (argv [CMD_ARG_POSITION_SORT], CMD_ARG_ARTIST_SORT))
		*track_sort_type = TRACK_SORT_BY_ARTIST;
	if (!strcmp (argv [CMD_ARG_POSITION_SORT], CMD_ARG_GENRE_SORT))
		*track_sort_type = TRACK_SORT_BY_GENRE;
	if (*track_sort_type != TRACK_SORT_BY_NAME && *track_sort_type != TRACK_SORT_BY_ARTIST
	 && *track_sort_type != TRACK_SORT_BY_GENRE)
		return ERROR_PROG_INVOCATION;
	return OK;
}


status_t set_context (context_t * context, const size_t mp3_files_quantity)
{
	char xml_context_tags [XML_NUMBER_OF_TAG + 1][XML_MAX_TAG_LENGTH + 1 ] =
	{
		XML_PROCESSING_INTRUCTION,
		XML_OPEN_TRACKS_TAG,
		XML_OPEN_TRACK_TAG,
		XML_OPEN_NAME_TAG,
		XML_CLOSE_NAME_TAG,
		XML_OPEN_ARTIST_TAG,
		XML_CLOSE_ARTIST_TAG,
		XML_OPEN_YEAR_TAG,
		XML_CLOSE_YEAR_TAG,
		XML_OPEN_GENRE_TAG,
		XML_CLOSE_GENRE_TAG,
		XML_CLOSE_TRACK_TAG,
		XML_CLOSE_TRACKS_TAG,
	};
	size_t i;

	context -> csv_delimiter = CSV_DELIMITER;
	context -> xml_index = 0;
	context -> xml_close_mark = mp3_files_quantity;
	if (context == NULL)
		return ERROR_NULL_POINTER;
	for (i = 0; i < XML_NUMBER_OF_TAG + 1; ++i)
		strcpy (context -> xml_tags [i], xml_context_tags [i]);
	return OK;
}
```

/*** main.h ***/

```
# ifndef MAIN__H
# define MAIN__H

# include <stdio.h>
# include "errors.h"
# include "setup.h"

# define CMD_ARG_POSITION_FIRST_MP3_FILE 7
# define CMD_ARG_POSITION_FLAG_FORMAT 1
# define CMD_ARG_POSITION_FORMAT 2
# define CMD_ARG_POSITION_FLAG_SORT 3
# define CMD_ARG_POSITION_SORT 4
# define CMD_ARG_POSITION_FLAG_OUTPUT_FILE 5
# define CMD_ARG_POSITION_OUTPUT_FILE 6
# define CMD_ARG_FLAG_FORMAT "-fmt"
# define CMD_ARG_CSV_FORMAT "csv"
# define CMD_ARG_XML_FORMAT "xml"
# define CMD_ARG_FLAG_SORT "-sort"
# define CMD_ARG_NAME_SORT "name"
# define CMD_ARG_ARTIST_SORT "artist"
# define CMD_ARG_GENRE_SORT "genre"
# define CMD_ARG_FLAG_OUTPUT_FILE "-out"

status_t set_context (context_t * context, const size_t mp3_files_quantity);
status_t validate_arguments (int argc, char * argv [], track_list_format_t * track_list_format,
track_sort_type_t * track_sort_type, size_t * mp3_files_quantity );
status_t validate_format_argument (char * argv [], track_list_format_t * track_list_format);
status_t validate_sort_argument (char * argv [], track_sort_type_t * track_sort_type);

# endif
```

/*** types.h ***/

```
# ifndef TYPES__H
# define TYPES__H

# include <stdio.h>
# include "errors.h"

typedef char * string;
typedef unsigned short ushort;

typedef enum
{
        TRUE,
        FALSE
} bool_t;

typedef status_t (* destructor_t) (void *);
typedef status_t (* clone_t ) (const void *, void ** );
typedef status_t (* exporter_t) (const void * pvoid, const void * pcontext, FILE * fo);
typedef status_t (* printer_t) (const void * pvoid, const void * pcontext, FILE * fo);
typedef int (* comparer_t) (const void * pvoid1, const void * pvoid2);

# endif
```

```
/*** setup.h ***/

# ifndef SETUP__H
# define SETUP__H

# include <stdio.h>

# define CSV_DELIMITER '|'
# define XML_PROCESSING_INTRUCTION "<?xml version=\"1.0\" ?>"
# define XML_MAX_TAG_LENGTH 40
# define XML_NUMBER_OF_TAG 12
# define XML_OPEN_TRACKS_TAG "<Tracks>"
# define XML_CLOSE_TRACKS_TAG "</Tracks>"
# define XML_OPEN_TRACK_TAG "<Track>"
# define XML_CLOSE_TRACK_TAG "</Track>"
# define XML_OPEN_NAME_TAG "<Name>"
# define XML_CLOSE_NAME_TAG "</Name>"
# define XML_OPEN_ARTIST_TAG "<Artist>"
# define XML_CLOSE_ARTIST_TAG "</Artist>"
# define XML_OPEN_YEAR_TAG "<Year>"
# define XML_CLOSE_YEAR_TAG "</Year>"
# define XML_OPEN_GENRE_TAG "<Genre>"
# define XML_CLOSE_GENRE_TAG "</Genre>"

typedef struct
{
        char csv_delimiter;
        size_t xml_index;
        size_t xml_close_mark;
        char xml_tags [XML_NUMBER_OF_TAG + 1][XML_MAX_TAG_LENGTH + 1];
} context_t;

typedef enum
{
        TRACK_LIST_FORMAT_CSV,
        TRACK_LIST_FORMAT_XML
} track_list_format_t;

typedef enum
{
        TRACK_SORT_BY_NAME,
        TRACK_SORT_BY_ARTIST,
        TRACK_SORT_BY_GENRE
} track_sort_type_t;

# endif
```

```
/*** vector.c ***/

# include <stdio.h>
# include <stdlib.h>
# include "errors.h"
# include "types.h"
# include "vector.h"

/*Crea un nuevo Vector (tipo de dato abstracto), precarga INIT_CHOP elementos y los incializa a NULL,
indica la cantidad de elementos almacenados con alloc_size (inicialmente cero)
y indica la capacidad de almacenamiento de elementos con size */

status_t ADT_Vector_new ( ADT_Vector_t ** ADT_Vector)
{
        size_t i;

   if (ADT_Vector == NULL)
      return ERROR_NULL_POINTER;
   if (((*ADT_Vector) = (ADT_Vector_t*) malloc (sizeof (ADT_Vector_t))) == NULL)
        return ERROR_NO_MEMORY;
   if (((*ADT_Vector) -> elements = (void **) malloc (INIT_CHOP * sizeof ( void *))) == NULL)
        {
                free (*ADT_Vector);
                *ADT_Vector = NULL;
                return ERROR_NO_MEMORY;
        }
   for (i = 0; i < INIT_CHOP; i++)
        (*ADT_Vector) -> elements [i] = NULL;
   (*ADT_Vector) -> alloc_size = 0;
   (*ADT_Vector) -> size = INIT_CHOP;
   return OK;
}

/*Destruye un nuevo Vector (tipo de dato abstracto), libera la memoria pedida para el Vector y para
los elementos almacenados en él, requiere un puntero a una función destructora de los elementos
correspondientes*/

status_t ADT_Vector_destroy (ADT_Vector_t ** ADT_Vector, status_t (*pf) (void *))
{
        status_t st;
        size_t i;

        if (pf == NULL || ADT_Vector == NULL)
                return ERROR_NULL_POINTER;
        for (i = 0; i < (*ADT_Vector) -> size; ++i)
        {
                if ( (*ADT_Vector) -> elements [i] != NULL)
                {
                        st = (*pf) ((*ADT_Vector) -> elements [i]);
                        if (st != OK)
                                return st;
                        (*ADT_Vector) -> elements [i] = NULL;
                }
        }
        free ( (*ADT_Vector) -> elements);
        (*ADT_Vector) -> elements = NULL;
        free (*ADT_Vector);
        *ADT_Vector = NULL;
        return OK;
}

/*Setea en un Vector (tipo de dato abstracto) en la posición indicada por index,
requiere una función de creación de los elementos correspondientes.
Si existia previamente un elemento en la posición index, en la cual se queria cargar un elementos, se
coincidera un error*/

status_t ADT_Vector_set_element (ADT_Vector_t ** ADT_Vector, status_t (*pf) (const void *, void **), void *
pvoid, size_t index)
{
```

```
	status_t st;
	void ** aux;

	if (pf == NULL || ADT_Vector == NULL)
		return ERROR_NULL_POINTER;
	if ((*ADT_Vector) -> alloc_size == (*ADT_Vector) -> size)
	{
		if ((aux = (void **) realloc ((*ADT_Vector) -> elements , ((*ADT_Vector) -> alloc_size +
CHOP_SIZE) * sizeof ( void *))) == NULL)
			return ERROR_NO_MEMORY;
		(*ADT_Vector) -> elements = aux;
		(*ADT_Vector) -> size += CHOP_SIZE;
	}
	if ((*ADT_Vector) -> elements [index] != NULL)
		return ERROR_OCUPPIED_MEMORY;
	(*ADT_Vector) -> alloc_size ++;
	st = (*pf) (pvoid, (&(*ADT_Vector) -> elements [index]));
	if (st != OK)
		return st;
	return OK;
}

/*Exporta un Vector (tipo de dato abstracto) en el stream fo, requiere un función que imprima elementos
en un formato correspondiente, y un contexto de impresion.*/

status_t ADT_Vector_export (const ADT_Vector_t * ADT_Vector, void * context, FILE * fo, status_t (*pf)
(const void * pvoid, const void * pcontext, FILE * fo))
{
	status_t st;
	size_t i;

	if (pf == NULL || ADT_Vector == NULL || context == NULL)
		return ERROR_NULL_POINTER;
	for (i = 0; i < ADT_Vector -> alloc_size; ++i)
	{
		if ((st = (*pf) (ADT_Vector -> elements [i], context, fo ) != OK))
			return st;
	}
	return OK;
}

/*Ordena un Vector (tipo de dato abstracto) con el metodo SELECTION SORT, requiere un función que
compare (segun un criterio) los elementos del vector.*/

status_t ADT_Vector_sort (ADT_Vector_t ** ADT_Vector, int (* pf_comparer) (const void * pvoid1, const void
* pvoid2))
{
	size_t i, j;
	int min;
	void * clone_element;

	if ( pf_comparer == NULL || ADT_Vector == NULL)
		return ERROR_NULL_POINTER;

	for (i = 0; i < (*ADT_Vector) -> alloc_size - 1; i++)
	{
		min = i;
		for (j = i + 1; j < (*ADT_Vector) -> alloc_size ; j ++)
		{
			if (((* pf_comparer) ( (*ADT_Vector) -> elements [j],  (*ADT_Vector) -> elements
[min]))  < 0)
			{
				min = j;
			}
		}
		clone_element = (*ADT_Vector) -> elements [i];
		(*ADT_Vector) -> elements [i] = (*ADT_Vector) -> elements [min];
		(*ADT_Vector) -> elements [min] = clone_element;
	}
```

```
	return OK;
}

/*** vector.h ***/

# ifndef VECTOR__H
# define VECTOR__H

# include <stdio.h>
# include "errors.h"
# include "types.h"

# define INIT_CHOP 1
# define CHOP_SIZE 2

typedef struct
{
	void ** elements;
	size_t size;
	size_t alloc_size;
} ADT_Vector_t;

status_t ADT_Vector_new (ADT_Vector_t ** ADT_Vector);
status_t ADT_Vector_destroy (ADT_Vector_t ** ADT_Vector, status_t (*pf) (void *));

status_t ADT_Vector_set_element (ADT_Vector_t ** ADT_Vector, status_t (*pf)
 (const void *, void **), void * pvoid, size_t index);

status_t ADT_Vector_export (const ADT_Vector_t * ADT_Vector, void * context, FILE * fo,
	status_t (*pf) (const void * pvoid, const void * pcontext, FILE * fo));

status_t ADT_Vector_sort (ADT_Vector_t ** ADT_Vector, int (* pf_comparer) (const void * pvoid1, const void
* pvoid2));

# endif
```

```
/*** track.c ***/

# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include "errors.h"
# include "mp3.h"
# include "types.h"
# include "track.h"
# include "setup.h"

/*Crea un ADT_Track (tipo de dato abstracto), atraves de un archivo .mp3
 y lo almacena en una variable estatica.*/

status_t ADT_Track_new_from_file (void * pvoid, FILE * file_mp3)
{
   size_t length;
   char * temp;
   char header[MP3_HEADER_SIZE];
   char buf[MP3_HEADER_SIZE];
   ADT_Track_t * ptrack;

   if (file_mp3 == NULL || pvoid == NULL)
     return ERROR_NULL_POINTER;

         ptrack = (ADT_Track_t *) pvoid;
   fseek(file_mp3, 0, SEEK_END);                    /*manda el puntero al final del archivo*/
   length=ftell(file_mp3);                          /*da la distancia al comienzo*/
   fseek(file_mp3,length-MP3_HEADER_SIZE,SEEK_SET);          /*se para en el header MP3*/

   fread(header,sizeof(char), MP3_HEADER_SIZE, file_mp3);

   memcpy(buf,header+LEXEM_START_TITLE,LEXEM_SPAN_TITLE);

   buf[LEXEM_SPAN_TITLE] = '\0';

   strcpy (ptrack -> name, buf);

   memcpy(buf,header+LEXEM_START_ARTIST,LEXEM_SPAN_ARTIST);

   buf[LEXEM_SPAN_ARTIST] = '\0';

   strcpy (ptrack -> artist, buf);


   memcpy(buf,header+LEXEM_START_YEAR,LEXEM_SPAN_YEAR);

   buf[LEXEM_SPAN_YEAR] = '\0';

   ptrack -> year = strtol (buf, &temp, 10);

   if (*temp)
         return ERROR_CORRUPTED_FILE;

   memcpy(buf,header+LEXEM_START_GENRE,LEXEM_SPAN_GENRE);
   ptrack -> genre = buf [0];

   return OK;
}

/*Destruye un ADT_Track (tipo de dato abstracto), libera la memoria pedida y pone
sus campos a un valor seguro.*/

status_t ADT_Track_destroy (void * pvoid)
{
         ADT_Track_t * ptrack;

         if (pvoid == NULL)
                  return ERROR_NULL_POINTER;
```

```
	ptrack = (ADT_Track_t *) pvoid;

	strcpy (ptrack -> name, DEFAULT_TRACK_NAME);
	strcpy (ptrack -> artist, DEFAULT_TRACK_ARTIST);

	ptrack -> year = DEFAULT_TRACK_YEAR;
	ptrack -> genre = Other;

	free (ptrack);
	ptrack = NULL;

	return OK;
}

/*Clona un ADT_Track (tipo de dato abstracto), pide memoria y copia los campos de otro
ADT_Track (este puede ser estatico o no).*/

status_t ADT_Track_clone (const void * pvoid1, void ** pvoid2)
{
	ADT_Track_t * ptrack1;
	ADT_Track_t ** ptrack2;

	if (pvoid1 == NULL)		/*pvoid2 puede ser NULL.*/
		return ERROR_NULL_POINTER;

	ptrack1 = (ADT_Track_t *) pvoid1;
	ptrack2 = (ADT_Track_t **) pvoid2;

	if ((*ptrack2 = (ADT_Track_t *) malloc (sizeof (ADT_Track_t))) == NULL)
	return ERROR_NO_MEMORY;

	strcpy ( (*ptrack2) -> name, ptrack1 -> name);
	strcpy ( (*ptrack2) -> artist, ptrack1 -> artist);

	(*ptrack2) -> year = ptrack1 -> year;
	(*ptrack2) -> genre = ptrack1 -> genre;

	return OK;
}


/*Compara 2 ADT_Track (tipo de dato abstracto) por campo nombre*/

int ADT_Track_compare_by_name (const void * pvoid1, const void * pvoid2)
{
	ADT_Track_t * ptrack1;
	ADT_Track_t * ptrack2;

	if (pvoid1 == NULL || pvoid2 == NULL)
		return ERROR_NULL_POINTER;

	ptrack1 = (ADT_Track_t *) pvoid1;
	ptrack2 = (ADT_Track_t *) pvoid2;

	return strcmp (ptrack1 -> name, ptrack2 -> name);
}

/*Compara 2 ADT_Track (tipo de dato abstracto) por campo Artista*/

int ADT_Track_compare_by_artist (const void * pvoid1, const void * pvoid2)
{
	ADT_Track_t * ptrack1;
	ADT_Track_t * ptrack2;

	if (pvoid1 == NULL || pvoid2 == NULL)
		return ERROR_NULL_POINTER;

	ptrack1 = (ADT_Track_t *) pvoid1;
```

```
	ptrack2 = (ADT_Track_t *) pvoid2;

	return strcmp (ptrack1 -> artist, ptrack2 -> artist);
}

/*Compara 2 ADT_Track (tipo de dato abstracto) por campo genero*/

int ADT_Track_compare_by_genre (const void * pvoid1, const void * pvoid2)
{
	ADT_Track_t * ptrack1;
	ADT_Track_t * ptrack2;

	if (pvoid1 == NULL || pvoid2 == NULL)
		return ERROR_NULL_POINTER;

	ptrack1 = (ADT_Track_t *) pvoid1;
	ptrack2 = (ADT_Track_t *) pvoid2;

	return ptrack1 -> genre - ptrack2 -> genre;
}


/*Exporta un ADT_Track (tipo de dato abstracto) con formato csv en un flujo fo, requiere un
contexto de impresion.*/

status_t ADT_Track_export_as_csv (const void * pvoid, const void * pcontext, FILE * fo)
{
	ADT_Track_t * ptrack;
	context_t * context;

	static string genres [NUMBER_OF_GENRES] =
	{
		GENRE_BLUES,
		GENRE_CLASSIC_ROCK,
		GENRE_COUNTRY,
		GENRE_DANCE,
		GENRE_DISCO,
		GENRE_FUNK,
		GENRE_GRUNGE,
		GENRE_HIP_HOP,
		GENRE_JAZZ,
		GENRE_METAL,
		GENRE_NEW_AGE,
		GENRE_OLDIES,
		GENRE_OTHER,
		GENRE_POP,
		GENRE_R_AND_B,
		GENRE_RAP,
		GENRE_REGGAE,
		GENRE_ROCK,
		GENRE_TECHNO,
		GENRE_INDUSTRIAL,
		GENRE_ALTERNATIVE,
		GENRE_SKA,
		GENRE_DEATH_METAL,
		GENRE_PRANKS,
		GENRE_SOUNDTRACK,
		GENRE_EURO_TECHNO,
		GENRE_AMBIENT,
		GENRE_TRIP_HOP,
		GENRE_VOCAL,
		GENRE_JAZZ_PLUS_FUNK,
		GENRE_FUSION,
		GENRE_TRANCE,
		GENRE_CLASSICAL,
		GENRE_INSTRUMENTAL,
		GENRE_ACID,
		GENRE_HOUSE,
		GENRE_GAME,
```

GENRE_SOUND_CLIP,
GENRE_GOSPEL,
GENRE_NOISE,
GENRE_ALTERNROCK,
GENRE_BASS,
GENRE_SOUL,
GENRE_PUNK,
GENRE_SPACE,
GENRE_MEDITATIVE,
GENRE_INSTRUMENTAL_POP,
GENRE_INSTRUMENTAL_ROCK,
GENRE_ETHNIC,
GENRE_GOTHIC,
GENRE_DARKWAVE,
GENRE_TECHNO_INDUSTRIAL,
GENRE_ELECTRONIC,
GENRE_POP_FOLK,
GENRE_EURODANCE,
GENRE_DREAM,
GENRE_SOUTHERN_ROCK,
GENRE_COMEDY,
GENRE_CULT,
GENRE_GANGSTA,
GENRE_TOP_40,
GENRE_CHRISTIAN_RAP,
GENRE_POP_FUNK,
GENRE_JUNGLE,
GENRE_NATIVE_AMERICAN,
GENRE_CABARET,
GENRE_NEW_WAVE,
GENRE_PSYCHADELIC,
GENRE_RAVE,
GENRE_SHOWTUNES,
GENRE_TRAILER,
GENRE_Lo_Fi,
GENRE_TRIBAL,
GENRE_ACID_PUNK,
GENRE_ACID_JAZZ,
GENRE_POLKA,
GENRE_RETRO,
GENRE_MUSICAL,
GENRE_ROCK_AND_ROLL,
GENRE_HARD_ROCK,
GENRE_FOLK,
GENRE_FOLK_ROCK,
GENRE_NATIONAL_FOLK,
GENRE_SWING,
GENRE_FAST_FUSION,
GENRE_BEBOB,
GENRE_LATIN,
GENRE_REVIVAL,
GENRE_CELTIC,
GENRE_BLUEGRASS,
GENRE_AVANTGARDE,
GENRE_GOTHIC_ROCK,
GENRE_PROGRESSIVE_ROCK,
GENRE_PSYCHEDELIC_ROCK,
GENRE_SYMPHONIC_ROCK,
GENRE_SLOW_ROCK,
GENRE_BIG_BAND,
GENRE_CHORUS,
GENRE_EASY_LISTENING,
GENRE_ACOUSTIC,
GENRE_HUMOUR,
GENRE_SPEECH,
GENRE_CHANSON,
GENRE_OPERA,
GENRE_CHAMBER_MUSIC,
GENRE_SONATA,

```
                GENRE_SYMPHONY,
                GENRE_BOOTY_BRASS,
                GENRE_PRIMUS,
                GENRE_PORN_GROOVE,
                GENRE_SATIRE,
                GENRE_SLOW_JAM,
                GENRE_CLUB,
                GENRE_TANGO,
                GENRE_SAMBA,
                GENRE_FOLKLORE,
                GENRE_BALLAD,
                GENRE_POWEER_BALLLAD,
                GENRE_RHYTMIC_SOUL,
                GENRE_FREESTYLE,
                GENRE_DUET,
                GENRE_PUNK_ROCK,
                GENRE_DRUM_SOLO,
                GENRE_A_CAPELA,
                GENRE_EURO_HOUSE,
                GENRE_DANCE_HALL,
        };

        if (pvoid == NULL || pcontext == NULL || fo == NULL)
                return ERROR_NULL_POINTER;

        ptrack = (ADT_Track_t *) pvoid;
        context = (context_t *) pcontext;

        fprintf (fo,"%s%c%s%c%hu%c%s\n", ptrack -> name, context -> csv_delimiter, ptrack -> artist,
                context -> csv_delimiter, ptrack -> year, context -> csv_delimiter , genres [ptrack->
genre]);

        return OK;
}

/*Exporta un ADT_Track (tipo de dato abstracto) con formato xml en un flujo fo, requiere un
contexto de impresion.*/

status_t ADT_Track_export_as_xml (const void * pvoid, const void * pcontext, FILE * fo)
{
        size_t i;
        ADT_Track_t * ptrack;
        context_t * context;

        static string genres [NUMBER_OF_GENRES] =
        {
                GENRE_BLUES,
                GENRE_CLASSIC_ROCK,
                GENRE_COUNTRY,
                GENRE_DANCE,
                GENRE_DISCO,
                GENRE_FUNK,
                GENRE_GRUNGE,
                GENRE_HIP_HOP,
                GENRE_JAZZ,
                GENRE_METAL,
                GENRE_NEW_AGE,
                GENRE_OLDIES,
                GENRE_OTHER,
                GENRE_POP,
                GENRE_R_AND_B,
                GENRE_RAP,
                GENRE_REGGAE,
                GENRE_ROCK,
                GENRE_TECHNO,
                GENRE_INDUSTRIAL,
                GENRE_ALTERNATIVE,
                GENRE_SKA,
                GENRE_DEATH_METAL,
```

GENRE_PRANKS,
GENRE_SOUNDTRACK,
GENRE_EURO_TECHNO,
GENRE_AMBIENT,
GENRE_TRIP_HOP,
GENRE_VOCAL,
GENRE_JAZZ_PLUS_FUNK,
GENRE_FUSION,
GENRE_TRANCE,
GENRE_CLASSICAL,
GENRE_INSTRUMENTAL,
GENRE_ACID,
GENRE_HOUSE,
GENRE_GAME,
GENRE_SOUND_CLIP,
GENRE_GOSPEL,
GENRE_NOISE,
GENRE_ALTERNROCK,
GENRE_BASS,
GENRE_SOUL,
GENRE_PUNK,
GENRE_SPACE,
GENRE_MEDITATIVE,
GENRE_INSTRUMENTAL_POP,
GENRE_INSTRUMENTAL_ROCK,
GENRE_ETHNIC,
GENRE_GOTHIC,
GENRE_DARKWAVE,
GENRE_TECHNO_INDUSTRIAL,
GENRE_ELECTRONIC,
GENRE_POP_FOLK,
GENRE_EURODANCE,
GENRE_DREAM,
GENRE_SOUTHERN_ROCK,
GENRE_COMEDY,
GENRE_CULT,
GENRE_GANGSTA,
GENRE_TOP_40,
GENRE_CHRISTIAN_RAP,
GENRE_POP_FUNK,
GENRE_JUNGLE,
GENRE_NATIVE_AMERICAN,
GENRE_CABARET,
GENRE_NEW_WAVE,
GENRE_PSYCHADELIC,
GENRE_RAVE,
GENRE_SHOWTUNES,
GENRE_TRAILER,
GENRE_Lo_Fi,
GENRE_TRIBAL,
GENRE_ACID_PUNK,
GENRE_ACID_JAZZ,
GENRE_POLKA,
GENRE_RETRO,
GENRE_MUSICAL,
GENRE_ROCK_AND_ROLL,
GENRE_HARD_ROCK,
GENRE_FOLK,
GENRE_FOLK_ROCK,
GENRE_NATIONAL_FOLK,
GENRE_SWING,
GENRE_FAST_FUSION,
GENRE_BEBOB,
GENRE_LATIN,
GENRE_REVIVAL,
GENRE_CELTIC,
GENRE_BLUEGRASS,
GENRE_AVANTGARDE,
GENRE_GOTHIC_ROCK,

```
                GENRE_PROGRESSIVE_ROCK,
                GENRE_PSYCHEDELIC_ROCK,
                GENRE_SYMPHONIC_ROCK,
                GENRE_SLOW_ROCK,
                GENRE_BIG_BAND,
                GENRE_CHORUS,
                GENRE_EASY_LISTENING,
                GENRE_ACOUSTIC,
                GENRE_HUMOUR,
                GENRE_SPEECH,
                GENRE_CHANSON,
                GENRE_OPERA,
                GENRE_CHAMBER_MUSIC,
                GENRE_SONATA,
                GENRE_SYMPHONY,
                GENRE_BOOTY_BRASS,
                GENRE_PRIMUS,
                GENRE_PORN_GROOVE,
                GENRE_SATIRE,
                GENRE_SLOW_JAM,
                GENRE_CLUB,
                GENRE_TANGO,
                GENRE_SAMBA,
                GENRE_FOLKLORE,
                GENRE_BALLAD,
                GENRE_POWEER_BALLLAD,
                GENRE_RHYTMIC_SOUL,
                GENRE_FREESTYLE,
                GENRE_DUET,
                GENRE_PUNK_ROCK,
                GENRE_DRUM_SOLO,
                GENRE_A_CAPELA,
                GENRE_EURO_HOUSE,
                GENRE_DANCE_HALL,
        };

        if (pvoid == NULL || pcontext == NULL || fo == NULL)
                return ERROR_NULL_POINTER;

        ptrack = (ADT_Track_t *) pvoid;
        context = (context_t *) pcontext;

        if (context -> xml_index == 0)
        {
                for (i = 0; i < 2; ++i)
                        fprintf (fo, "%s\n" ,context -> xml_tags [i]);
        }
        context -> xml_index ++;
        i = 2;
        fprintf (fo, "\t%s\n" ,context -> xml_tags [i]);
        i ++ ;
        fprintf (fo, "\t\t%s %s %s\n" ,context -> xml_tags [i], ptrack -> name,  context -> xml_tags [i + 1]);
        i += 2;
        fprintf (fo, "\t\t%s %s %s\n" ,context -> xml_tags [i], ptrack -> artist,  context -> xml_tags [i + 1]);
        i += 2;
        fprintf (fo, "\t\t%s %hu %s\n" ,context -> xml_tags [i], ptrack -> year,  context -> xml_tags [i +
1]);
        i += 2;
        fprintf (fo, "\t\t%s %s %s\n" ,context -> xml_tags [i], genres [ptrack -> genre] ,  context ->
xml_tags [i + 1]);
        i += 2;
        fprintf (fo, "%s\n" , context -> xml_tags [i]);
        i ++;

        if (context -> xml_index == context -> xml_close_mark)
                fprintf (fo, "%s\n" , context -> xml_tags [i]);

        return OK;
}
```

```
/*** track.h ***/

# ifndef TRACK__H
# define TRACK__H

# include <stdio.h>
# include "errors.h"
# include "mp3.h"
# include "types.h"

# define MAX_TRACK_NAME_LENGTH 31
# define MAX_TRACK_ARTIST_LENGTH 31
# define DEFAULT_TRACK_NAME "Unknown"
# define DEFAULT_TRACK_ARTIST "Unknown"
# define DEFAULT_TRACK_YEAR 1900
# define NUMBER_OF_PRINTERS_FUNCTIONS 2
# define NUMBER_OF_COMPARATORS_FUNCTIONS 3

typedef struct
{
        char name [MAX_TRACK_NAME_LENGTH];
        char artist [MAX_TRACK_ARTIST_LENGTH];
        ushort year;
        track_genre_t genre;
} ADT_Track_t;

status_t ADT_Track_new_from_file (void * pvoid, FILE * file_mp3);
status_t ADT_Track_destroy (void * pvoid);
status_t ADT_Track_clone (const void * pvoid1, void ** pvoid2);
int ADT_Track_compare_by_name (const void * pvoid1, const void * pvoid2);
int ADT_Track_compare_by_artist (const void * pvoid1, const void * pvoid2);
int ADT_Track_compare_by_genre (const void * pvoid1, const void * pvoid2);
status_t ADT_Track_export_as_csv (const void * pvoid, const void * pcontext, FILE * fo);
status_t ADT_Track_export_as_xml (const void * pvoid, const void * pcontext, FILE * fo);

# endif


/*** mp3.c ***/


# include <stdio.h>
# include <string.h>
# include "errors.h"
# include "main.h"
# include "mp3.h"
# include "track.h"
# include "setup.h"

/* AGREGAR TDA VECTOR COMO ARGUMENTO*/

status_t get_mp3_header (FILE * file_mp3)
{
   status_t st;
   ADT_Track_t * ADT_Track;

    /*PRUEBA*/
   context_t context;
   context . csv_delimiter = CSV_DELIMITER;
   /*FIN DE PRUEBA - BORRAR LUEGO*/

   if (file_mp3 == NULL)
      return ERROR_NULL_POINTER;
   if ((st = TDA_Track_new_from_file (&ADT_Track, file_mp3)) != OK)
      return st;

   /*PRUEBA*/
   ADT_Track_export_as_csv (ADT_Track, &context, stdout);
   /*FIN DE PRUEBA - BORRAR LUEGO*/
```

```
    if ((st = TDA_Track_destroy (&ADT_Track)) != OK)
        return st;
    return OK;
}


/*** mp3.h ***/

# ifndef MP3__H
# define MP3__H

# include <stdio.h>
# include "errors.h"
# include "vector.h"

# define MP3_HEADER_SIZE              128
# define LEXEM_START_TAG     0
# define LEXEM_SPAN_TAG      3
# define LEXEM_START_TITLE   3
# define LEXEM_SPAN_TITLE   30
# define LEXEM_START_ARTIST 33
# define LEXEM_SPAN_ARTIST  30
# define LEXEM_START_ALBUM  63
# define LEXEM_SPAN_ALBUM   30
# define LEXEM_START_YEAR   93
# define LEXEM_SPAN_YEAR     4
# define LEXEM_START_COMMENT 97
# define LEXEM_SPAN_COMMENT  30
# define LEXEM_START_GENRE  127
# define LEXEM_SPAN_GENRE     1
# define NUMBER_OF_GENRES 126
# define GENRE_BLUES "Blues"
# define GENRE_CLASSIC_ROCK "Classic Rock"
# define GENRE_COUNTRY "Country"
# define GENRE_DANCE "Dance"
# define GENRE_DISCO "Disco"
# define GENRE_FUNK "Funk"
# define GENRE_GRUNGE "Grunge"
# define GENRE_HIP_HOP "Hip Hop"
# define GENRE_JAZZ "Jazz"
# define GENRE_METAL "Metal"
# define GENRE_NEW_AGE "New Age"
# define GENRE_OLDIES "oldies"
# define GENRE_OTHER "OTHER"
# define GENRE_POP "Pop"
# define GENRE_R_AND_B "R&B"
# define GENRE_RAP "Rap"
# define GENRE_REGGAE "Reggae"
# define GENRE_ROCK "Rock"
# define GENRE_TECHNO "Techno"
# define GENRE_INDUSTRIAL "Industrial"
# define GENRE_ALTERNATIVE "Alternative"
# define GENRE_SKA "Ska"
# define GENRE_DEATH_METAL "Death Metal"
# define GENRE_PRANKS "Pranks"
# define GENRE_SOUNDTRACK "Soundtrack"
# define GENRE_EURO_TECHNO "Euro Techno"
# define GENRE_AMBIENT "Ambient"
# define GENRE_TRIP_HOP "Trip Hop"
# define GENRE_VOCAL "Vocal"
# define GENRE_JAZZ_PLUS_FUNK "Jazz + Funk"
# define GENRE_FUSION "Fusion"
# define GENRE_TRANCE "Trance"
# define GENRE_CLASSICAL "Classical"
# define GENRE_INSTRUMENTAL "Instrumental"
# define GENRE_ACID "Acid"
# define GENRE_HOUSE "House"
# define GENRE_GAME "Game"
```

```
# define GENRE_SOUND_CLIP "Sound Clip"
# define GENRE_GOSPEL "Gospel"
# define GENRE_NOISE "Noise"
# define GENRE_ALTERNROCK "AlternRock"
# define GENRE_BASS "Bass"
# define GENRE_SOUL "Soul"
# define GENRE_PUNK "Punk"
# define GENRE_SPACE "Space"
# define GENRE_MEDITATIVE "Meditative"
# define GENRE_INSTRUMENTAL_POP "Instrumental Pop"
# define GENRE_INSTRUMENTAL_ROCK "Instrumental Rock"
# define GENRE_ETHNIC "Ethnic"
# define GENRE_GOTHIC "Gothic"
# define GENRE_DARKWAVE "Darkwave"
# define GENRE_TECHNO_INDUSTRIAL "Techno Industrial"
# define GENRE_ELECTRONIC "Electronic"
# define GENRE_POP_FOLK "Pop Folk"
# define GENRE_EURODANCE "Eurodance"
# define GENRE_DREAM "Dream"
# define GENRE_SOUTHERN_ROCK "Southern Rock"
# define GENRE_COMEDY "Comedy"
# define GENRE_CULT "Cult"
# define GENRE_GANGSTA "Gangsta"
# define GENRE_TOP_40 "Top 40"
# define GENRE_CHRISTIAN_RAP "Christian Rap"
# define GENRE_POP_FUNK "Pop Funk"
# define GENRE_JUNGLE "Jungle"
# define GENRE_NATIVE_AMERICAN "Native American"
# define GENRE_CABARET "Cabaret"
# define GENRE_NEW_WAVE "New Wake"
# define GENRE_PSYCHADELIC "Psychadelic"
# define GENRE_RAVE "Rave"
# define GENRE_SHOWTUNES "Showtunes"
# define GENRE_TRAILER "Trailer"
# define GENRE_Lo_Fi "Lo-Fi"
# define GENRE_TRIBAL "Tribal"
# define GENRE_ACID_PUNK "Acid Punk"
# define GENRE_ACID_JAZZ "Acid Jazz"
# define GENRE_POLKA "Polka"
# define GENRE_RETRO "Retro"
# define GENRE_MUSICAL "Musical"
# define GENRE_ROCK_AND_ROLL "Rock&Roll"
# define GENRE_HARD_ROCK "Hard Rock"
# define GENRE_FOLK "Folk"
# define GENRE_FOLK_ROCK "Folk Rock"
# define GENRE_NATIONAL_FOLK "National Folk"
# define GENRE_SWING "Swing"
# define GENRE_FAST_FUSION "Fast Fusion"
# define GENRE_BEBOB "Bebob"
# define GENRE_LATIN "Latin"
# define GENRE_REVIVAL "Revival"
# define GENRE_CELTIC "Celtic"
# define GENRE_BLUEGRASS "Bluegrass"
# define GENRE_AVANTGARDE "Avantgarde"
# define GENRE_GOTHIC_ROCK "Gothic Rock"
# define GENRE_PROGRESSIVE_ROCK "Progressive Rock"
# define GENRE_PSYCHEDELIC_ROCK "Psychedelic Rock"
# define GENRE_SYMPHONIC_ROCK "Symphonic Rock"
# define GENRE_SLOW_ROCK "Slow Rock"
# define GENRE_BIG_BAND "Big Band"
# define GENRE_CHORUS "Chorus"
# define GENRE_EASY_LISTENING "Easy Listening"
# define GENRE_ACOUSTIC "Acoustic"
# define GENRE_HUMOUR "Humour"
# define GENRE_SPEECH "Speech"
# define GENRE_CHANSON "Chanson"
# define GENRE_OPERA "Opera"
# define GENRE_CHAMBER_MUSIC "Chamber Music"
# define GENRE_SONATA "Sonata"
```

```
# define GENRE_SYMPHONY "Symphony"
# define GENRE_BOOTY_BRASS "Booty Brass"
# define GENRE_PRIMUS "Primus"
# define GENRE_PORN_GROOVE "Porn Groove"
# define GENRE_SATIRE "Satire"
# define GENRE_SLOW_JAM "Slow Jam"
# define GENRE_CLUB "Club"
# define GENRE_TANGO "Tango"
# define GENRE_SAMBA "Samba"
# define GENRE_FOLKLORE "Folklore"
# define GENRE_BALLAD "Ballad"
# define GENRE_POWEER_BALLLAD "Poweer Ballad"
# define GENRE_RHYTMIC_SOUL "Rhytmic Soul"
# define GENRE_FREESTYLE "Freestyle"
# define GENRE_DUET "Duet"
# define GENRE_PUNK_ROCK "Punk Rock"
# define GENRE_DRUM_SOLO "Drum Solo"
# define GENRE_A_CAPELA "A Capela"
# define GENRE_EURO_HOUSE "Euro_House"
# define GENRE_DANCE_HALL "Dance_Hall"

typedef enum
{
        Blues = 0,
        Classic_Rock = 1,
        Country = 2,
        Dance = 3,
        Disco = 4,
        Funk = 5,
        Grunge = 6,
        Hip_Hop = 7,
        Jazz = 8,
        Metal = 9,
        New_Age = 10,
        Oldies = 11,
        Other = 12,
        Pop = 13,
        R_and_B = 14,
        Rap = 15,
        Reggae = 16,
        Rock = 17,
        Techno = 18,
        Industrial = 19,
        Alternative = 20,
        Ska = 21,
        Death_Metal = 22,
        Pranks = 23,
        Soundtrack = 24,
        Euro_Techno = 25,
        Ambient = 26,
        Trip_Hop = 27,
        Vocal = 28,
        Jazz_plus_Funk = 29,
        Fusion = 30,
        Trance = 31,
        Classical = 32,
        Instrumental = 33,
        Acid = 34,
        House = 35,
        Game = 36,
        Sound_Clip = 37,
        Gospel = 38,
        Noise = 39,
        AlternRock = 40,
        Bass = 41,
        Soul = 42,
        Punk = 43,
        Space = 44,
        Meditative = 45,
```

Instrumental_Pop = 46,
Instrumental_Rock = 47,
Ethnic = 48,
Gothic = 49,
Darkwave = 50,
Techno_Industrial = 51,
Electronic = 52,
Pop_Folk = 53,
Eurodance = 54,
Dream = 55,
Southern_Rock = 56,
Comedy = 57,
Cult = 58,
Gangsta = 59,
Top_40 = 60,
Christian_Rap = 61,
Pop_Funk = 62,
Jungle = 63,
Native_American = 64,
Cabaret = 65,
New_Wave = 66,
Psychadelic = 67,
Rave = 68,
Showtunes = 69,
Trailer = 70,
Lo_Fi = 71,
Tribal = 72,
Acid_Punk = 73,
Acid_Jazz = 74,
Polka = 75,
Retro = 76,
Musical = 77,
Rock_and_Roll = 78,
Hard_Rock = 79,
Folk = 80,
Folk_Rock = 81,
National_Folk = 82,
Swing = 83,
Fast_Fusion = 84,
Bebob = 85,
Latin = 86,
Revival = 87,
Celtic = 88,
Bluegrass = 89,
Avantgarde = 90,
Gothic_Rock = 91,
Progressive_Rock = 92,
Psychedelic_Rock = 93,
Symphonic_Rock = 94,
Slow_Rock = 95,
Big_Band = 96,
Chorus = 97,
Easy_Listening = 98,
Acoustic = 99,
Humour = 100,
Speech = 101,
Chanson = 102,
Opera = 103,
Chamber_Music = 104,
Sonata = 105,
Symphony = 106,
Booty_Brass = 107,
Primus = 108,
Porn_Groove = 109,
Satire = 110,
Slow_Jam = 111,
Club = 112,
Tango = 113,
Samba = 114,

```
        Folklore = 115,
        Ballad = 116,
        Poweer_Ballad = 117,
        Rhytmic_Soul = 118,
        Freestyle = 119,
        Duet = 120,
        Punk_Rock = 121,
        Drum_Solo = 122,
        A_Capela = 123,
        Euro_House = 124,
        Dance_Hall = 125
} track_genre_t;

status_t get_mp3_header (FILE * file_mp3, ADT_Vector_t ** ADT_Vector);

# endif


/*** errors.c ***/

# include <stdio.h>
# include <string.h>
# include "errors.h"

status_t print_error_msg (status_t st)
{
        static char * errors [MAX_ERRORS] =
        {
                MSG_OK,
                MSG_ERROR_NULL_POINTER,
                MSG_ERROR_NO_MEMORY,
                MSG_ERROR_INPUT_MP3_FILE,
                MSG_ERROR_OUTPUT_FILE,
                MSG_ERROR_DISK_SPACE,
                MSG_ERROR_CORRUPTED_FILE,
                MSG_ERROR_PROG_INVOCATION,
                MSG_ERROR_OCUPPIED_MEMORY
        };
        fprintf (stderr, "%s\n", errors [st]);
        return OK;
}


/*** errors.h ***/

# ifndef ERRORS__H
# define ERRORS__H

# include <stdio.h>

typedef enum
{
        OK,
        ERROR_NULL_POINTER,
        ERROR_NO_MEMORY,
        ERROR_INPUT_MP3_FILE,
        ERROR_OUTPUT_FILE,
        ERROR_DISK_SPACE,
        ERROR_CORRUPTED_FILE,
        ERROR_PROG_INVOCATION,
        ERROR_OCUPPIED_MEMORY
} status_t;

# define MAX_ERRORS 9
# define MSG_OK "OK"
# define MSG_ERROR_NULL_POINTER "Puntero nulo"
# define MSG_ERROR_NO_MEMORY "Memoria insuficiente en el sistema"
# define MSG_ERROR_INPUT_MP3_FILE "Archivo .mp3 inválido"
# define MSG_ERROR_OUTPUT_FILE "Archivo de salida inválido"
```

```
# define MSG_ERROR_DISK_SPACE "Falta de espacio en disco"
# define MSG_ERROR_CORRUPTED_FILE "Archivo corrupto"
# define MSG_ERROR_PROG_INVOCATION "Los argumentos en linea de orden son inválidos"
# define MSG_ERROR_OCUPPIED_MEMORY "Se intento sobreescribir una posición de memoria ya
ocupada"

status_t print_error_msg (status_t st);

# endif
```

```
/*** Makefile ***/



CC=gcc
CFLAGS=-Wall -ansi -pedantic -c
LFLAGS=-Wall -ansi -pedantic

all: mp3explorer clean

mp3explorer: main.o vector.o track.o errors.o
	$(CC) $(LFLAGS) -o mp3explorer  main.o vector.o track.o errors.o

main.o: main.c main.h setup.h mp3.h errors.h types.h vector.h track.h
	$(CC) $(CFLAGS) -o main.o main.c

vector.o: vector.c errors.h vector.h types.h
	$(CC) $(CFLAGS) -o vector.o vector.c

track.o: track.c errors.h track.h mp3.h types.h setup.h
	$(CC) $(CFLAGS) -o track.o track.c

errors.o: errors.c errors.h
	$(CC) $(CFLAGS) -o errors.o errors.c

clean:
	rm *.o
```