FACULTAD DE INFORMÁTICA
UNIVERSIDAD POLITÉCNICA DE MADRID

Diseño e Implementación de un TAD para Tablas de Símbolos: TS2006

Manual de Usuario
(común para las versiones C y C++)

Realizado por:
Daniel Peña Vázquez

Tutor:
José Luis Fuertes Castro

Editado el 29 de Marzo de 2011

ÍNDICE (1/2)

	Pág.
1. ¿Qué es una tabla de símbolos?	4
2. Ventajas de esta tabla de símbolos TS2006	5
3. Requisitos previos a la utilización de TS2006	6
4. Cómo usar la versión C de TS2006	7
5. Cómo usar la versión C++ de TS2006	9
6. Funciones C y ejemplos de uso	11
6.1 Crear una tabla de símbolos	11
6.2 Destruir una tabla de símbolos	11
6.3 Crear una nueva entrada	11
6.4 Asignar el tipo de entrada a una entrada	12
6.5 Consultar el tipo de entrada de una entrada	12
6.6 Crear un atributo entero en una entrada	12
6.7 Crear un atributo cadena en una entrada	13
6.8 Asignar el valor de un atributo entero	13
6.9 Asignar el valor de un atributo cadena	14
6.10 Consultar el valor de un atributo entero	14
6.11 Consultar el valor de un atributo cadena	15
6.12 Determinar la existencia de una tabla de símbolos	15
6.13 Determinar la existencia de una entrada	16
6.14 Determinar la existencia de un atributo	16
6.15 Escribir el contenido actual de una tabla en un fichero	17
6.16 Consultar el ID del último error que se ha producido	17
6.17 Consultar la descripción del último error que se ha producido	17
6.18 Buscar posición de una entrada	18
6.19 Asignar el tipo de entrada a una entrada (por posición)	18
6.20 Consultar el tipo de entrada de una entrada (por posición)	18
6.21 Crear un atributo entero en una entrada (por posición)	19
6.22 Crear un atributo cadena en una entrada (por posición)	19
6.23 Asignar el valor de un atributo entero (por posición)	20
6.24 Asignar el valor de un atributo cadena (por posición)	20
6.25 Consultar el valor de un atributo entero (por posición)	21
6.26 Consultar el valor de un atributo cadena (por posición)	21
6.27 Determinar la existencia de un atributo (por posición)	22
6.28 Resetear la variable que almacena el último error producido.	22

ÍNDICE (2/2)

	Pág.
7. Funciones C++ y ejemplos de uso	23
7.1 Crear una tabla de símbolos	23
7.2 Destruir una tabla de símbolos	23
7.3 Crear una nueva entrada	24
7.4 Asignar el tipo de entrada a una entrada	24
7.5 Consultar el tipo de entrada de una entrada	24
7.6 Crear un atributo entero en una entrada	25
7.7 Crear un atributo cadena en una entrada	25
7.8 Asignar el valor de un atributo entero	26
7.9 Asignar el valor de un atributo cadena	26
7.10 Consultar el valor de un atributo entero	27
7.11 Consultar el valor de un atributo cadena	27
7.12 Determinar la existencia de una tabla de símbolos	28
7.13 Determinar la existencia de una entrada	29
7.14 Determinar la existencia de un atributo	29
7.15 Escribir el contenido actual de una tabla en un fichero	30
7.16 Consultar el ID del último error que se ha producido	30
7.17 Consultar la descripción del último error que se ha producido	30
7.18 Buscar posición de una entrada	31
7.19 Asignar el tipo de entrada a una entrada (por posición)	31
7.20 Consultar el tipo de entrada de una entrada (por posición)	32
7.21 Crear un atributo entero en una entrada (por posición)	32
7.22 Crear un atributo cadena en una entrada (por posición)	33
7.23 Asignar el valor de un atributo entero (por posición)	33
7.24 Asignar el valor de un atributo cadena (por posición)	34
7.25 Consultar el valor de un atributo entero (por posición)	34
7.26 Consultar el valor de un atributo cadena (por posición)	35
7.27 Determinar la existencia de un atributo (por posición)	35
7.28 Resetear la variable que almacena el último error producido.	36
8. Posibles problemas y sus soluciones	37

1. ¿Qué es una tabla de símbolos?

En Informática, una tabla de símbolos es una estructura de datos usada en el proceso de traducción de un lenguaje de programación, por un compilador o un intérprete, donde cada símbolo en el código fuente de un programa está asociado con información tal como la ubicación, el tipo de datos y el ámbito de cada variable, constante o procedimiento.

Una implementación común de una tabla de símbolos puede ser mediante una tabla *hash*.

La tabla de símbolos será mantenida a lo largo de todas las fases del proceso de compilación.

Puede tratarse como una estructura transitoria o volátil, que sea utilizada únicamente en el proceso de traducción de un lenguaje de programación, para luego ser descartada, o integrada en la salida del proceso de compilación para una explotación posterior, como puede ser por ejemplo, durante una sesión de depuración, o como recurso para obtener un informe de diagnóstico durante o después de la ejecución de un programa.

Los símbolos en la tabla de símbolos pueden referirse a variables, a funciones o a tipos de datos en el código fuente de un programa.

Una tabla de símbolos es un tipo abstracto de datos (TAD) que permite al compilador escribir, leer, modificar y buscar información acerca de los identificadores del código fuente a medida que se va leyendo en el proceso de compilación. Generalmente, un compilador necesitará crear varias instancias de tablas de símbolos durante este proceso, una para cada ámbito de identificadores, y podrá destruirlas una vez que ya no necesite su información.

2. Ventajas de esta tabla de símbolos TS2006

Las ventajas de TS2006 respecto a la antigua tabla de símbolos que se ofrecía a los alumnos hasta ahora son las siguientes:

- No produce por sí misma errores ni *warnings* de compilación en ningún compilador que sea compatible con C estándar (en el caso de la versión de C++ tampoco los producirá pero necesitará lógicamente un compilador de C++ estándar)
- Permite consultar el último error.
- En principio, evita al usuario tener que manejar directamente posiciones de memorias internas a la tabla de símbolos, eliminando así de raíz muchos de los posibles *crash* que ocasionarían accesos indebidos, si bien también se han añadido las versiones de acceso directo por posición de memoria, que habrá que manejar con cuidado.
- La interfaz de usuario es más sencilla e intuitiva.
- Ofrece una mayor flexibilidad a la hora de almacenar información sobre los parámetros de las funciones. Ahora no son necesarias funciones especiales para esto.
- La implementación es transparente al alumno. No es imprescindible saber cómo funciona internamente para poder usar las funciones. En el caso de que pese a todo se tenga interés en conocer su implementación, se observará un diseño sencillo y un código bien comentado.
- Se ofrece el código en dos versiones: C estándar y C++

3. Requisitos previos a la utilización de TS2006

Antes de comenzar a usar la tabla de símbolos TS2006 el usuario deberá cumplir los siguientes requisitos:

- Haber recibido clases o haber estudiado al menos un curso básico sobre compiladores. Debe saberse en qué consiste un compilador y tener siempre presente en todo momento cuales son las fases de la compilación y en qué consisten. Es importante tener una idea global del proceso en conjunto para encaminar correctamente el compilador desde el primer momento. Saber de antemano qué es lo que tendrá que consultarse en la tabla de símbolos en las últimas fases es importante para saber qué tiene que almacenarse en las primeras.

- Conocer el lenguaje para el cual se va a crear un compilador. Aspectos como los diferentes ámbitos posibles para las variables y funciones determinarán la forma en la que se debe crear/destruir las tablas y la forma en la que se debe introducir y consultar los lexemas.

- Tener instalado un compilador compatible con C estándar o C++ estándar (según la versión que se vaya a usar) para poder usar TS2006, ya que estas funciones están implementadas en dicho lenguaje. Deben estar configuradas todas las variables de entorno necesarias para compilar programas desde cualquier ruta.

- Opcionalmente, se pueden disponer de herramientas auxiliares para procesar el léxico y la sintaxis, como por ejemplo Flex y Bison.

4. Cómo usar la versión C de TS2006

Para poder usar las funciones de TS2006 (C) necesitará ponerse la siguiente línea en todos los módulos que requieran usar tablas de símbolos:

```
#include "ts2006.h"
```

En el caso de los archivos de Flex y Bison se debe poner esa línea en la sección reservada para los *include* de C. Ver la prueba "prueba01" para un ejemplo.

Se debe compilar el código de ts2006.c junto al resto de los códigos .c del programa. Esto en Visual C++ o en Borland se hace añadiendo los diferentes archivos .c al proyecto. En gcc el comando que se usa para compilar deberá tener este aspecto:

```
gcc -o programa.exe programaprincipal.c unmodulo.c otromodulo.c ts2006.c
```

Cuidado con las mayúsculas y las minúsculas: TS2006 es *case sensitive* (como C). Esto significa que por ejemplo considerará diferentes las cadenas "abc" y "ABC". Si se desea que mayúsculas y minúsculas no tengan importancia para diferenciar los lexemas, entonces hay que hacer siempre una conversión previa de las cadenas que se pasen por parámetro a todo minúsculas o bien siempre convertir todo a mayúsculas.

Las cadenas son la principal fuente de problemas de memoria en C. Para evitar estos problemas, se debe tener en cuenta lo siguiente:

- En la versión C no se manejan excepciones producidas por accesos indebidos a posiciones de memoria liberadas o no inicializadas. ¡Ojo con las funciones de acceso por posición!

- Funciones que requieren cadenas en algún parámetro: En el caso de cadenas para las que has reservado memoria manualmente (con *malloc*, *realloc*…), no se debería liberar (*free*) esta memoria hasta después de las llamadas. Si no se ha tenido que reservar memoria para la cadena que se está pasando como parámetro, no se tiene que liberar nada luego.

- Funciones que devuelven cadenas como valor de retorno: En este caso, lo que realmente devuelven es la posición de memoria del inicio de esta cadena resultado. No intentar liberar la memoria de esta posición obtenida. Hay que limitarse a mostrarlas o pasarlas como parámetros a otras funciones. En el caso de querer modificarlas hay que hacerse una copia primero y hay que trabajar sobre esta copia. En ese caso, se debe liberar luego la memoria de dicha copia, pero no se debe nunca intentar liberar la memoria de una cadena devuelta por una función, o se obtendrán fallos catastróficos (*crash*)

- Las funciones de TS2006 nunca aceptarán un NULL como parámetro.

Es posible que halla que mantener una lista o pila de tablas de símbolos. Para esta tarea hay que tener en cuenta que no se necesita almacenar tablas de símbolos enteras y ni siquiera punteros, sino simplemente los identificadores de las tablas de símbolos, es decir, simples enteros (*int* de C). Existen diferentes TAD para el manejo de listas de enteros y pilas de enteros disponibles en Internet.

Si bien no siempre es imprescindible consultar los valores de retorno de las funciones, éstos podrían ayudar a controlar y localizar los errores. En cualquier caso siempre se pueden usar las siguientes funciones para consultar el último error se ha cometido:

consultar_id_ultimo_error
consultar_descripcion_ultimo_error

5. Cómo usar la versión C++ de TS2006 en tu programa

Para poder usar las funciones de TS2006 (C++) necesitará ponerse la siguiente línea en todos los módulos que requieran usar tablas de símbolos:

```
#include "ts2006.h"
```

Se debe compilar el código de ts2006.cpp junto al resto de los códigos .cpp del programa. Esto, en Visual C++ o en Borland se hace añadiendo los diferentes archivos .cpp al proyecto. En g++ el comando que hay que usar para compilar deberá tener este aspecto:

```
g++ -o programa.exe programaprincipal.cpp unmodulo.cpp otromodulo.cpp
ts2006.cpp
```

Cuidado con las mayúsculas y las minúsculas: TS2006 es *case sensitive* (como C++). Esto significa que por ejemplo considerará diferentes las cadenas "abc" y "ABC". Si se desea que mayúsculas y minúsculas no tengan importancia para diferenciar los lexemas, hay que asegurarse de hacer siempre una conversión previa de las cadenas que se pasen por parámetro a todo minúsculas o bien siempre convertir todo a mayúsculas.

Las cadenas siempre son un tipo de datos problemáticos, aunque en la versión de C++ se ha simplificado su uso mediante el objeto *string*, pero aún así cabe realizar algunas advertencias:

- En C++ no hace falta hacer `new` ni `delete` de un *string* si había sido declarado como variable local. Si lo que se ha declarado fuera un puntero a *string*, entonces sí haría falta, pero no se recomienda hacer esto.

- Las funciones del objeto gestor nunca aceptarán un NULL como parámetro y esto incluye cadenas.

- Un objeto *string* de C++ nunca puede ser NULL. En todo caso, se podría conseguir que un puntero a *string* fuese NULL, pero no se recomienda.

- En el caso de las funciones para consultar cadenas, cuando se produzca un error se devolverá la cadena vacía "" (no es un NULL, ¡ojo!). En algunos casos se debería consultar el último error producido (tras la llamada) para determinar si era un error o si la cadena consultada estaba realmente vacía.

Es posible que halla que mantener una lista o pila de tablas de símbolos. Para esta tarea hay que tener en cuenta que no se necesita almacenar tablas de símbolos enteras y ni siquiera punteros, sino simplemente los identificadores de las tablas de símbolos, es decir, simples enteros (*int* de C++). Existen diferentes TAD para el manejo de listas de enteros y pilas de enteros disponibles en Internet.

Si bien no siempre es imprescindible consultar los valores de retorno de las funciones, éstos podrían ayudar a controlar y localizar los errores. En cualquier caso siempre se pueden usar las siguientes funciones para consultar el último error se ha cometido:

consultar_id_ultimo_error
consultar_descripcion_ultimo_error

6. Funciones C y ejemplos de uso

A continuación se muestra en detalle cada función, incluyendo la entrada, salida, una descripción y un ejemplo en C.

6.1 Crear una tabla de símbolos	
Cabecera:	int crear_tabla();
Entrada:	<nada>
Salida:	- Identificador de la tabla creada (número positivo) si se creó. - Devuelve -1 si se produce un error.
Descripción:	Crea una nueva tabla de símbolos y devuelve el identificador que se le ha asignado.

Ejemplo:	`int idtabla;`
	`idtabla= crear_tabla();`
	`if (idtabla<0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.2 Destruir una tabla de símbolos	
Cabecera:	int destruir_tabla(int id_tabla);
Entrada:	- Identificador de la tabla que se quiere destruir.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Se destruye la tabla indicada.

Ejemplo:	`int resultadoOp;`
	`resultadoOp=destruir_tabla (idtabla);`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.3 Crear una nueva entrada	
Cabecera:	int crear_entrada(int id_tabla, char *lexema);
Entrada:	- Identificador de la tabla donde se quiere crear la entrada. - Lexema que se quiere insertar en la nueva entrada.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea una nueva entrada en la tabla indicada con el lexema indicado.

Ejemplo:	`int resultadoOp;`
	`resultadoOp=crear_entrada(idtabla, "mientrada");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.4 Asignar el tipo de entrada a una entrada	
Cabecera:	int asignar_tipo_entrada(int id_tabla, char *lexema, char *tipo);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Tipo que se le quiere dar a la entrada.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Asigna un tipo a una entrada existente.

Ejemplo:	`int resultadoOp;`
	`resultadoOp=asignar_tipo_entrada(idtabla, "mientrada", "estetipo");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.5 Consultar el tipo de entrada de una entrada	
Cabecera:	char* consultar_tipo_entrada(int id_tabla, char *lexema);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada.
Salida:	- Devuelve tipo de la entrada si se ha definido uno y no hay errores. - Devuelve NULL si se produce un error.
Descripción:	Consulta el tipo de una entrada.

Ejemplo:	`char* cadenaResultadoOp;`
	`cadenaResultadoOp= consultar_tipo_entrada(idtabla, "mientrada");`
	`if (cadenaResultadoOp == NULL)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.6 Crear un atributo entero en una entrada	
Cabecera:	int crear_atributo_entero(int id_tabla, char *lexema, char *alias_at, int valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo (ejemplos: "desplazamiento", "ancho", "tabla") - Valor inicial para el atributo que estamos creando (ejemplos: 16, 48, 3)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar valores enteros (tipo int de C)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=crear_atributo_entero(idtabla, "mientrada",`
	`"miatributoentero", 1234);`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.7 Crear un atributo cadena en una entrada	
Cabecera:	int crear_atributo_cadena(int id_tabla, char *lexema, char *alias_at, char *valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo (ejemplos: "cadena", "real","cosa") - Valor inicial para el atributo que estamos creando (ejemplos: "algo", "12.345", "123abcd")
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar cadenas (tipo char* de C)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= crear_atributo_cadena(idtabla, "mientrada", "miatributocadena", "valorinicial");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.8 Asignar el valor de un atributo entero	
Cabecera:	int asignar_valor_atributo_entero(int id_tabla, char *lexema, char *alias_at, int valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (un entero)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (entero) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= asignar_valor_atributo_entero(idtabla, "mientrada", "miatributoentero", 456);`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.9 Asignar el valor de un atributo cadena	
Cabecera:	int asignar_valor_atributo_cadena(int id_tabla, char *lexema, char *alias_at, char *valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (una cadena)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (cadena) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= asignar_valor_atributo_cadena(idtabla, "mientrada", "miatributocadena", "nuevovalor");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.10 Consultar el valor de un atributo entero	
Cabecera:	int consultar_valor_atributo_entero(int id_tabla, char *lexema, char *alias_at);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo.
Salida:	- Valor del atributo entero. - Devuelve 0 y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo entero.

Ejemplo:	`int resultadoOp;`
	`resetear_ultimo_error();`
	`resultadoOp=consultar_valor_atributo_entero(idtabla, "mientrada", "miatributoentero");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.11 Consultar el valor de un atributo cadena	
Cabecera:	char* consultar_valor_atributo_cadena(int id_tabla, char *lexema, char *alias_at);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo.
Salida:	- Valor del atributo cadena. - Devuelve NULL y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo cadena.

Ejemplo:	`char* cadenaResultadoOp;`
	`cadenaResultadoOp=consultar_valor_atributo_cadena(idtabla,`
	`"mientrada", "miatributocadena");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.12 Determinar la existencia de una tabla de símbolos	
Cabecera:	int existe_tabla(int id_tabla);
Entrada:	- Identificador de la tabla que se quiere consultar.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si una tabla existe (ya fue creada y aun no ha sido destruida)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=existe_tabla(idtabla);`
	`if (resultadoOp = 0)`
	`{`
	`  printf("Existe\n");`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  printf("NO existe\n");`
	`}`
	`else`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.13 Determinar la existencia de una entrada	
Cabecera:	int existe_entrada(int id_tabla, char *lexema);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si una entrada existe (fue ya creada y aun no se ha destruido su tabla)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=existe_entrada(idtabla, "estaentrada");`
	`if (resultadoOp = 0)`
	`{`
	`  printf("Existe\n");`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  printf("NO existe\n");`
	`}`
	`else`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.14 Determinar la existencia de un atributo	
Cabecera:	int existe_atributo(int id_tabla, char *lexema, char *alias_at);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si un atributo (con el alias indicado) existe en una determinada entrada (si existe la entrada y aun su tabla no se ha destruido)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=existe_atributo(idtabla, "mientrada", "esteatributo");`
	`if (resultadoOp = 0)`
	`{`
	`  printf("Existe\n");`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  printf("NO existe\n");`
	`}`
	`else`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.15 Escribir el contenido actual de una tabla en un fichero	
Cabecera:	int escribir_tabla(int id_tabla, char *nombre_fichero);
Entrada:	- Identificador de la tabla de símbolos. - Nombre del fichero donde se quiere escribir su contenido.
Salida:	- Da una salida por fichero representando textualmente el contenido de la tabla indicada en el momento de la llamada. - Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Refleja el contenido actual de una tabla de símbolos en un fichero, de forma textual.

Ejemplo:	`int resultadoOp;`
	`resultadoOp= escribir_tabla(idtabla, "tabla.txt");`
	`if (resultadoOp < 0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.16 Consultar el ID del último error que se ha producido	
Cabecera:	int consultar_id_ultimo_error();
Entrada:	<nada>
Salida:	- Devuelve un entero positivo que representa el último error que se ha producido. - Devuelve 0 si de momento no han ocurrido errores.
Descripción:	Devuelve un valor numérico que identifica el último error que se ha producido.
Ejemplo:	`printf("ID del último error: %d\n",` `consultar_id_ultimo_error());`

6.17 Consultar la descripción del último error que se ha producido	
Cabecera:	char* consultar_descripcion_ultimo_error();
Entrada:	<nada>
Salida:	- Devuelve una cadena indicando la descripción del último error que se ha producido. - Devuelve la cadena vacía ("") si de momento no han ocurrido errores.
Descripción:	Devuelve una descripción textual en lenguaje natural del último error que se ha producido.
Ejemplo:	`printf("Descripción del último error: %s\n",` `consultar_descripcion_ultimo_error());`

6.18 Buscar posición de una entrada	
Cabecera:	int buscar_posicion_entrada(int id_tabla, char *lexema);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada.
Salida:	- Devuelve la posición si existe - Devuelve 0 si no existe.
Descripción:	Determina si una entrada existe (fue ya creada y aun no se ha destruido su tabla) En el caso de que exista, lo que devuelve es su posición.

Ejemplo:	`int posicion;`
	`posicion=buscar_posicion_entrada2(idtabla, "estaentrada");`
	`if (posicion ==0)`
	`{`
	`  printf("No existe esa entrada\n");`
	`}`

6.19 Asignar el tipo de entrada a una entrada (por posición)	
Cabecera:	int asignar_tipo_entrada2(int posicion, char *tipo);
Entrada:	- Posición de la entrada. - Tipo que se le quiere dar a la entrada.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Asigna un tipo a una entrada existente.

Ejemplo:	`int resultadoOp;`
	`resultadoOp=asignar_tipo_entrada2(posicion, "estetipo");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.20 Consultar el tipo de entrada de una entrada (por posición)	
Cabecera:	char* consultar_tipo_entrada2(int posicion);
Entrada:	- Posición de la entrada.
Salida:	- Devuelve tipo de la entrada si se ha definido uno y no hay errores. - Devuelve NULL si se produce un error.
Descripción:	Consulta el tipo de una entrada.

Ejemplo:	`char* cadenaResultadoOp;`
	`cadenaResultadoOp=consultar_tipo_entrada2(posicion);`
	`if (cadenaResultadoOp == NULL)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.21 Crear un atributo entero en una entrada (por posición)	
Cabecera:	int crear_atributo_entero2(int posicion, char *alias_at, int valor);
Entrada:	- Posición de la entrada. - Alias del atributo (ejemplos: "desplazamiento", "ancho","tabla") - Valor inicial para el atributo que estamos creando (ejemplos: 16, 48, 3)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar valores enteros (tipo int de C)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=crear_atributo_entero2(posicion, "miatributoentero", 1234);`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.22 Crear un atributo cadena en una entrada (por posición)	
Cabecera:	int crear_atributo_cadena2(int posicion, char *alias_at, char *valor);
Entrada:	- Posición de la entrada. - Alias del atributo (ejemplos: "cadena", "real","cosa") - Valor inicial para el atributo que estamos creando (ejemplos: "algo", "12.345", "123abcd")
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar cadenas (tipo char* de C)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=crear_atributo_cadena2(posición, miatributocadena", "valorinicial");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.23 Asignar el valor de un atributo entero (por posición)	
Cabecera:	int asignar_valor_atributo_entero2(int posicion, char *alias_at, int valor);
Entrada:	- Posición de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (un entero)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (entero) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=asignar_valor_atributo_entero2(posicion,`
	`"miatributoentero", 456);`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.24 Asignar el valor de un atributo cadena (por posición)	
Cabecera:	int asignar_valor_atributo_cadena2(int posicion, char *alias_at, char *valor);
Entrada:	- Posición de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (una cadena)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (cadena) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= asignar_valor_atributo_cadena2(posicion,`
	`"miatributocadena","nuevovalor");`
	`if (resultadoOp <0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.25 Consultar el valor de un atributo entero (por posición)	
Cabecera:	int consultar_valor_atributo_entero2(int posicion, char *alias_at);
Entrada:	- Posición de la entrada. - Alias del atributo.
Salida:	- Valor del atributo entero. - Devuelve 0 y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo entero.

Ejemplo:	`int resultadoOp;`
	`resetear_ultimo_error();`
	`resultadoOp=consultar_valor_atributo_entero2(posicion,`
	`"miatributoentero");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.26 Consultar el valor de un atributo cadena (por posición)	
Cabecera:	char* consultar_valor_atributo_cadena2(int posicion, char *alias_at);
Entrada:	- Posición de la entrada. - Alias del atributo.
Salida:	- Valor del atributo cadena. - Devuelve NULL y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo cadena.

Ejemplo:	`char* cadenaResultadoOp;`
	`cadenaResultadoOp=consultar_valor_atributo_cadena2(posicion,`
	`"miatributocadena");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.27 Determinar la existencia de un atributo (por posición)	
Cabecera:	int existe_atributo2(int posicion, char *alias_at);
Entrada:	- Posición de la entrada. - Alias del atributo.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si un atributo (con el alias indicado) existe en una determinada entrada (si existe la entrada y aun su tabla no se ha destruido)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=existe_atributo2(posicion, "esteatributo");`
	`if (resultadoOp = 0)`
	`{`
	`  printf("Existe\n");`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  printf("NO existe\n");`
	`}`
	`else`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

6.28 Resetear la variable que almacena el último error producido.	
Cabecera:	void resetear_ultimo_error();
Entrada:	<nada>
Salida:	<nada>
Descripción:	Resetea la variable interna que indica el último error que se ha producido.

Ejemplo:	`int resultadoOp;`
	`resetear_ultimo_error();`
	`resultadoOp=consultar_valor_atributo_entero(idtabla, "mientrada",`
	`"miatributoentero");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  printf("Error: %s\n",consultar_descripcion_ultimo_error());`
	`}`

7. Funciones C++ y ejemplos de uso

A continuación se mostrará en detalle cada función, incluyendo la entrada, salida, una descripción y un ejemplo en C++.

En la versión de C++ en principio se puede manejar de forma similar a la de C, mediante el objeto ts2006, pero también se podría crear directamente los objetos tabla, entrada y atributo que se necesiten y gestionarlos de forma personalizada si se ve más conveniente de ese modo.

Para todos los ejemplos, se supone que se han hecho los *#include* necesarios *(ver apartado 5)* y que se ha declarado un objeto ts2006 al que se denominará gestor. Hay que recordar destruirlo al final de la compilación para liberar la memoria de todo lo que quede: se liberarán (y perderán) todas las tablas, entradas y atributos que se crearon a través de ese gestor y que aún no estén destruidos.

```
ts2006* gestor;
gestor=new ts2006();
// operaciones
delete gestor;
```

7.1 Crear una tabla de símbolos	
Cabecera:	int ts2006::crear_tabla();
Entrada:	<nada>
Salida:	- Identificador de la tabla creada (número positivo) si se creó. - Devuelve -1 si se produce un error.
Descripción:	Crea una nueva tabla de símbolos y devuelve el identificador que se le ha asignado.

Ejemplo:	`int idtabla;`
	`idtabla=gestor->crear_tabla();`
	`if (idtabla<0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.2 Destruir una tabla de símbolos	
Cabecera:	int ts2006::destruir_tabla(int id_tabla);
Entrada:	- Identificador de la tabla que se quiere destruir.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Se destruye la tabla indicada.

Ejemplo:	`int resultadoOp;`
	`resultadoOp=gestor->destruir_tabla (idtabla);`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.3 Crear una nueva entrada	
Cabecera:	int ts2006::crear_entrada(int id_tabla, string lexema);
Entrada:	- Identificador de la tabla donde se quiere crear la entrada. - Lexema que se quiere insertar en la nueva entrada.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea una nueva entrada en la tabla indicada con el lexema indicado.

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->crear_entrada(idtabla, "mientrada");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.4 Asignar el tipo de entrada a una entrada	
Cabecera:	int ts2006::asignar_tipo_entrada(int id_tabla, string lexema, string tipo);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Tipo que se le quiere dar a la entrada.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Asigna un tipo a una entrada existente.

Ejemplo:	`int resultadoOp;`
	`resultadoOp=gestor->asignar_tipo_entrada(idtabla, "mientrada",`
	`"estetipo");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.5 Consultar el tipo de entrada de una entrada	
Cabecera:	string ts2006::consultar_tipo_entrada(int id_tabla, string lexema);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada.
Salida:	- Devuelve tipo de la entrada si se ha definido uno y no hay errores. - Devuelve "" si se produce un error o la entrada no esta inicializada
Descripción:	Consulta el tipo de una entrada.

Ejemplo:	`string cadenaResultadoOp;`
	`cadenaResultadoOp= gestor->consultar_tipo_entrada(idtabla,`
	`"mientrada");`
	`if (gestor->consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.6 Crear un atributo entero en una entrada	
Cabecera:	int ts2006::crear_atributo_entero(int id_tabla, string lexema, string alias_at, int valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo (ejemplos: "desplazamiento", "ancho","tabla") - Valor inicial para el atributo que estamos creando (ejemplos: 16, 48, 3)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar valores enteros (tipo int de C++)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->crear_atributo_entero(idtabla, "mientrada", "miatributoentero", 1234);`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.7 Crear un atributo cadena en una entrada	
Cabecera:	int ts2006::crear_atributo_cadena(int id_tabla, string lexema, string alias_at, string valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo (ejemplos: "cadena", "real","cosa") - Valor inicial para el atributo que estamos creando (ejemplos: "algo", "12.345", "123abcd")
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar cadenas (tipo string de C++)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=gestor->crear_atributo_cadena(idtabla, "mientrada", "miatributocadena", "valorinicial");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.8 Asignar el valor de un atributo entero	
Cabecera:	int ts2006::asignar_valor_atributo_entero(int id_tabla, string lexema, string alias_at, int valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (un entero)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (entero) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=gestor->asignar_valor_atributo_entero(idtabla, "mientrada", "miatributoentero", 456);`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.9 Asignar el valor de un atributo cadena	
Cabecera:	int ts2006::asignar_valor_atributo_cadena(int id_tabla, string lexema, string alias_at, string valor);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (una cadena)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (cadena) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=gestor->asignar_valor_atributo_cadena(idtabla, "mientrada", "miatributocadena", "nuevovalor");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.10 Consultar el valor de un atributo entero	
Cabecera:	int ts2006::consultar_valor_atributo_entero(int id_tabla, string lexema, string alias_at);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo.
Salida:	- Valor del atributo entero. - Devuelve 0 y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo entero.

Ejemplo:	`int resultadoOp;`
	`gestor->resetear_ultimo_error();`
	`resultadoOp=gestor->consultar_valor_atributo_entero(idtabla,`
	`"mientrada", "miatributoentero");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.11 Consultar el valor de un atributo cadena	
Cabecera:	string ts2006::consultar_valor_atributo_cadena(int id_tabla, string lexema, string alias_at);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo.
Salida:	- Valor del atributo cadena. - Devuelve "" y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo cadena.

Ejemplo:	`string cadenaResultadoOp;`
	`cadenaResultadoOp=gestor->consultar_valor_atributo_cadena`
	
	`(idtabla,"mientrada","miatributocadena");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.12 Determinar la existencia de una tabla de símbolos	
Cabecera:	int ts2006::existe_tabla(int id_tabla);
Entrada:	- Identificador de la tabla que se quiere consultar.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si una tabla existe (ya fue creada y aun no ha sido destruida)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->existe_tabla(idtabla);`
	`if (resultadoOp = 0)`
	`{`
	`  cout << "Existe" << endl;`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  cout << "NO existe" << endl;`
	`}`
	`else`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.13 Determinar la existencia de una entrada	
Cabecera:	int ts2006::existe_entrada(int id_tabla, string lexema);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si una entrada existe (fue ya creada y aun no se ha destruido su tabla)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->existe_entrada(idtabla,"estaentrada");`
	`if (resultadoOp = 0)`
	`{`
	`  cout << "Existe" << endl;`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  cout << "NO existe" << endl;`
	`}`
	`else`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.14 Determinar la existencia de un atributo	
Cabecera:	int ts2006::existe_atributo(int id_tabla, string lexema, string alias_at);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada. - Alias del atributo.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si un atributo (con el alias indicado) existe en una determinada entrada (si existe la entrada y aun su tabla no se ha destruido)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->existe_atributo(idtabla, "mientrada",`
	`"esteatributo");`
	`if (resultadoOp = 0)`
	`{`
	`  cout << "Existe" << endl;`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`  cout << "NO existe" << endl;`
	`}`
	`else`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.15 Escribir el contenido actual de una tabla en un fichero	
Cabecera:	int ts2006::escribir_tabla(int id_tabla, string nombre_fichero);
Entrada:	- Identificador de la tabla de símbolos. - Nombre del fichero donde se quiere escribir su contenido.
Salida:	- Da una salida por fichero representando textualmente el contenido de la tabla indicada en el momento de la llamada. - Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Refleja el contenido actual de una tabla de símbolos en un fichero, de forma textual.

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->escribir_tabla(idtabla, "tabla.txt");`
	`if (resultadoOp < 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.16 Consultar el ID del último error que se ha producido	
Cabecera:	int ts2006::consultar_id_ultimo_error();
Entrada:	<nada>
Salida:	- Devuelve un entero positivo que representa el último error que se ha producido. - Devuelve 0 si de momento no han ocurrido errores.
Descripción:	Devuelve un valor numérico que identifica el último error que se ha producido.
Ejemplo:	`cout << "ID del último error:";` `cout << gestor->consultar_id_ultimo_error() << endl;`

7.17 Consultar la descripción del último error que se ha producido	
Cabecera:	string ts2006::consultar_descripcion_ultimo_error();
Entrada:	<nada>
Salida:	- Devuelve una cadena indicando la descripción del último error que se ha producido. - Devuelve la cadena vacía ("") si de momento no han ocurrido errores.
Descripción:	Devuelve una descripción textual en lenguaje natural del último error que se ha producido.

Ejemplo:	`cout << "Descripción del último error:";`
	`cout << gestor->consultar_descripcion_ultimo_error();`
	`cout << endl;`

7.18 Buscar posición de una entrada	
Cabecera:	int ts2006::buscar_posicion_entrada(int id_tabla, string lexema);
Entrada:	- Identificador de la tabla de símbolos de la entrada. - Lexema de la entrada.
Salida:	- Devuelve la posición si existe - Devuelve 0 si no existe.
Descripción:	Determina si una entrada existe (fue ya creada y aun no se ha destruido su tabla) En el caso de que exista, lo que devuelve es su posición.

Ejemplo:	`int posicion;`
	`posicion= gestor->buscar_posicion_entrada(idtabla, "estaentrada");`
	`if (posicion ==0)`
	`{`
	`  cout << "No existe esa entrada" << endl;`
	`}`

7.19 Asignar el tipo de entrada a una entrada (por posición)	
Cabecera:	int ts2006:asignar_tipo_entrada2(int posicion, string tipo);
Entrada:	- Posición de la entrada. - Tipo que se le quiere dar a la entrada.
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Asigna un tipo a una entrada existente.

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->asignar_tipo_entrada2(posicion, "estetipo");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.20 Consultar el tipo de entrada de una entrada (por posición)	
Cabecera:	string ts2006:consultar_tipo_entrada2(int posicion);
Entrada:	- Posición de la entrada.
Salida:	- Devuelve tipo de la entrada si se ha definido uno y no hay errores. - Devuelve "" y genera error si se produce un error.
Descripción:	Consulta el tipo de una entrada.

Ejemplo:	`string cadenaResultadoOp;`
	`cadenaResultadoOp= gestor->consultar_tipo_entrada2(posicion);`
	`if (gestor->consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.21 Crear un atributo entero en una entrada (por posición)	
Cabecera:	int ts2006:crear_atributo_entero2(int posicion, string alias_at, int valor);
Entrada:	- Posición de la entrada. - Alias del atributo (ejemplos: "desplazamiento", "ancho","tabla") - Valor inicial para el atributo que estamos creando (ejemplos: 16, 48, 3)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar valores enteros (tipo int de C++)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->crear_atributo_entero2(posicion,`
	`"miatributoentero", 1234);`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.22 Crear un atributo cadena en una entrada (por posición)	
Cabecera:	int ts2006:crear_atributo_cadena2(int posicion, string alias_at, string valor);
Entrada:	- Posición de la entrada. - Alias del atributo (ejemplos: "cadena", "real","cosa") - Valor inicial para el atributo que estamos creando (ejemplos: "algo", "12.345", "123abcd")
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Crea un nuevo atributo en la entrada, con el alias indicado. Este atributo solo podrá almacenar cadenas (tipo string de C++)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->crear_atributo_cadena2(posición, miatributocadena", "valorinicial");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.23 Asignar el valor de un atributo entero (por posición)	
Cabecera:	int ts2006:asignar_valor_atributo_entero2(int posicion, string alias_at, int valor);
Entrada:	- Posición de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (un entero)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (entero) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->asignar_valor_atributo_entero2(posicion, "miatributoentero", 456);`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.24 Asignar el valor de un atributo cadena (por posición)	
Cabecera:	int ts2006:asignar_valor_atributo_cadena2(int posicion, string alias_at, string valor);
Entrada:	- Posición de la entrada. - Alias del atributo. - Nuevo valor que se quiere escribir en ese atributo (una cadena)
Salida:	- Devuelve 0 si la operación se realizó correctamente. - Devuelve -1 si se produce un error.
Descripción:	Da un valor al atributo (cadena) de alias indicado en la entrada indicada (sobrescribe el valor antiguo)

Ejemplo:	`int resultadoOp;`
	`resultadoOp=gestor->asignar_valor_atributo_cadena2(posicion, "miatributocadena", "nuevovalor");`
	`if (resultadoOp <0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.25 Consultar el valor de un atributo entero (por posición)	
Cabecera:	int ts2006:consultar_valor_atributo_entero2(int posicion,string alias_at);
Entrada:	- Posición de la entrada. - Alias del atributo.
Salida:	- Valor del atributo entero. - Devuelve 0 y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo entero.

Ejemplo:	`int resultadoOp;`
	`gestor->resetear_ultimo_error();`
	`resultadoOp= gestor->consultar_valor_atributo_entero2(posicion, "miatributoentero");`
	`if (gestor->consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.26 Consultar el valor de un atributo cadena (por posición)	
Cabecera:	string ts2006:consultar_valor_atributo_cadena2(int posicion,string alias_at);
Entrada:	- Posición de la entrada. - Alias del atributo.
Salida:	- Valor del atributo cadena. - Devuelve "" y genera un error si se produce error.
Descripción:	Consulta el valor de un atributo cadena.

Ejemplo:	`string cadenaResultadoOp;`
	`cadenaResultadoOp=gestor->consultar_valor_atributo_cadena2(posicion, "miatributocadena");`
	`if (gestor->consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.27 Determinar la existencia de un atributo (por posición)	
Cabecera:	int ts2006:existe_atributo2(int posicion, string alias_at);
Entrada:	- Posición de la entrada. - Alias del atributo.
Salida:	- Devuelve 0 si existe. - Devuelve 1 si no existe. - Devuelve -1 si se produce un error.
Descripción:	Determina si un atributo (con el alias indicado) existe en una determinada entrada (si existe la entrada y aun su tabla no se ha destruido)

Ejemplo:	`int resultadoOp;`
	`resultadoOp= gestor->existe_atributo2(posicion, "esteatributo");`
	`if (resultadoOp = 0)`
	`{`
	`    cout << "Existe" << endl;`
	`}`
	`else if (resultadoOp =1)`
	`{`
	`    cout << "NO existe" << endl;`
	`}`
	`else`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

7.28 Resetear la variable que almacena el último error producido.	
Cabecera:	void resetear_ultimo_error();
Entrada:	<nada>
Salida:	<nada>
Descripción:	Resetea la variable interna que indica el último error que se ha producido.

Ejemplo:	`int resultadoOp;`
	`gestor->resetear_ultimo_error();`
	`resultadoOp=gestor->consultar_valor_atributo_entero(idtabla,`
	`"mientrada", "miatributoentero");`
	`if (consultar_id_ultimoerror() != 0)`
	`{`
	`  cout << "Error:";`
	`  cout << gestor->consultar_descripcion_ultimo_error();`
	`  cout << endl;`
	`}`

8. Posibles problemas y sus soluciones

Problema: No se compilan los programas básicos, ni tampoco las pruebas

Solución: Hay que asegurarse de que el compilador esté bien instalado y que las variables de entorno estén bien configuradas. En el caso del gcc, si no se puede cambiar la configuración, entonces crear el siguiente archivo .bat:

```
set path=C:\djgpp\bin;%PATH%
set djgpp=C:\djgpp\djgpp.env
```

Nota: este .bat debería ser ejecutado en modo terminal y gcc debería ser usado en un terminal (también llamado símbolo del sistema o ventana MS-DOS en Windows).

Las variables de entorno también se pueden gestionar en el sistema operativo de forma que se queden configuradas de forma permanente incluso aunque se reinicie el ordenador. La forma exacta de hacerlo depende de tu versión del sistema operativo.

Problema: No se compilan bien programas que usan Flex , Bison y TS2006.

Solución: Ambos programas deben estar bien instalados y deben ser accesibles. La versión de Flex (Lex) que se elija debe ser compatible con la versión de Bison (yacc) que se elija. Las versiones disponibles en la página web de la asignatura deberían funcionar bien juntas. Si no se puede cambiar la configuración del sistema (variables de entorno), se puede simplemente copiar ambos programas en la misma ubicación del código fuente de la práctica.

Problema: El programa termina de forma anormal (*crash*) de forma más o menos aleatoria.

Solución: Hay que asegurarse de no estar liberando cadenas que sean usadas más adelante. Probar a comentar todas las instrucciones *free* que halla y comprobar que ahora no se producen *crash* (aunque entonces surgirán lógicamente agujeros de memoria). Luego hay que ir descomentándolos uno por uno hasta que se encuentre al responsable de los *crash*. También hay que asegurarse de no estar usando variables cadenas que no hayan sido inicializadas ni *buffers* que no tengan un carácter terminador '\0', ni que se sobrepase el tamaño reservado. Por último, comprobar que la cantidad de memoria que hay disponible al comienzo y al final del programa coinciden. Dependiendo de la versión concreta de tu compilador de C o C++, esto se hará de una forma u otra.

Problema: Se necesita hacer una referencia a una tabla, pero no se dispone de funciones que devuelvan un puntero a ésta.

Solución: Usar el identificador de la tabla. No hay necesidad alguna de manejar punteros a la memoria interna de las tablas.

Problema: Se necesita almacenar en un atributo algo que no es ni *char ni *int* pero no hay funciones para ello.**

Solución: Es muy raro que realmente se necesite almacenar otra cosa. No obstante, si realmente es necesario, cualquier tipo puede almacenarse en un *char** o en un *int*. Por ejemplo, un valor lógico puede almacenarse como los enteros 0 ó 1. Un valor real puede almacenarse como cadena ("123.456"). Un valor entero muy grande puede almacenarse como cadena ("12345678901234567890"). Un registro puede ser almacenado en otra tabla de símbolos y lo que se debería almacenar aquí por tanto sería su identificador (*int*).

Problema: No se puede consultar una entrada que había insertado previamente.

Solución: Recuerda que las funciones son *case sensitive* y tienes que tener cuidado con las mayúsculas y minúsculas. Si insertaste un lexema "algo" e intentas consultar el lexema "ALGO", generalmente la función indicará que no ha encontrado nada.

Problema: Se ha encontrado un *bug* de TS2006

Solución: Si se está seguro de que es realmente un bug de TS2006 y no del práctica, se puede reportar al email del creador de TS2006: daniel@blackdtools.com o al webmaster de la asignatura de Compiladores, indicando "TS2006" en el *subject* de dicho email. En el caso de *bugs* que ya han sido corregidos, se recibirá directamente por email un *link* para descargar la versión más actualizada de TS2006. En el caso de otros *bugs*, habrá que esperar, puede que varias semanas, hasta que sea corregido. En cualquier caso, se es libre de corregir directamente el *bug* si no se puede esperar, y se es libre de añadir o modificar la funcionalidad de TS2006 según convenga para la práctica. En este último caso, los compañeros agradecerán que se envíe al creador la versión corregida, indicando qué fallaba, por qué y qué modificaciones se han realizado.

Problema: gcc no compila el programa usando la versión C++

Solución: algunas versiones de gcc parecen no ser compatibles con el nuevo estándar C++: lo mejor es instalar g++, que se usa de la misma forma que gcc pero es totalmente compatible con el nuevo estándar C++ y se ha comprobado que compila correctamente y sin *warnings* el código y las pruebas de la versión C++ de la tabla de símbolos.

Problema: La misma función me provoca un *crash* en el programa en la versión C, pero no lo hace en la versión C++

Solución: Seguramente se está haciendo un acceso indebido por posición de memoria a una posición liberada o no inicializada. En la versión C no se realiza manejo de excepciones, por lo que provocará un *crash* o dará resultados impredecibles. Sin embargo en la versión C++ las funciones de acceso por memoria están protegidas contra su mal uso (mediante *try* / *catch*) con lo cual sólo devolverá un código de error en estos casos, pero el programa no terminará de forma anormal.

Problema: Los acentos no se muestran correctamente en la consola MS-DOS

Solución: El código fuente fue editado en Windows y en principio no se muestran los acentos correctamente en MS-DOS. Para solucionarlo se deben exportar todos los ficheros de código fuente al formato que usa MS-DOS. Por ejemplo, en con el wordpad habrá que guardarlo como "*Documento de texto – formato de MSDOS*"