Laboratory 5 – Electronic Diary

The CSV (Comma Separated Value) file format is a simple and convenient way to store a wide variety of data.

In this laboratory, we use a CSV file to store events of a personal diary. A similar method using CSV is used in Microsoft Outlook to memorise personal calendars.

Our command-line application can handle a list of "events" consisting in a date/time timestamp and a description. The commands to manage the events are the following:

- `load`: prompts the user for a file name. If the filename indicated by the user exists in the local directory, all the events are loaded and merged with the existing events. The application indicates the number of events loaded.
- `list`: prints on screen the list of current events. For each event, the command prints its sequential number starting from 1, its timestamp (date/time), and its description.
- `new`: adds an event to the list. The command prompts the user for the day of the month (1-31), the month (1-12), the hours (0-23), the minutes (0-59), and a textual description (title) of the event. The new event is added to the list in chronological order.
- `delete`: prompts the user for the event number to delete. The event is removed from the list and the corresponding allocated memory is freed.
- `save`: prompts the user for a file name to save the current schedule in a file in the local directory.

The printout on the right shows an example of a session.

```
>load
  filename: appointment.txt
Loaded 3 events from
appointment.txt

>list
1 Tue Nov 22 17:30:29 2022
basket

2 Mon Nov 21 10:30:29 2022
surf

3 Sun Nov 20 09:00:29 2022
dentist

>new
  day   [17]: 29
  month [10] :10
  hours [19]: 20
  mins  [01]: 00
  title: dinner

>list
1 Tue Nov 22 17:30:29 2022
basket

2 Mon Nov 21 10:30:29 2022
surf

3 Sun Nov 20 09:00:29 2022
dentist

4 Sat Oct 29 20:00:14 2022
dinner

>delete
  event: 1

>list
1 Mon Nov 21 10:30:29 2022
surf

2 Sun Nov 20 09:00:29 2022
dentist

3 Sat Oct 29 20:00:14 2022
dinner

>save
  filename: appointment.txt
Diary saved in appointment.txt

>exit
```

Description

The data is represented internally as a linked list with the following declaration:

```
typedef struct event {
    time_t time;
    char* desc;
    struct event* next;
} event;

event* head = NULL;
```

The struct event includes a timestamp *time_t* representing the number of seconds from a past time reference (the Epoch - 01/01/1970), a *char* desc* that references a description string, and a pointer to the next element of the array. The typedef *time_t* as well as the functions manipulating timestamps are available in the standard C library time.h.

The global variable *head* maintains a reference to the list (initially empty).

Part 1

The linked list is manipulated by the following functions whose prototype and description is given in the table below.

Function Prototype	Description
`void add_event(time_t a, char* ed);`	Adds an entry to the list initialising the time with the first argument of the function and the pointer desc to reference the string *ed*. NB: The memory for the new string must be allocated using the malloc() and copied using strcpy(). The entry is then inserted in the list maintaining the chronological order.
`void print_diary();`	Prints the list of entries, specifying sequence number, date/time, and description. The function ctime() (in time.h) can be used to print the date/time in full from the timestamp in seconds.
`void save_diary(char* fname);`	Saves the list into the filename in the local directory provided as argument. If the file cannot be opened for writing, an error message should be printed. The entry should be printed in CSV format as outlined below.
`void load_diary(char* fname);`	Loads all entries from the filename in the local directory given as argument. If the file cannot be opened for reading, a message should be printed. The entries are read as CSV fields (see below).
`void delete(int k);`	This function deletes the k-the entry (between 1 and N). If an entry number outside the interval is given, no deletion occurs.

A CSV file stores each event in a separated line of text. The fields of an CSV entry are separated by comas and can contain any character (including spaces) except the coma. The following listing represents a valid CSV file with three entries.

```
Timestamp,Description
1669026629,going for surfing
1668934829,dentist appointment
1667070014,dinner for two
```

The first line usually contains the names of the fields also separated by comas (in this case two fields “Timestamp” and “Description”).

To read a CSV file we can use the function scanf() with the usual formatters (%d for integers, %f for floats, %s for strings, etc). However, if the string contains spaces, the formatter %s cannot be used and the more sophisticated syntax using the formatter %[ ] is needed.

The formatter %[ ] matches all the strings with the list of characters between square brackets. For example, %[ab] matches strings with alternated sequences of "a" and "b", such as "abbab" or "bbbaaab". However, if the characters in brackets are prefixed by a caret (^), the formatter matches all the characters *except* those in brackets. For example, %[^ab] matches any sequence of characters that do not contain "a" or "b".

Thus, the formatter %[^,] matches all the strings between comas, and %[^\n] matches all characters in the last field up to the newline. Since individual characters in scanf() formatters match themselves, the following formatter matches the requested syntax:

```
"%ld,%[^\n]\n"
```

A scanf() formatter is part of a broader pattern matching grammar called Regular Expressions.

Part 2

The functions in Part 1 are called by a parser of user commands implemented in the main(). The infinite loop receiving user commands in main() should do the following:

1. Prompt the user for a command and store the command in a buffer (char buffer[]).
2. Compare the buffer with any of the commands using *strcmp()*.
3. If a matched command requires additional input, request extra input in the body of the if-statement. As the user provides the info needed, call the corresponding function.
4. If no command is matched, print "unknown command" and restart the cycle.

The command "new" requires a date/time to be converted into a *time_t* value. The library time.h helps this process by providing the *struct tm* with a large number of useful date/time fields:

```
int tm_sec;     /* seconds (0 - 60) */
int tm_min;     /* minutes (0 - 59) */
int tm_hour;    /* hours (0 - 23) */
int tm_mday;    /* day of month (1 - 31) */
int tm_mon;     /* month of year (0 - 11) */
int tm_year;    /* year - 1900 */
int tm_wday;    /* day of week (Sunday = 0) */
int tm_yday;    /* day of year (0 - 365) */
int tm_isdst;   /* is summer time in effect? */
char *tm_zone;  /* abbreviation of timezone name */
long tm_gmtoff; /* offset from UTC in seconds */
```

The program should initialise the requested fields (day, month, hour, & mins) of a *struct tm* variable and convert the *struct tm* into a *time_t* before calling *add_event()*. The function timelocal() does exactly the job of converting a *struct tm* into a *time_t* variable:

```
time_t* timelocal(struct tm* timeptr)
```

Note that this function operates on references (pointers) not the structure or time_t to avoid passing the entire structure. Thus, the structures must exist before calling the function. The function localtime() does the opposite conversion instead:

```
struct tm* localtime(time_t* timeptr)
```