LIBRERIA “TUTTO IN UNO”

di Aldo Carpanelli – v1.0.0, 25/10/2023

Cos'è

Una libreria in C per uso personale che comprende alcune tra le funzioni che uso con una certa frequenza.

Include molti file sia dalla libreria standard del C, sia da *Win32*.
Eccone l'elenco:

```
#include <windows.h>  #include <stdlib.h>  #include <ctype.h>
#include <commdlg.h>  #include <stdint.h>  #include <stdarg.h>
#include <stdio.h>    #include <string.h>  #include <locale.h>
```

Per via dell'impiego di API da *Win32*, occorre collegare `comdlg32` e `gdi32` tramite il *linker*.

Tipi di dati

Per usare la funzione `T1_TracciaAreaTesto()` viene definito il tipo `T1_AreaTesto_struct`:

```
typedef struct {
    // il riquadro che contiene l'intera area
    INT x, y, w, h;
    // il raggio di arrotondamento degli spigoli
    INT arrotondamento_orizzontale, arrotondamento_verticale;
    // i margini tra bordo e testo
    INT margine_sinistro, margine_destro;
    INT margine_superiore, margine_inferiore;
    // l'allineamento del testo
    INT allineamento_orizzontale, allineamento_verticale;
    // il testo da tracciare
    LPCSTR testo;
} T1_AreaTesto_struct;
```

Per poter impiegare il tipo `ARGB` senza includere la libreria di sistema GDI+ (non adatta all'uso in C), qualora fosse necessario viene ridefinito…

```
#ifndef ARGB
typedef COLORREF ARGB;
#endif
```

Macro

Sono fornite due macro:

```
#define T1_KEYDOWN(vk_code) \
        ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
```

Restituisce `1` se il tasto `vk_code` è premuto, `0` in caso contrario.

```
#define T1_KEYUP(vk_code) \
        ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)
```

Restituisce `1` se il tasto `vk_code` **non** è premuto, `0` in caso contrario.

Funzioni correlate all'uso della console del C

```
int T1_alternativa( const char *msg, int opzione_a, int opzione_b );
```

I parametri `opzione_a` e `opzione_b` devono corrispondere ai valori di due caratteri visualizzabili ed inseribili come *input* in *console*.

Restituisce `opzione_a` o `opzione_b`, a seconda di quale tra le opzioni viene scelta in *console*.

Non ammette l'immissione di risposte non univoche (o esattamente `opzione_a`, o esattamente `opzione_b`). In caso di risposte non univocamente accettabili insiste ostinatamente nel chiedere l'immissione di una risposta valida.

```
void T1_attendi_invio( const char *msg );
```

Presentato il messaggio puntato da `msg`, si pone in attesa di una qualsiasi immissione da parte dell'utente. In uscita, il *buffer* di *stdin* è vuoto.

```
long long T1_chiedi_intero_da_console( void );
```

Con `fgets()` e un *buffer* da 256 caratteri rileva una stringa da *stdin*.

Se la stringa inserita non supera le dimensioni del *buffer*, usa `strtoll()` per convertirla in un valore di tipo `long long`, che viene restituito come valore di ritorno. In caso di errore, invia a *stdout* messaggi che descrivono cosa è andato storto, quindi chiede un nuovo inserimento. La funzione non ritorna fino a quando l'acquisizione del valore `long long` non ha avuto successo.

```
void T1_localizzazione_italiana( void );
```

Imposta la localizzazione italiana per le applicazioni tramite *console* del C.

```
int T1_menu_va( const char *titolo, int qVoci, ... );
```

Mostra nella *console* del C un menu costituito da un titolo (opzionale) seguito da una riga separatrice e da una serie di voci numerate. Assicura la restituzione di un valore coerente con le caratteristiche del menu, insistendo con le richieste finché non si sceglie una voce di menu valida. Ad esempio…

```
int scelta = T1_menu_va( "MENU DI PROVA", 4,
                         "Uscita", "Voce 1", "Voce 2", "Voce 3");
```

La chiamata appena riportata dà come risultato questo menu:

```
  MENU DI PROVA
================
  1. Voce 1
  2. Voce 2
  3. Voce 3
  0. Uscita
Scegli una voce dal menu:
```

Un menu di questo tipo, restituisce sempre un valore compreso tra 0 e 3, a meno che siano stati mal formattati i parametri.

PARAMETRI

`const char *titolo`	Una stringa C che viene usata come titolo del menu. Dopo il titolo, viene tracciata una riga separatrice costituita da una serie di caratteri `'='`. Se questo parametro è `NULL`, il menu viene presentato senza un titolo, e la riga separatrice non viene tracciata.
`int qVoci`	Un valore intero `>=1` che indica la quantità delle voci complessivamente presenti nel menu (inclusa quella di uscita).
`const char* ...`	Una serie di puntatori a stringhe C, in quantità pari al valore indicato in `qVoci`. La prima stringa viene usata come etichetta per la voce di uscita dal menu, e collocata in coda al menu stesso, numerandola con `0`. Ciascuna delle stringhe successive viene usata come

	etichetta delle altre voci, numerandole a partire da `1`, secondo il loro ordine di successione tra i parametri. Nessuno dei puntatori può essere `NULL`.

Restituisce un `int` che è `0` se è stata scelta la voce riservata all'uscita, `>0` se è stata scelta un'altra voce, sempre compresa entro quelle definite tramite il secondo parametro e quelli successivi. Il valore di errore `-1` viene restituito solo se il secondo parametro è `<1` o anche uno solo tra i parametri successivi è `NULL`.

Funzioni correlate all'uso delle stringhe C

```
int T1_caratteri_univoci_da_stringa( const char *sIn, char *sOut );
```

Analizza la stringa `sIn` e colloca in `sOut` l'elenco dei caratteri univoci che compaiono in essa, nello stesso ordine in cui si presentano la prima volta.

Esempio: `sIn  => "Cane non mangia cane"`
`sOut => "Cane omgic"`

La funzione non fa nulla se `sIn`, `sOut` o entrambi sono `NULL`.

Se tutto va bene restituisce `1` (*true*), altrimenti restituisce `0` (*false*).

NOTA: lo spazio di memoria puntato da `sOut` dev'essere almeno 256 *byte.* La capacità di `sOut` **non** viene viene verificata dalla funzione.

```
int T1_converti_frazione_in_double( const char *str, double *d );
```

Riceve in `str` una stringa C che si suppone rappresenti una frazione e ne effettua internamente una copia nella quale sostituisce tutti i caratteri `','` con caratteri `'.'`.

Applica `strtod()` alla copia di `str` per ricavarne il numeratore.

Salta gli eventuali spazi presenti dopo il numeratore.

Verifica che sia presente il carattere che indica l'operatore `'/'`.

Applica `strtod()` a ciò che segue il carattere `'/'` per ricavarne il denominatore.

Verifica che non esistano caratteri "spuri" dopo il denominatore.

Se numeratore e denominatore sono stati "estratti" correttamente, colloca in `*d` l'esito dell'operazione numeratore/denominatore.

In caso di successo delle operazioni restituisce `1` (come per *true*).

In caso di insuccesso restituisce `0` (come per *false*).

La funzione fallisce se:

- `str` è `NULL` o `d` è `NULL`
- risulta impossibile effettuare la copia temporanea di `str`
- `strtod()` fallisce nel rilevare il numeratore o il denominatore
- il carattere dell'operatore `'/'` è assente o non è collocato correttamente
- `str` contiene caratteri "spuri" estranei alla frazione da convertire
- il denominatore estratto è `0.0`

```
char *T1_copia_elemento( const char *elementi,
                         char separatore, int indice );
```

Con la funzione `T1_trova_elemento()` individua l'elemento richiesto nell'ambito della stringa C `elementi` (che si suppone contenga una lista di sottostringhe separate dal carattere indicato tramite il parametro `separatore`), quindi ne effettua una copia in uno spazio di memoria dinamica appositamente allocato con `malloc()`.

Se la funzione ha successo restituisce il puntatore allo spazio di memoria dinamica nel quale è stato copiato l'elemento richiesto (spetta al chiamante liberare la memoria allocata con `free()`).

Se la funzione fallisce restituisce `NULL`.

```
void T1_elimina_caratteri_multipli_adiacenti( char *str, char c );
```

Analizza la stringa `str` in cerca delle ricorrenze di caratteri `c` multipli adiacenti. Conserva uno solo dei caratteri `c` per ogni gruppo di caratteri adiacenti.

```
void T1_elimina_occorrenze_stringa( char *str,
                                    const char *daEliminare );
```

Analizza la stringa `str` in cerca delle ricorrenze della stringa `daEliminare` e, ovviamente le elimina una ad una.

```
size_t T1_elimina_spaziature_multiple_adiacenti( char *str,
                                                 char spaziatore );
```

Elimina “sul posto” i caratteri spaziatori consecutivamente multipli, facendo inoltre in modo che la stringa modificata non contenga mai un carattere spaziatore nella sua prima nè nella sua ultima posizione.

Restituisce la dimensione della stringa modificata. In caso d’errore, `((size_t)-1)`.

```
void T1_elimina_tra_stringhe( char *str,
                              const char *delim_1,
                              const char *delim_2,
                              int preserva_delim_1,
                              int preserva_delim_2 );
```

Analizza la stringa `str` in cerca di coppie consecutive dei delimitatori indicati da `delim_1` e `delim_2`, eliminando le porzioni di `str` contenute tra di essi. I delimitatori vengono a loro volta eliminati o preservati secondo quanto indicato dai parametri `preserva_delim_1` e `preserva_delim_2`.

```
char *T1_estrai_elemento( char *elementi,
                          char separatore, int indice );
```

Con la funzione `T1_trova_elemento()` individua l'elemento richiesto nell'ambito della stringa C `elementi` (che si suppone essere una lista di sottostringhe separate dal carattere `separatore`), quindi ne effettua una copia in uno spazio di memoria dinamica allocato con `malloc()`, eliminando da `elementi` quanto appena copiato.

Se la funzione ha successo restituisce il puntatore allo spazio di memoria dinamica nel quale è stato copiato l'elemento richiesto (spetta al chiamante liberare la memoria allocata con `free()`). Se fallisce restituisce `NULL`.

```
char *T1_inverti_stringa( char *s );
```

Riceve, tramite il parametro `s`, un puntatore ad una stringa C e la "rovescia" *in loco* invertendone l'ordine dei caratteri.

Restituisce il puntatore `s`.

```
int T1_parentesi_bilanciate( const char *s, char pa, char pc );
```

Verifica che nella stringa C `s` ad ogni parentesi aperta `pa` corrisponda una parentesi chiusa `pc`, e che non si ripresentino parentesi aperte prima che la parentesi aperta precedente sia stata chiusa.

Se le parentesi sono bilanciate restituisce `1`, in caso contrario restituisce `0`.

Esempi:

`"<prova>" -> 1`	`"<<prova><prova>>" -> 0`
`"<prova><prova>" -> 1`	`"<<prova>" -> 0`
`"<><><>" -> 1`	`"<prova>>" -> 0`
`"prova" -> 1`	`"<<prova>>" -> 0`
`"" -> 1`	

```
int T1_str_in_elementi_frazione( const char *str,
                                 double *n, double *d );
```

Riceve in `str` una stringa C che si suppone rappresenti una frazione e ne effettua internamente una copia nella quale sostituisce tutti i caratteri `','` con caratteri `'.'` (la memoria allocata viene comunque liberata prima che la funzione ritorni).

Applica `strtod()` alla copia di `str` per ricavarne il numeratore.

Salta gli eventuali spazi presenti dopo il numeratore.

Verifica che sia presente l'operatore `'/'`.

Applica `strtod()` a ciò che segue l'operatore `'/'` per ricavarne il denominatore.

Verifica che non esistano caratteri "spuri" dopo il denominatore,

Se numeratore e denominatore sono stati "estratti" correttamente, li colloca ripettivamente in `*n` e `*d`.

In caso di successo delle operazioni restituisce `1` (come per *true*).
In caso di insuccesso restituisce `0` (come per *false*).

La funzione fallisce se:

- `str` è `NULL` o `n` è `NULL` o `d` è `NULL`
- risulta impossibile effettuare la copia temporanea di `str`
- `strtod()` fallisce nel rilevare il numeratore o il denominatore
- l'operatore `'/'` è assente o non è collocato correttamente
- `str` contiene caratteri "spuri" estranei alla frazione da convertire

```
char *T1_str_binario( char *buff, uint64_t n, int qBit );
```

Colloca in `buff` (che deve poter contenere una stringa C di **ALMENO** `qBit` caratteri più uno) una stringa costituita da `'0'` e `'1'` che corrisponde al valore di `n` espresso in forma binaria.

NOTA: Il puntatore restituito è lo stesso passato come parametro `buff`. Se `qBit` non è compreso tra `1` e `64` nel buffer viene collocata una stringa C vuota (`""`).

```
int T1_strcat_dinamico( char **sd, size_t *lSd,
                        const char *sa, size_t lSa );
```

Una versione di `strcat()` che agisce su una stringa allocata dinamicamente.

Parametri

`char **sd`	Indirizzo del puntatore a uno spazio di memoria allocato dinamicamente con `malloc()`, `calloc()` o `realloc()` nel quale risiede la stringa in coda alla quale effettuare il concatenamento. Può essere `NULL`.
`size_t *lSd`	Indirizzo di un valore `size_t` nel quale può essere collocata la quantità di caratteri dopo la quale concatenare la stringa da aggiungere `sa`. Il puntatore `lSd` può essere `NULL`, così come il valore puntato può essere `((size_t)-1)` - in entrambi i casi, la funzione tenta di ricavare la lunghezza della stringa `sd` tramite `strlen()`, ma se `*sd` è `NULL`, presuppone per `*sd` una lunghezza `0`. Se il puntatore non è `NULL`, in uscita `*lSd` contiene la dimensione dello spazio di memoria nel quale risiede la stringa `*sd`.

`const char *sa`	Puntatore alla stringa da concatenare a `*sd`. Può essere `NULL`.
`size_t lSa`	Se è `((size_t)-1)` e `sa` non è `NULL`, la funzione impiega `strlen()` per ricavare autonomamente la lunghezza della stringa da concatenare.

```
char *T1_strdup_dinamico( const char *str, size_t lStr );
```

Questa funzione si occupa di creare in memoria dinamica un duplicato della stringa `str`. In uscita, `str` risulta invariata.

Parametri

`const char *str`	`str`: stringa; un puntatore a `const char` che punta ad una stringa C valida; dopo l'elaborazione, il contenuto della stringa rimane invariato
`size_t lStr`	`lStr`: lunghezza della stringa; se il valore di questo parametro è `((size_t)-1)` la funzione usa `strlen()` per ricavare la lunghezza di `str`, in caso contrario per allocare il *buffer* nel quale collocare la copia di `str` usa il valore passato in `lStr`

Restituisce il puntatore allo spazio di memoria allocato con `malloc()` nel quale è stato collocato il duplicato della stringa passata in `str`. Spetta al chiamante liberare con `free()` la memoria puntata.

```
int T1_stringa_da_valore( uint32_t valore,
                          const char *cifre, uint8_t base,
                          char *n_str );
```

Dato un valore intero positivo, lo converte in una stringa C che lo rappresenta impiegando le cifre contenute nell'*array* puntato dal secondo parametro.

La base di numerazione utilizzata per la conversione corrisponde alla lunghezza dell'*array* `cifre`. Le cifre possono essere costituite da caratteri qualsiasi.

NOTA: ogni cifra DEVE essere univoca (*case sensitive*) ma la funzione non effettua alcun controllo relativamente a questa condizione, dando per scontato che la validità delle cifre passate sia già stata accertata dal chiamante

Parametri

`uint32_t valore`	(input) il valore numerico intero positivo da convertire
`const char *cifre`	(input) puntatore a un *array* di `base` caratteri che rappresentano nell'ordine, l'ntero insieme delle cifre impiegate nel sistema di numerazione
`uint8_t base`	(input) la quantità dei caratteri contenuti nell'*array* `cifre`; il valore minimo ammesso è `2`
`const char *n_str`	(output) la stringa C che contiene la rappresentazione del valore intero positivo

Se va a buon fine restituisce la quantità delle cifre inserite nella stringa in uscita, altrimenti restituisce `0`.

La funzione può fallire se…

- uno qualsiasi dei parametri è `NULL`
- `n_str` o `cifre` sono stringhe vuote
- `base` è minore di `2`

```
int T1_trova_coppia_delimitatori( const char *s, /* input  */
     size_t ll,      /* input  */
     char d,         /* input  */
     size_t *d1,     /* output */
     size_t *d2 );   /* output */
```

Presupposti

Il parametro `s` punti a una stringa C valida.
Il parametro `ll` sia compreso entro i limiti della stringa `s` o sia `-1`.
(se `ll` è `-1`, viene usata l'intera stringa; se `ll` eccede i limiti della stringa, viene comunque usata solo la stringa, senza superarne i limiti).
I parametri `d1` e `d2` siano due puntatori validi.

Compito

Individuare, nell'ambito della stringa `s`, gli indici del primo carattere delimitatore e del carattere delimitatore immediatamente successivo.

Esito

Se entrambi i caratteri vengono individuati, la ricerca ha avuto successo.
Se non viene individuato alcun carattere o se ne viene individuato uno solo, la ricerca è fallita.
Se `s` è `NULL`, la ricerca è impossibile e quindi si considera fallita.
Se `d1`, `d2` o entrambi sono `NULL`, la comunicazione degli esiti della ricerca è impossibile, dunque la ricerca stessa si considera fallita.

Se la ricerca ha successo, viene restituito `1`, altrimenti viene restituito `0`.

Parametri

`const char *s`	(input) una stringa C
`size_t ll`	(input) la lunghezza oltre la quale non spingersi nell'analisi di `s`
`char d`	(input) il carattere delimitatore del quale trovare una coppia
`size_t *d1`	(output) l'indice del primo delimitatore a partire da `s`
`size_t *d2`	(output) l'indice del secondo delimitatore a partire da `s`

```
const char *T1_trova_elemento( const char *elementi,
                               char separatore, size_t indice,
                               size_t *lunghezza_elemento );
```

Analizza la stringa C `elementi`, che si suppone essere una lista di sottostringhe separate dal carattere indicato tramite il parametro `separatore`, in cerca dell'elemento indicato dal parametro `indice` (*zero based*).

Se l'elemento richiesto esiste (ovvero se indice è minore della quantità degli elementi presenti nella lista) la funzione ne restituisce il puntatore e colloca in `*lunghezza_elemento` la quantità dei caratteri che costituiscono la sottostringa dell'elemento trovato.

NOTA: `lunghezza_elemento` può essere `NULL`, nel qual caso la quantità dei caratteri non viene notificata al chiamante.

Se l'elemento richiesto non esiste la funzione restituisce `NULL` e `*lunghezza_elemento` non viene modificato.

```
int T1_valore_da_stringa( const char *n_str,
                          const char *cifre, uint8_t base,
                          uint32_t *valore );
```

Data una stringa che contiene la rappresentazione di un valore intero positivo ottenuta con le cifre contenute nell'*array* puntato dal secondo parametro, la converte nell'`uint32_t` corrispondente.

La base di numerazione utilizzata per la conversione corrisponde alla lunghezza dell'*array* `cifre`. Le cifre possono essere costituite da caratteri qualsiasi.

NOTA: ogni cifra DEVE essere univoca (*case sensitive*) ma la funzione non effettua alcun controllo relativamente a questa condizione, dando per scontato che la validità delle cifre passate sia già stata accertata dal chiamante

Parametri

`const char *n_str`	(input) la stringa C che contiene la rappresentazione del valore intero positivo

`const char *cifre`	(input) puntatore a un *array* di `base` caratteri che rappresentano nell'ordine, l'intero insieme delle cifre impiegate nel sistema di numerazione
`uint8_t base`	(input) la quantità dei caratteri contenuti nell'*array* `cifre`; il valore minimo ammesso è `2`
`uint32_t *valore`	(output) il valore numerico intero positivo ottenuto dalla conversione viene collocato in `*valore`

Valore di ritorno

Se ha successo restituisce `1` ("*true*"), altrimenti restituisce `0` ("*false*").

La funzione può fallire se…

- uno qualsiasi dei parametri è `NULL`
- `n_str` o cifre sono stringhe vuote
- in `n_str` sono presenti caratteri che mancano in `cifre`
- `base` è minore di `2`

```
int T1_caratteri_univoci_da_stringa( const char *sIn, char *sOut );
```

Analizza la stringa `sIn` e colloca in `sOut` l'elenco dei caratteri univoci che compaiono in essa, nello stesso ordine in cui si presentano la prima volta.

Esempio:
```
sIn  => "Cane non mangia cane"
sOut => "Cane omgic"
```

La funzione non fa nulla se...

- `sIn` è `NULL`
- `sOut` è `NULL`
- `sIn` e `sOut` sono entrambi `NULL`

Se tutto va bene restituisce `1` (*true*), altrimenti restituisce `0` (*false*).

NOTA: lo spazio di memoria puntato da `sOut` dev'essere almeno 256 *byte*; la capacità di `sOut` non viene viene verificata dalla funzione.

Funzioni correlate ai file

```
int T1_caratteri_univoci_da_file( FILE *fIn, char *sOut );
```

A partire dalla posizione corrente, analizza il testo contenuto nel file `fIn` e colloca in `sOut` l'elenco dei caratteri univoci che compaiono in esso, nello stesso ordine in cui si presentano la prima volta nel file stesso.

Esempio: `fIn  => "Cane non mangia cane"`
`sOut => "Cane omgic"`

In uscita, `fIn` è nella posizione in cui si trovava al momento della chiamata.
La funzione non fa nulla se `fIn`, `sOut` o entrambi sono `NULL`, o se non è possibile leggere da `fIn`.
Se tutto va bene restituisce `1` (*true*), altrimenti restituisce `0` (*false*).

NOTA: lo spazio di memoria puntato da `sOut` dev'essere almeno 256 *byte*; la capacità di `sOut` non viene viene verificata dalla funzione.

```
char *T1_carica_file_txt( const char *nome_file, long *l_txt );
```

Carica come stringa C il testo contenuto nel file `nome_file` in uno spazio di memoria dinamica allocato tramite `malloc()`/`realloc()` e ne restituisce il puntatore.

Se `l_txt` non è `NULL`, colloca in `*l_txt` la lunghezza della stringa C contenuta nello spazio di memoria allocato e puntato dal valore di ritorno.

Spetta al chiamante liberare la memoria dinamica eventualmente allocata.

In caso di errore restituisce `NULL` e non alloca alcuna memoria.

NOTA: la funzione fallisce se…

- non è possibile accedere al file tramite `fopen( "nome_file", "r" )`
- non è possibile allocare la memoria necessaria
- le dimensioni del file di testo da caricare superano quelle rilevabili tramite `ftell()`

```
const char **T1_file2array( size_t *itemsQuantity,
                            const char *fileName,
                            char itemsSeparator );
```

Converte un file di testo che contiene una lista di sottostringhe separate da caratteri separatori in un array di stringhe C site in un unico blocco di memoria.

Parametri

`size_t *itemsQuantity`	(**output**) La funzione collocherà in `*itemsQuantity` la quantità degli elementi nell'*array* in uscita. Se questo parametro è `NULL`, la funzione fallisce.
`const char *fileName`	(**input**) Il nome del file di testo da caricare come *array* di stringhe. Il file deve contenere una lista di sottostringhe separate da un carattere separatore singolo. Se questo parametro è una stringa vuota, la funzione fallisce.
`char itemsSeparator`	(**input**) Il carattere separatore usato per riconoscere le sottostringhe nel file.

Se la funzione porta a termine il suo compito restituisce un `char**` che punta all'*array* di stringhe ricavate dal file `fileName`. Dal momento che l'*array* risiede in uno spazio di memoria allocato dinamicamente, il chiamante **deve** liberare la memoria con `free()`.

Se la funzione fallisce restituisce `NULL`. La funzione fallisce se `itemsQuantity` è `NULL` o se il file `fileName` non è accessibile, o se l'allocazione della memoria non è possibile.

```
size_t T1_lunghezza_massima_riga_file( const char *nome_file );
```

Apre il file `nome_file` con requisiti di accesso `"r"`, quindi ne analizza il contenuto in cerca della quantità dei caratteri contenuti nella riga più lunga. Chiude il file senza modificarlo e restituisce il valore rilevato.

Funzioni relative all'allocazione di array

```
void *T1_crea_array_2d( size_t qr, size_t qc, size_t dim_el );
```

Alloca dinamicamente spazio in memoria per una matrice di `qr*qc` elementi di dimensioni `dim_el` *byte* ciascuno. L'area della memoria allocata riservata ai dati viene inizializzata azzerandola.

In caso di successo, restituisce un `void*` che deve essere assegnato ad un puntatore a doppia indirezione di tipo corrispondente al tipo di dati che si intende immagazzinare nella matrice appena creata.

La matrice può essere regolarmente impiegata con i classici operatori di indicizzazione (`matrice[riga][colonna]`).

La memoria allocata deve essere liberata dal chiamante tramite `free()`.

In caso di errore restituisce `NULL` e nessuna memoria risulta allocata.

```
void *T1_crea_array_3d( size_t w, size_t h, size_t d,
                        size_t dim_el );
```

Crea in un unico blocco di memoria dinamica un *array* a tre dimensioni di `w*h*d` celle. I dati veri e propri sono preceduti da una serie di puntatori atti ad assicurare che l'*array* possa essere usato con i regolari operatori `[]`.

Il valore di ritorno, un `void*`, fornisce il punto di accesso dell'*array* tridimensionale allocato, e va assegnato ad un puntatore a triplice indirezione del tipo desiderato. In caso di fallimento restituisce `NULL` (l'unica ragione di fallimento possibile è l'impossibilità da parte di `calloc()` d'allocare la memoria necessaria).

L'allocazione della memoria avviene per mezzo di `calloc()`.

È responsabilità del chiamante liberare con `free()` la memoria allocata.

Funzioni correlate alle date

```
int T1_anno_bisestile( int anno );
```

La funzione `T1_anno_bisestile()` restituisce `1` o `0`, a seconda se l'anno è o non è bisestile.

Funzioni correlate ai valori casuali

```
uint32_t T1_rand_32( void );
```

Estrae a sorte un numero pseudocasuale compreso tra `0x00000000` e `0xFFFFFFFF`.

```
double T1_rand_double( void );
```

Estrae a sorte un numero pseudocasuale compreso tra `0.0` e `1.0`.

Altre funzioni non-Win32

```
size_t T1_accoda_puntatori_nulli( void *array, size_t qEl );
```

Riceve tramite il parametro `array` un puntatore ad un *array* di puntatori di tipo qualsiasi che contiene la quantità di elementi indicata dal parametro `qEl`.

Compatta tutti i puntatori non `NULL` in testa all'*array*, spostando in coda tutti i puntatori `NULL`.

Occorre fare molta attenzione al dato da passare alla funzione tramite il parametro `array`, assicurandosi che sia effettivamente un *array* di puntatori.

In caso di successo restituisce la quantità dei puntatori non `NULL` presenti nell'*array*. In caso di insuccesso restituisce `SIZE_MAX`. L'unica causa di insuccesso segnalata è il passaggio di un parametro `array` nullo.

```
int T1_scambia_nomi_file( const char *f1, const char *f2 );
```

Scambia i nomi di due file identificati tramite i due parametri `f1` e `f2`, senza modificarne in alcun modo il contenuto.

Se lo scambio dei nomi va a buon fine restituisce `1`; in caso di errore, `0`.

Win32 — Funzioni correlate alla gestione degli appunti

```
BOOL T1_ImpostaAppuntiTxt( HWND hFin, const char *str, UINT lStr );
```

Immette negli appunti riferibili alla finestra `hFin` la stringa passata tramite il parametro `str`.

È possibile passare in `lStr` la quantità dei caratteri da immettere negli appunti. Se `lStr` non è `((UINT)-1)` la stringa puntata da `str` può anche non essere *zero terminated*. Se `lStr` è `((UINT)-1)` la funzione ricava autonomamente la lunghezza della stringa C (*zero terminated*) passata in `str`.

Se l'operazione va a buon fine, `ImpostaAppuntiTxt()` restituisce `TRUE`, altrimenti restituisce `FALSE`.

```
char *T1_RilevaAppuntiTxt( HWND hFin, UINT *lTesto );
```

Carica in memoria i dati in formato `TEXT` contenuti negli appunti riferibili alla finestra `hFin` (`hFin` può essere `NULL` se gli appunti non riguardano una particolare finestra).

Il valore restituito è il puntatore a uno spazio di memoria allocato tramite `malloc()` nel quale è contenuta la stringa C ricavata dagli appunti. Se non esistono appunti leggibili o se è impossibile allocare spazio in memoria, la funzione restituisce `NULL`.

In uscita, inoltre, se il parametro `lTesto` non è `NULL` `*lTesto` riporta la quantità di caratteri contenuta nella stringa restituita come valore di ritorno.

Win32 — Funzioni correlate alla gestione dei colori

```
COLORREF T1_ARGB2COLORREF( ARGB colore, ARGB sfondo );
```

Si comporta come `RemoveAlpha()`, ma restituisce sotto forma di `COLORREF` il colore risultante dalla miscelazione tra `colore` e `sfondo`.

NOTA: dà per scontato che `sfondo` sia un colore completamente opaco (l'eventuale trasparenza non viene neppure presa in considerazione).

```
ARGB T1_COLORREF2ARGB( COLORREF colore );
```

Riceve un colore espresso come `COLORREF` e lo restituisce in formato `ARGB`.

Dal momento che un `COLORREF` rappresenta sempre un colore del tutto opaco, il canale *alpha* del colore restituito come `ARGB` è sempre `0xFF`.

```
ARGB T1_ComponiARGB( BYTE a, BYTE r, BYTE g, BYTE b );
```

Riceve le quattro componenti *alpha*, *red*, *green* e *blue* di un colore e le restituisce assemblate sotto forma di valore `ARGB`.

```
COLORREF T1_ComponiCOLORREF( BYTE r, BYTE g, BYTE b );
```

Riceve le tre componenti *red*, *green* e *blue* di un colore e le restituisce assemblate sotto forma di valore `COLORREF`.

```
COLORREF T1_rby2rgb( double r, double b, double y );
```

Converte un colore espresso come rosso (`r`), blu (`b`) e e giallo (`y`) con valori compresi tra `0.0` e `1.0` per ciascuna componente nel corrispondente colore espresso come `COLORREF` (`0x00BBGGRR`).

```
ARGB T1_RemoveAlpha( ARGB colore, ARGB sfondo );
```

Dati un `colore` di primo piano e uno di `sfondo`, provvede a miscelarli in modo che il valore di *alpha* venga “incorporato” negli altri canali.

Il colore derivante dalla miscela viene restituito come valore di ritorno.

Il colore restituito come valore di ritorno ha sempre il canale *alpha* impostato su `0xFF` (totalmente opaco).

NOTA: dà per scontato che lo sfondo sia un colore completamente opaco (l’eventuale trasparenza non viene neppure presa in considerazione).

```
COLORREF T1_SchiarisciColore( COLORREF colore,
                              BYTE variazione, BOOL schiarisci );
```

Applica uno schiarimento di entità definita dal parametro `variazione` al colore indicato dal parametro `colore`. Restituisce il colore risultante come valore di ritorno.

Se il parametro schiarisci è `TRUE`, il colore viene schiarito. Il cambiamento avviene come se il colore, con canale alfa impostato su `(0xFF-variazione)`, venisse sovrapposto a uno sfondo bianco.

Se il parametro `schiarisci` è `FALSE`, il colore viene scurito. In questo caso, il cambiamento avviene come se il colore, con canale alfa `(0xFF-variazione)`, venisse sovrapposto a uno sfondo nero.

```
void T1_ScomponiARGB( ARGB colore,
                      BYTE *a, BYTE *r, BYTE *g, BYTE *b );
```

Riceve un `colore` espresso come `ARGB` e ne restituisce le componenti in `*a`, `*r`, `*g` e `*b`.

```
void T1_ScomponiCOLORREF( COLORREF colore,
                          BYTE *a, BYTE *r, BYTE *g, BYTE *b );
```

Riceve un colore espresso come `COLORREF` e ne restituisce le componenti.
Da ricordare: in un `COLORREF` la componente `a` (*alpha*) non è significativa.

Win32 — *Gestione di coordinate e RECT*

```
BOOL T1_ClientPoint2ScreenPoint( HWND hwnd, CONST LPPOINT cPt,
                                 LPPOINT sPt );
```

Inserisce in `*sPt` le coordinate a schermo corrispondenti alle coordinate locali `*cPt`. I puntatori `cPt` e `sPt` possono puntare alla stessa struttura `POINT`.

In caso di successo viene restituito `TRUE`, mentre in caso di fallimento viene restituito `FALSE`.

La funzione fallisce e restituisce `FALSE` in tre situazioni:

- se anche uno solo dei parametri è `NULL`;
- se `hwnd` non identifica una finestra valida di *Win32*;
- se la funzione di *Win32* `ClientToScreen()` fallisce a sua volta.

```
BOOL T1_ClientRect2ScreenRect( HWND hwnd, LPCRECT cRect,
                               LPRECT sRect );
```

Inserisce in `*cRect` la versione locale del `RECT` riferito allo schermo `*sRect`.

I puntatori `sRect` e `cRect` possono puntare alla stessa struttura `RECT`.

In caso di successo viene restituito `TRUE`, mentre in caso di fallimento viene restituito `FALSE`.

La funzione fallisce e restituisce `FALSE` in tre situazioni:

- se anche uno solo dei parametri è `NULL`;
- se `hwnd` non identifica una finestra valida di *Win32*;
- se la funzione di *Win32* `ClientToScreen()` fallisce a sua volta.

```
RECT **T1_CreaMatriceRECT( const RECT *limiti,
                           size_t qr, size_t qc );
```

Divide l'area delimitata dal parametro `limiti` in `qr` righe per `qc` colonne, quindi crea `qr*qc` elementi di tipo `RECT` e li imposta in modo che definiscano una griglia regolare entro il `RECT *limiti`.

Restituisce il puntatore al punto di accesso della matrice, ovvero allo spazio di memoria allocato con `calloc()` nel quale la matrice è contenuta.

Spetta al chiamante deallocare con `free()` la memoria puntata dal valore di ritorno.

```
BOOL T1_MassimizzaRectInRect( const RECT *rEst,
                              const RECT *rInt,
                              RECT *rMax );
```

Riceve gli indirizzi di tre `RECT`.

`const RECT *rEst`	un `RECT` di riferimento entro il quale sarà centrato, scalandolo in modo che sia il più grande possibile, l'altro `RECT` puntato da `rInterno`
`const RECT *rInt`	il `RECT` da centrare, scalato in modo che sia il più grande possibile, entro l'altro `RECT` puntato da `rEsterno`; l'indirizzo può coincidere con quello di `rMassimizzato`
`RECT *rMax`	l'indirizzo a un `RECT` che riceverà la versione centrata e scalata del `RECT` puntato da `rInterno`; questo indirizzo può coincidere con quello di `rInterno`, nel qual caso il `RECT` puntato da `rInterno` sarà modificato anche se `rInterno` è un parametro `const`

Restituisce `TRUE` se la funzione è andata a buon fine, `FALSE` se si sono verificati errori. Fallisce solo se viene passato anche un solo puntatore `NULL`.

```
BOOL T1_RilevaAreaUtileSchermo( LPRECT area_utile,
                                LPLONG wMax, LPLONG hMax );
```

L'area utile dello schermo è quella che nella documentazione di *Win32* viene denominata "*work area*" e consiste nella porzione di schermo potenzialmente a disposizione di un programma. Ad esempio, l'area utile può essere più o meno estesa a seconda della condizione di visibilità della barra del menu *Start* di *Windows*.

Questa funzione riporta all'indirizzo segnalato tramite il parametro corrispondente i dati rilevati. Alcuni parametri possono essere `NULL`, a condizione che non lo siano tutti. Se un parametro è `NULL`, il dato corrispondente non viene segnalato al chiamante.

In caso di successo viene restituito `TRUE`, mentre in caso di fallimento viene restituito `FALSE`.

La funzione fallisce e restituisce `FALSE` in due situazioni:

- se TUTTI i parametri sono `NULL`;
- se la funzione di *Win32* `SystemParametersInfo()` fallisce a sua volta.

```
BOOL T1_RiquadroBordato( HDC hdc, const RECT *area,
                         COLORREF colore, INT16 spessore_bordo,
                         CHAR direzione_rilievo,
                         BYTE entita_rilievo);
```

Entro i limiti imposti dal parametro `r`, traccia nel *Device Context* rappresentato dal parametro `hdc` un riquadro bordato che può essere piano, incassato o in rilievo.

Il colore dello sfondo del riquadro è dato ovviamente dal parametro `colore`.

Lo spessore del bordo del riquadro è dato ovviamente dal parametro `spessore_bordo`. Lo spessore del bordo viene inteso come valore assoluto, per cui un valore di `spessore_bordo` negativo viene reso comunque positivo internamente alla funzione. Se `spessore_bordo` è `0` (zero), il bordo non viene tracciato.

Il parametro `direzione_rilievo` determina la “sporgenza” del bordo:

- se è positivo il riquadro è in rilievo
- se è negativo il riquadro è incassato
- se è `0` (zero), il riquadro ha un bordo piano

Con `direzione_rilievo` uguale a zero, il bordo è scurito se `spessore_bordo` è negativo e schiarito se `spessore_bordo` è positivo.

Il livello di schiarimento/scurimento dei bordi è determinato dal parametro `entita_rilievo`. Il valore di `entita_rilievo` può variare da `0x00` a `0xFF`.

```
BOOL T1_ScreenPoint2ClientPoint( HWND hwnd,
                                 CONST LPPOINT sPt, LPPOINT cPt );
```

Inserisce in `*cPt` le coordinate locali corrispondenti alle coordinate a schermo `*sPt`. I puntatori `sPt` e `cPt` possono puntare alla stessa struttura `POINT`.

In caso di successo viene restituito `TRUE`, mentre in caso di fallimento viene restituito `FALSE`.

La funzione fallisce e restituisce `FALSE` in tre situazioni:

- se anche uno solo dei parametri è `NULL`;
- se `hwnd` non identifica una finestra valida di *Win32*;
- se la funzione di *Win32* `ClientToScreen()` fallisce a sua volta.

```
BOOL T1_ScreenRect2ClientRect( HWND hwnd,
                               LPCRECT sRect, LPRECT cRect );
```

Inserisce in `*sRect` la versione riferita allo schermo del `RECT` locale `*cRect`.

I puntatori `cRect` e `sRect` possono puntare alla stessa struttura `RECT`.

In caso di successo viene restituito `TRUE`, mentre in caso di fallimento viene restituito `FALSE`.

La funzione fallisce e restituisce `FALSE` in tre situazioni:

- se anche uno solo dei parametri è `NULL`;
- se `hwnd` non identifica una finestra valida di *Win32*;
- se la funzione di *Win32* `ScreenToClient()` fallisce a sua volta.

Win32 — Funzioni riferite alla gestione delle stringhe

```
BOOL T1_CambiaCapitalizzazione( LPSTR t, DWORD lt, INT dir );
```

Converte in maiuscolo o in minuscolo `lt` caratteri della stringa C `t`, secondo il valore del parametro `dir` (`dir>0`, maiucolo; `dir<0`, minuscolo; `dir==0`, niente).

Se lt è `((DWORD)-1)`, viene convertita l'intera stringa C `t`.

Se la funzione va a buon fine, restituisce `TRUE`, altrimenti restituisce `FALSE`.

La funzione fallisce e restituisce `FALSE` in due situazioni:

- se `t` è `NULL`
- se `CharUpperBuffA()` o `CharLowerBuffA()` fallisce a sua volta.

```
WCHAR *T1_CStr2WStr( LPCSTR cStr );
```

Crea in un'area di memoria dinamica allocata con `malloc()` una versione `WCHAR` della stringa C di `char cStr`.

Se la funzione va a buon fine, restituisce il puntatore alla memoria allocata, che contiene la versione `WCHAR` di `cStr`, altrimenti restituisce `NULL`. Spetta al chiamante deallocare con `free()` la memoria puntata dal valore di ritorno.

La funzione fallisce restituendo `NULL` solo `cStr` è `NULL` o se `malloc()` fallisce a sua volta.

```
BOOL T1_RilevaDimensioniCarattere( HDC hdc, CHAR c, LPSIZE dim );
```

Rileva le dimensioni del carattere `c` per il *font* correntemente selezionato nel *device context* `hdc` e le inserisce in `*dim`.

Se va tutto bene, restituisce `TRUE`. Restituisce `FALSE` solo in due situazioni:

- se `dim` è `NULL`
- se la funzione di *Win32* `GetTextExtentPoint32()` fallisce a sua volta.

```
BOOL T1_RilevaDimensioniTesto( HDC hdc, LPCSTR t, LPSIZE dim );
```

Rileva le dimensioni della stringa `t` per il *font* correntemente selezionato nel *device context* `hdc` e le inserisce in `*dim`.

Se va tutto bene, restituisce `TRUE`. Restituisce `FALSE` solo in due situazioni:

- se `dim` è `NULL` o `t` è `NULL`
- se la funzione di *Win32* `GetTextExtentPoint32()` fallisce a sua volta.

```
void T1_TracciaAreaTesto( HDC hdc,
                          const T1_AreaTesto_struct
                          *aTxt );
```

Usando le impostazioni correnti del *device context* `hdc`, traccia il testo secondo quanto definito nella struttura `aTxt` (la struttura `T1_AreaTesto_struct` è definita in testa a questo file).

Se `hdc` è `NULL` o `aTxt` è `NULL`, la funzione non fa nulla.

Win32 — Altre funzioni

```
BOOL T1_RilevaDimensioniBitmap( HBITMAP hBmp, SIZE *dim );
```

Permette di rilevare le dimensioni di un'immagine `BITMAP`. Se tutto funziona a dovere, restituisce `TRUE`, altrimenti restituisce `FALSE`. Se le cose vanno a buon fine, le dimensioni della *bitmap* sono immagazzinate in `*dim`; se le cose vanno storte `*dim` rimane invariato.

```
BOOL T1_RilevaBordiFinestra( HWND hwnd, LPRECT bordi );
```

Confrontando il *window RECT* di una finestra e il suo *client RECT*, ricava lo "spessore" dei bordi, collocandoli nei quattro campi di `*bordi`.

In caso di successo viene restituito `TRUE`, in caso di fallimento viene restituito `FALSE`.

La funzione fallisce e restituisce `FALSE` in cinque situazioni:

- se `hwnd` non identifica una finestra valida di *Win32*;
- se `bordi` è `NULL`;
- se la funzione di *Win32* `GetWindowRect()` fallisce;
- se la funzione di *Win32* `GetClientRect()` fallisce;
- se la funzione di *Win32* `ClientToScreen()` fallisce.