Note

È considerato errore qualsiasi output non richiesto dagli esercizi.
È importante scrivere il proprio main in Visual Studio per poter fare correttamente il debug delle funzioni realizzate!

Esercizio 1 (punti 5)

Nel file `formato.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern bool is_date(const char *s);
```

La funzione accetta una stringa zero terminata e deve verificare se rispetta il formato di una data in cui il giorno e il mese sono rappresentati da due cifre, l'anno da quattro cifre e sono separati dal carattere "/". La funzione ritorna 1 se il formato è corretto, 0 se il puntatore è NULL, o se la stringa non rispetta il formato indicato.
Ad esempio la stringa "10/07/2015" rispetta il formato, mentre "10-7-15" o "10.07.2015" no.

Esercizio 2 (punti 6)

Nel file `trigonometria.c` implementare in linguaggio C la funzione corrispondente alla seguente dichiarazione:

```
extern double seno_iperbolico(double x);
```

La funzione deve calcolare il valore di $\sinh(x)$ utilizzando il seguente sviluppo in serie di Taylor:

$$\sinh(x) = \sum_{n=0}^{\infty} \frac{1}{(2n+1)!} x^{2n+1}$$

Il ciclo (teoricamente infinito) deve continuare finché il risultato non cambia più.
Nel realizzare la funzione non è consentito l'uso di librerie esterne.

Esercizio 3 (punti 7)

Nel file `stringhe.c` implementare la definizione della funzione:

```
extern char *parola_piu_lunga(const char *sz);
```

La funzione accetta come parametro una stringa C (zero terminata) e deve restituire un puntatore ad una nuova stringa zero terminata (allocata dinamicamente nell'heap) contenente una copia della parola più lunga presente all'interno della stringa `sz`. Con "parola" si intende qualsiasi sequenza di caratteri che non contenga whitespace (spazi, tabulazioni o a capo). Se ci sono più parole della lunghezza massima, la funzione ritorna la prima incontrata.

Esercizio 4 (7 punti)

Creare i file `rational.h` e `rational.c` che consentano di utilizzare la seguente struttura che consente di rappresentare frazioni:

```
struct rational {
	int num;
	unsigned int den;
};
```

e la funzione:

```
extern void rational_sum(struct rational *sum, const struct rational *first,
                         const struct rational *second);
```

La funzione deve essere in grado di eseguire la somma tra frazioni e di restituire il risultato ridotto ai minimi termini.
La funzione accetta tre parametri, un puntatore a una `struct rational` già allocata che dovrà contenere il risultato una volta eseguita l'operazione, un puntatore al primo addendo e un puntatore al secondo addendo.
Dopo aver calcolato il risultato come segue:

$$\frac{num_1}{den_1} + \frac{num_2}{den_2} = \frac{num_1 \cdot den_2 + den_1 \cdot num_2}{den_1 \cdot den_2}$$

il risultato deve essere ridotto ai minimi termini, ovvero numeratore e denominatore non devono avere divisori comuni oltre all'unità. Per ottenere una frazione ai minimi termini, si devono dividere numeratore e denominatore per il loro massimo comune divisore.
Per convenzione uno zero viene indicato come una frazione con `num=0` e `den=1`. I puntatori passati saranno sempre validi e già allocati.

Esercizio 5 (punti 8)

Creare i file `matrix.h` e `matrix.c` che consentano di utilizzare la seguente struttura:

```
struct matrix {
	size_t rows, cols;
	double *data;
};
```

e la funzione:

```
extern bool matrix_read(struct matrix *m, FILE *f);
```

La struct consente di rappresentare matrici di dimensioni arbitraria, dove `rows` è il numero di righe, `cols` è il numero di colonne e `data` è un puntatore a `rows×cols` valori di tipo `double` memorizzati per righe. Consideriamo ad esempio la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

questo corrisponderebbe ad una variabile `struct matrix A`, con `A.rows = 2`, `A.cols = 3` e `A.data` che punta ad un area di memoria contenente i valori `{1.0, 2.0, 3.0, 4.0, 5.0, 6.0 }`.

La funzione deve leggere la matrice `m` dal file `f` (già aperto). La struct è già allocata, ma non lo sono i suoi dati che dovranno essere allocati dopo aver capito quante righe e colonne ha la matrice sul file. Questa è memorizzata in formato testuale decimale con il valore di `rows` seguito da <a capo>, il valore di `cols` seguito da <a capo>, i valori della matrice separati da <tabulazione> (\t in linguaggio C) all'interno della riga e con un <a capo> alla fine di ogni riga (compresa l'ultima).

Facendo riferimento all'esempio precedente, A sarebbe scritta sul file come

```
2↵
3↵
1.000000	→	2.000000	→	3.000000↵
4.000000	→	5.000000	→	6.000000↵
```

Visualizzando in una tabella i singoli caratteri si vedrebbe

2	\n																							
3	\n																							
1	.	0	0	0	0	0	0	\t	2	.	0	0	0	0	0	0	\t	3	.	0	0	0	0	0	0	\n
4	.	0	0	0	0	0	0	\t	5	.	0	0	0	0	0	0	\t	6	.	0	0	0	0	0	0	\n

La funzione ritorna `true` se la lettura è andata a buon fine, `false` se per qualche motivo è fallita.