Compito 05/02/2020 (1)

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 (5 punti)

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

```
struct angolo {
    uint16_t gradi;
    uint8_t primi;
    uint8_t secondi;
};
```

e la funzione:

```
extern struct angolo somma_angoli(struct angolo a, struct angolo b);
```

La `struct angolo` consente di rappresentare un angolo in termini di gradi, primi e secondi.

- Un `grado` è la 360-esima parte di un angolo giro (simbolo °).
- Un `primo` è la 60-esima parte di un grado (simbolo ').
- Un `secondo` è la 60-esima parte di un primo (simbolo '').

In altre parole, 60 secondi formano un primo, 60 primi formano un grado e 360 gradi formano un angolo giro.

Ad esempio, la struct:

```
{ 230, 47, 12 }
```

Rappresenta l'angolo 230° 47' 12''.

La misura di un angolo è detta in **forma normale** se i primi e i secondi sono minori di 60.

La funzione `somma_angoli` accetta come parametri due angoli e ne restituisce la somma, espressa in forma normale. pAd esempio, invocando la funzione con:

```
a = { 230, 47, 12 }
b = { 45, 23, 4 }
```

Essa deve restituire:

```
{ 276, 10, 16 }
```

Esercizio 2 (6 punti)

Nel gioco di ruolo "La leggenda dei cinque anelli", per superare una prova è necessario sommare il valore di abilità a quello di un tratto del personaggio, per ottenere il numero di dadi a 10 facce da tirare. Una volta tirati i dadi, si tiene un numero di dadi pari al punteggio del tratto e se ne sommano i valori. Questo sistema è noto come "Roll and Keep", in breve XkY, ovvero tira X dadi e tieni i migliori Y.

Nel file `15r.c`, inserire la definizione della seguente funzione:

```
extern int roll_and_keep(const int *r, size_t x, size_t y);
```

che dato un vettore `r` contenente i valori di `x` dadi, restituisce la somma degli `y` migliori. Se `r` è `NULL`, se `x=0` o `y=0` o `y>x`, la funzione ritorna `0`.

Ad esempio, con

```
r = { 3, 8, 2, 7, 1, 9 }
x = 6
y = 3
```

la funzione deve restituire 24, perché gli y (3) valori più grandi sono 8, 7 e 9, e `8 + 7 + 9 = 24`.

Esercizio 3 (7 punti)

Creare i file `rectangle.h` e `rectangle.c` che consentano di utilizzare le seguenti strutture:

```
struct point {
    int32_t x, y;
};
struct rect {
    struct point a, b;
};
```

e la funzione:

```
extern bool rect_load(FILE *f, struct rect *r);
```

La struct `point` consente di rappresentare un punto secondo le sue coordinate cartesiane `x` e `y`.

La struct `rect` consente di rappresentare un rettangolo identificato da una coppia di punti, dove `a` e `b` sono vertici opposti del poligono.

Consideriamo ad esempio il rettangolo:

$$P1(0;2) \quad P2(3;2) \quad P3(3;0) \quad P4(0;0)$$

```
    P1--------P2
    |         |
    |         |
    P4--------P3
```

Questo può essere rappresentato con una variabile `struct rect R` con

```
R.a = P1    oppure    R.a = P2    oppure    R.a = P3    oppure    R.a = P4
R.b = P3              R.b = P4              R.b = P1              R.b = P2
```

La funzione riceve in input un puntatore ad un file già aperto in modalità lettura non tradotta (binario) e legge un solo rettangolo memorizzato su file con 4 valori a 32 bit in little endian, nell'ordine a.x, a.y, b.x, b.y. I dati letti vanno inseriti nella `struct rect` puntata da `r`. Il puntatore passato sarà sempre valido e già allocato. Al termine della lettura la posizione nel file sarà avanzata di `4 * 4 = 16` byte e pronta per leggere eventuali dati successivi. Il file sarà sempre formattato correttamente.

La funzione ritorna `true` se riesce a leggere correttamente i 4 interi, `false` altrimenti.

Ad esempio il file seguente (visto come in un editor esadecimale):

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  00 00 00 00 01 00 00 00 04 00 00 00 FF FF FF FF  ............ÿÿÿÿ
00000010  03 00 00 00 05 00 00 00 FD FF FF FF 0A 00 00 00  ........ýÿÿÿ....
```

Contiene due rettangoli:

```
{ {0, 1}, {4, -1} } e { {3, 5}, {-3, 10} }
```

Notate che chiamando la funzione ne viene letto solo uno.

Esercizio 4 (7 punti)

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 struct matrix *mat_replicate(const struct matrix *m);
```

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 accetta come parametro un puntatore ad una matrice `m`, e deve restituire un puntatore a una nuova matrice allocata dinamicamente. La nuova matrice è ottenuta copiando i dati di quella di input e poi replicandoli in orizzontale, raddoppiando quindi il numero di colonne. Se `m` è `NULL`, la funzione restituisce `NULL`.

Ad esempio, data la matrice:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

La funzione restituisce la nuova matrice:

$$\begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 4 & 5 & 6 & 4 & 5 & 6 \\ 7 & 8 & 9 & 7 & 8 & 9 \end{pmatrix}$$

Esercizio 5 (8 punti)

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

```
extern int prodotto_altri_due(const int *v, size_t n);
```

La funzione accetta un vettore di `n` numeri di tipo `int` e restituisce il numero di elementi che sono uguali al prodotto di altri due.

Ad esempio, dato il vettore:

```
v = { 2, -3, -6, 7, 14 }
```

La funzione deve ritornare `2`.

Se `v` è `NULL` o `n=0`, la funzione restituisce `-1`.