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 `terna.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern bool is_terna_pitagorica(unsigned int v[3]);
```

La funzione accetta un puntatore a 3 numeri interi positivi o nulli e deve verificare se questi formano una terna pitagorica, ovvero se la somma dei quadrati di due di questi è pari al quadrato del numero rimanente.
Ad esempio la terna 5,4,3 è una terna pitagorica, mentre 1,2,3 no.

Esercizio 2 (punti 6)

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

```
extern bool cifre_duplicate_hex(unsigned int n);
```

La funzione prende come input un numero intero senza segno (`unsigned int`) e verifica se nella sua rappresentazione in base 16 una cifra esadecimale compare più di una volta. Ad esempio il numero 12F non ha cifre duplicate (la funzione deve ritornare `false`), perché 1 compare una sola volta, 2 compare una sola volta e F compare una sola volta. Invece A23A ha cifre duplicate (la funzione deve ritornare `true`), perché A compare due volte.

Esercizio 3 (punti 7)

Creare i file `stringhe.h` e `stringhe.c` che consentano di utilizzare la seguente funzione:

```
extern char *zfill(const char *str, size_t n);
```

La funzione riceve in input un puntatore a una stringa C `str` e una dimensione `n` e in uscita deve produrre una nuova stringa C, di almeno `n` caratteri, allocata dinamicamente su heap, che contenga la stringa `str` con davanti tanti caratteri '0'. Se str è già lunga `n` o più lunga, non deve aggiungere nulla.
Ad esempio invocando la funzione così: `zfill("ciao", 10)` otterremmo in output `"000000ciao"`, mentre invocando la funzione così: `zfill("ciao", 2)` otterremmo in output `"ciao"`
La funzione deve gestire correttamente qualsiasi carattere ASCII e se il puntatore è NULL, deve ritornare NULL.

Esercizio 4 (7 punti)

Creare i file `shapes.h` e `shapes.c` che consentano di utilizzare la seguente struct in grado di rappresentare rettangoli associati ad un simbolo:

```
struct rect {
    int16_t x, y;
    uint16_t width, height;
    char symbol;
};
```

e la funzione:

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

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 16 bit in little endian e un carattere ASCII. 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 9 byte e pronta per leggere i dati successivi.

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 0A 00 05 00 2B 05 00 05 00 14 00 0F  ........+.......
00000010  00 2A                                           .*
```

contiene due rettangoli [x=0, y=0, width=10, height=5, symbol='+'] e [x=5, y=5, width=20, height=15, symbol='*'].

Esercizio 5 (punti 8)

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

```
struct forest {
	size_t rows, cols;
	char *data;
};
```

e la funzione:

```
extern void propagate_fire(const struct forest *f);
```

Questa struct viene utilizzata in un simulatore di incendi boschivi e rappresenta un'area quadrata piena di alberi disposti secondo una griglia regolare di `rows` righe e `cols` colonne, con `data` un puntatore a `rows×cols` valori di tipo `char` memorizzati per righe. Ogni elemento è un albero se il carattere è un '.' o un incendio se è una 'F'.

La funzione deve simulare il propagarsi dell'incendio dopo una unità di tempo, durante il quale ogni casella contenente un incendio si propaga a nord, sud, est e ovest di una posizione. Attenzione ai bordi della matrice, perché non bisogna propagare l'incendio fuori dalla memoria puntata dal puntatore f.
Il puntatore f punta a dati const, perché non è consentito modificare il puntatore data, dato che non è detto che sia allocato su heap, ma si può modificare solo il suo contenuto. Usate una foresta di appoggio per memorizzare il vecchio o il nuovo stato della foresta, riportando poi su `f` il risultato.

Ad esempio, data la foresta

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . F F . .
. . . F F F . .
. . . . F . . .
. . . . . .
. . . . . .
. . . . . .
```

dopo la propagazione si otterrà

```
. . . . . .
. . . . . .
. . . . . .
. . . . F F . .
. . . F F F F . .
. . F F F F F . .
. . . F F F . . .
. . . . F . . .
. . . . . .
. . . . . .
```