LEGGETE LA GUIDA PER LA CREAZIONE DEI PROGETTI E PER IL DEBUGGING!

Gli esercizi seguenti devono essere risolti, compilati e testati utilizzando il debugger. Per ognuno si deve realizzare una funzione main() che ne testi il funzionamento. ***Fate progetti diversi per ogni esercizio.***

Esercizio 1

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

```
extern char *concatena(const char *prima, const char *seconda);
```

La funzione riceve due puntatori a stringhe di caratteri zero terminate e alloca dinamicamente sull'heap sufficiente memoria per contenerle entrambe (incluso un terminatore) e copia nel nuovo spazio allocato la prima, seguita dalla seconda. Un esempio di chiamata è il seguente:

```
int main(void) {
	char s1[] = "Questa e' la ";
	char s2[] = "stringa risultante.";
	char *s;

	s = concatena(s1, s2);

	free(s);
}
```

In questo caso s punterà ad un vettore di caratteri contenente "Questa e' la stringa risultante.". Se uno dei puntatori (`prima` o `seconda`) è NULL o punta ad una stringa vuota (cioè il primo carattere vale 0), la funzione lo tratta come una stringa di lunghezza 0. Ad esempio chiamando `concatena("a","")`, si allocano 2 char e la stringa ritornata contiene il carattere `'a'` e il carattere 0.

Esercizio 2

Sia data la struct seguente:

```
struct punto {
	double x, y;
};
```

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

```
extern int colineari(struct punto p1, struct punto p2, struct punto p3);
```

La funzione ritorna 1 se p_1, p_2 e p_3 giacciono sulla stessa retta, altrimenti 0. L'equazione da verificare è la seguente:

$$(x_3 - x_2)(y_1 - y_2) = (y_3 - y_2)(x_1 - x_2)$$

Esercizio 3

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

```
struct complesso {
	double re,im;
};
```

e la funzione:

```
extern void prodotto_complesso (struct complesso *comp1, const struct complesso *comp2);
```

La funzione `prodotto_complesso` esegue il prodotto dei due valori `comp1` e `comp2` e mette il risultato in `comp1`. Si ricorda che il prodotto di numeri complessi si esegue così:

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc)$$

Esercizio 4

Creare il file `encrypt.c` che contenga la definizione della seguente funzione:

```
extern void encrypt(char *s, unsigned int n);
```

La funzione accetta una sequenza `s` di `n` char e la codifica sostituendo ad ogni char il suo valore trasformato con uno XOR bit a bit con il valore esadecimale AA. Per le proprietà dello XOR, l'operazione è invertibile, quindi, riapplicando la funzione sulla sequenza codificata, si riottiene quella originale. Non è richiesto, né è garantito che la sequenza sia 0 terminata. Per questo motivo viene passato un parametro apposito.

Esercizio 5

Creare il file `conversione.c` in cui deve essere definita la funzione corrispondente alla seguente dichiarazione:

```
extern char *converti(unsigned int n);
```

La funzione accetta come parametro un numero naturale o nullo `n` e deve restituire un puntatore ad un array di caratteri zero terminato allocato dinamicamente, contenente la rappresentazione in base 10 del numero intero, con le singole cifre rappresentate in ASCII. Ad esempio:
il numero 4355 genererà l'array di caratteri: "4355", ovvero { 52, 51, 53, 53, 0 }, o anche { 0x34, 0x33, 0x35, 0x35, 0x00 }.

Provate a fare questo esercizio prima senza e poi con la funzione `sprintf()`.

In questo esercizio non create un file `conversione.h`, e nel file `main.c` mettete la funzione `main` che utilizza la funzione `converti`. Il `main` non deve generare warning.

Esercizio 6

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

```
extern size_t conta_parole (const char *s);
```

La funzione accetta come parametro una stringa C e deve restituire in un dato di tipo `size_t` quante parole sono presenti all'interno della stringa, dove con "parola" intendiamo una sequenza di caratteri diversi da spazio. Ad esempio, colla stringa `"  Questa e' una stringa lunga 45 caratteri.  "` dovrebbe ritornare 7. Colla stringa `"1 2 3 a b c"` dovrebbe ritornare 6.

In questo esercizio non create un file `conta.h`, e nel file `main.c` mettete la funzione `main` che utilizza la funzione `conta_parole`. Il `main` non deve generare warning.

Esercizio 7

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

```
extern unsigned short bin2bcd(unsigned short val);
```

La funzione accetta come parametro un numero intero non negativo minore di 10000 e lo ritorna codificato in Binary Coded Decimal (BCD).

La codifica BCD è un modo comunemente utilizzato in informatica ed elettronica per rappresentare le cifre decimali in codice binario. In questo formato ogni cifra di un numero è rappresentata da un codice binario di quattro bit, il cui valore è compreso tra 0 (0000) e 9 (1001). Le restanti sei combinazioni non sono utilizzate. Per esempio il numero 127 è rappresentato in BCD come la sequenza di bit 0001, 0010, 0111.

Questa funzione impacchetta le 4 cifre di val dalla più significativa alla meno significativa in 16 bit (senza segno). Nel caso di 127 quindi produrrebbe 0000.0001.0010.0111, o in esadecimale (alla C) 0x0127. Cioè ogni cifra in base 10 viene convertita nella corrispondente cifra in base 16.
Il numero 0 diventerebbe 0x0000, il numero 9999 diventerebbe 0x9999 e così via.