Prova scritta Programmazione I - 12 Settembre 2019

Nome e Cognome: _____ Matricola: _____

1. [3 punti] Elencare tutte le conversioni di tipo. Qual è il valore di b alla fine della funzione?

```
1 #define A 3.6
2 short int f(long long  p1, float
      p2){
3   return (p1 >= p2 ? p1 : p2);
4 }
5
6 int main(void) {
7   int a = A;
8   int b = 2UL;
9   b %= f(b, a);
10 }
11
```

linea 7: a da double a int
linea 8: 2UL da unsigned long int a int
linea 9: b da int a long long e a da int a float
linea 3: p1 da long long a float
linea 3: ?: ritorna il più alto tra long long e float, quindi ritorna float
quindi p1 convertito a float
linea 3: p1 convertito a float viene convertito a short (ritorno)
linea 9: b = b % 3, 3 da short a int

2 % 3 == 2
il valore finale di b è 2

2. [4 punti] Definire una funzione *rotate90()* che prenda in input una matrice di dimensione $m \times n$, e ne crei una $n \times m$ che corrisponde ad una rotazione in senso antiorario di 90 gradi, come in esempio. La funzione stampa anche la matrice risultato, ma non la ritorna al chiamante.

3. [4 punti] Definire una funzione che prende un puntatore ad una lista di elementi come specificato sotto e crea una nuova lista, totalmente nuova, copia di quella passata. La lista viene poi ritornata dalla funzione. Controllare se la lista iniziale è vuota.

```
1 struct Node {
2   int info;
3   struct Node* pNext;
4 }
```

4. [5 punti] Si definisca una funzione che prende in input due puntatori a liste di elementi come definito nell'esercizio precedente. Scrivere una funzione *alternate* che restituisca una nuova lista i cui elementi siano presi dalle due precedenti e disposti in maniera alternata, seguendo la seguente sequenza: *elem1lista1, elem1lista2, elem2lista1, elem2lista2,..., elemNlista1, elemNlista2.* Supporre che le liste abbiano la stessa lunghezza: controllare che non sia zero (due liste vuote) all'inizio.

5. [3 punti] Dato il seguente programma, definire la funzione *reversesentence()* ricorsivamente, in modo che si comporti come nell'esempio a destra: legga da tastiera un carattere alla volta, e in caso esso sia $\backslash n$, stampi tutti i caratteri al contrario al contrario.

```
1 #include <stdio.h>
2 void reverseSentence();
3 int main()
4 {
5   printf("Enter a sentence: ");
6   reverseSentence();
7   return 0;
8 }
```

Enter a sentence: margorp emosewa
awesome program

6. [4 punti] Per ogni identificatore di variabile e funzione scrivere se è definito o dichiarato, ed il suo linkage.

```
1  #define A 3
2  int a= A;
3  int a;
4  extern int b;
5  extern int cmp(float, float);
6  static double area(double);
7
8  int* func(int c) {
9  static double e= 4.0;
10 double* f= &e;
11 auto int q= 4;
12 extern int a;
13 // Altri comandi
14 }
15
```

a linea 2 definita/ext
a linea 3 tentativ di def, dichiarata/ext b dichiarata/ext
cmp dichiarata/ext
area dichiarata/int
func definita/ext
c definita/nolinkage
e definita/nolinkage
f definita/nolinkage
q definita/nolinkage
a linea 12 dichiarata/nolinkage

7. [4 punti] Cerchiare le affermazioni vere dato *int a[6]={1707,-761,37,INT_MAX,(INT_MAX+INT_MIN)+1 ,-1}; long long *p = (long long *) a; short *q= (short*) a; p[2]+= 255, q[7]+= 1, q[5]= ∼q[5];* sapendo che i tre tipi usati occupano 8, 4, e 2 byte, con valori rappresentati in *little endian* e complemento a due. Scrivere la mappa di memoria e giustificare le affermazioni (vere o false). Gli operatori | e & ritornano rispettivamente l'*or* e l'*and* bit-a-bit dei due operandi, ∼ è la negazione bit a bit.

(A) $(q[8] > q[11])$ B. $((char^*)(\&p[1]) < p[1])$ (C.) $(q[5] \mid (short^*)(\&a[4]))$ (D.) $(\sim a[4] < \sim a[2])$
(E.) $(q[22] \mid \sim q[22])$ F. $((\sim q[11] + SHORT\_MIN) > q[12])$

8. [4 punti] Quali delle seguenti affermazioni sono vere? Su foglio protocollo giustificare la risposta per tutti i casi.

◯ Le funzioni *malloc()* e *calloc()* restituiscono entrambe un puntatore del tipo passato come parametro.
◯ È possibile liberare la memoria allocata tramite *malloc()* utilizzando il metodo *free()*, oppure assegnando *null* alla variabile puntatore.
◯ Le funzioni *malloc()* e *free()* restituiscono entrambe un puntatore a *void*.
◯ Il puntatore restituito da *realloc()* non è mai lo stesso di quello passato come parametro.
◯ *malloc()* alloca la memoria ed inizializza tutti i byte a 0.
◯ *malloc(5,3)* e *calloc(15)* allocano la stessa quantità di byte in memoria.

Esercizio 8: tutte false
- malloc e callo ritornano un puntatore a void (void*)
- Solo l'utilizzo di free è corretto: assegnando un puntatore a NULL si rende solamente irraggiungibile la memoria, senza deallocarla
- free ritorna void e non void*
- realloc può riallocare la memoria richiesta anche a partire dallo stesso indirizzo (quindi, stesso puntatore ritornato come risultato)
- Questo è il comportamento di calloc, e non di malloc, che alloca memoria senza pulirla
- malloc prende come parametro solamente un valore: la quantità di byte da allocare: l'esempio è sbagliato in partenza, dovrebbe essere malloc(15) e calloc(5,3), che allocherebbero la stessa quantità di byte

Esercizio 5

```
void reverseSentence()
{
    char c;
    scanf("%c", &c);
    if( c != '\n')
    {
       reverseSentence();
       printf("%c",c);
    }
}
```

Esercizio 3

```
struct Node* copy_list(struct Node* list1) {
   if (list1 == NULL) {
    printf("Lista Vuota\n");
    return NULL;
   }
   else {
     struct Node* newList = (struct Node*) malloc(sizeof(struct Node));
     newList -> info = list1 -> info;
     struct Node* pLast =   newList;
     struct Node* pScan = list1 -> pNext;
     while (pScan != NULL) {
        struct Node* newP = (struct Node*) malloc(sizeof(struct Node));
        newP -> info = pscan -> info;
        pLast -> pNext = newP;
        pLast = newP;
        pScan = pScan -> pNext;
     }
    return newList;
  }
}
```

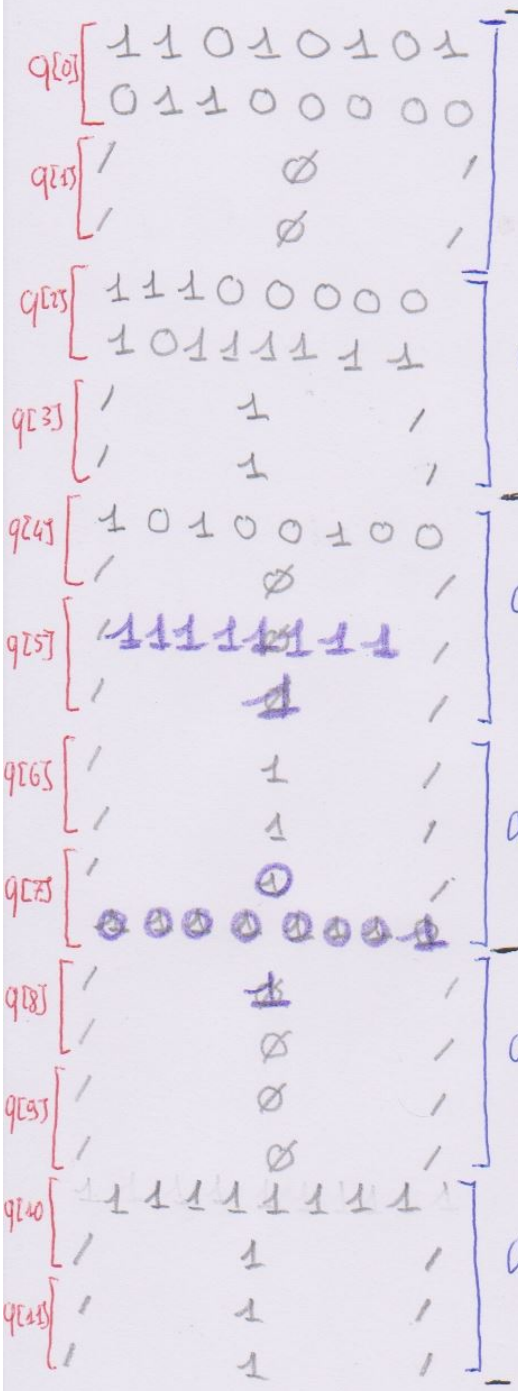
Esercizio 4

```
struct Node* alternate (struct Node* list1, struct Node* list2) {
   // CHECK LISTE VUOTA COME IN ES 3
  struct Node* newList = (struct Node*) malloc(sizeof(struct Node));
  newList -> info = list1 -> info;
  struct Node* newElem2 = (struct Node*) malloc(sizeof(struct Node));
  newElem2 -> info = list2 -> info;
  newList -> pNext = newElem2;
  struct Node* pLast = newElem2;
  struct Node* pScan = list1 -> pNext;
  while (pScan != NULL) {
     \\ INSERIRE il secondo elemento di list1 e il secondo di list2 come sopra
     \\ seguendo lo stesso algoritmo di Esercizio 3
  return newList;
}
```

int a[6] = {1707, -761, 37, INT_MAX, (INT_MAX+INTMIN)+1, -14}

long long *p = (long long*) a;

SHORT * q = (SHORT*) a;



q[5] = ~q[5]

P[1] += 1

P[2] += 255

A. (q[8] > q[11]) T

255 > -1

B. ((CHAR*)(&P[1]) < P[1]) F

37 < N° NEGATIVO

C. (q[5] | (*SHORT)(&a[4])) T

q[6] È COMPOSTO DA TUTTI 1

D. (~a[4] < ~a[2]) T

IL NEGATO DI UN NEGATIVO È POSITIVO E DI UN POSITIVO E NEGATIVO, BANALMENTE VERA

E. (q[22] | ~q[22]) T

ANCHE SE NON CONOSCIAMO A PRIORI q[22], POSSIAMO BANALMENTE OSSERVARE L'OR BIT A BIT DI UNA STRINGA BINARIA E LA SUA NEGATA SARÀ SEMPRE UNA STRINGA COMPOSTA DA TUTTI UNO

F. (~q[11] + SHORT_MIN) > q[12]) F

A PRESCINDERE DAL VALORE DI q[12] LA NEGAZIONE DI q[11] PIÙ IL MINIMO DA IL MINIMO, QUINDI IL MINIMO NON SARÀ MAI PIÙ GRANDE DI UN QUALUNQUE ALTRO NUMERO