Algoritmi 1 – Sperimentazioni
Quiz sul Linguaggio C

MARCO GUAZZONE
DiSIT, Università del Piemonte Orientale
marco.guazzone@uniupo.it

1 Cosa stampa?

Per ciascuno dei seguenti frammenti di codice, dire (senza implementare) cosa viene stampato in output.

* * *

Esercizio 1:

```
1 #define P printf
2 #define NL putchar('\n')
3
4 int x = 10;
5
6 P("%d", x+1); NL;
```

Soluzione 1:

Viene stampato 11 e poi un ritorno a capo. L'istruzione in linea 1 definisce la macro `P` che può essere utilizzata come nome simbolico al posto di `printf`. L'istruzione in linea 2 definisce la macro `NL` che stampa sullo standard output una nuova linea attraverso la funzione `putchar()` della libreria standard del `C`. Quando il codice viene compilato, il preprocessore del `C` sostituisce ogni occorrenza di `P` con `printf` e ogni occorrenza di `NL` con `putchar('\n')`. Per analizzare l'output del preprocessore del `C` con il compilatore `gcc` è possibile utilizzare il seguente comando:

```
1 $ gcc -E file.c
```

* * *

Esercizio 2:

```
1 int x;
2
3 x = - 3 + 4 * 5 - 6; printf("%d\n", x);
4 x = 3 + 4 % 5 - 6; printf("%d\n", x);
5 x = 3 * 4 % 6 / 5; printf("%d\n", x);
6 x = (7 + 6) % 5 / 2; printf("%d\n", x);
```

Soluzione 2:
Viene stampato:

```
11
1
0
1
```

Si noti che:

- L'operatore `%` è l'operatore "modulo" (cioè `x % y` restituisce il resto della divisione tra `x` e `y`).
- Gli operatori "somma" e "sottrazione" hanno una precedenza minore rispetto a "moltiplicazione", "divisione" e "modulo".
- Gli operatori aritmetici binari sono associativi a sinistra, mentre quelli unari sono associativi a destra.

* * *

Esercizio 3:

```
int x, y, z;

x = 2; y = 1; z = 0;
x = x && y || z; printf("%d\n", x);
printf("%d\n", x || ! y && z);

x = y = 1;
z = x ++ - 1; printf("%d\n%d\n", x, z);
z += - x ++ + ++ y; printf("%d\n%d\n", x, z);
```

Soluzione 3:
Viene stampato:

```
1
1
2
0
3
0
```

Si noti che:

- L'operatore "negazione logica" ha una precedenza maggiore dell'operatore "congiunzione logica" che a sua volta ha maggior precedenza dell'operatore "disgiunzione logica".
- Gli operatori "congiunzione logica" e "disgiunzione logica" sono associativi a sinistra, mentre l'operatore "negazione logica" è associativo a destra.

- Con l'operatore "incremento suffisso", l'espressione viene prima valutata e poi incrementata.
- Gli operatori "incremento suffisso" e "meno unario" hanno la stessa precedenza e sono associativi a destra; quindi, se `x` vale 2, nell'espressione `- x ++` si valuta prima `x++`, il cui effetto è quello di ritornare `2` e poi incrementare `x`, e successivamente si applica l'operatore "meno unario" al risultato della valutazione precedente, ottenendo quindi `-2`.

* * *

Esercizio 4:

```
double d = 3.2, x;
int i=2, y;

x = (y=d/i)*2; printf("x=%g, y=%g\n", x, (double) y);
y = (x=d/i)*2; printf("x=%g, y=%g\n", x, (double) y);

y = d * (x=2.5/d); printf("y=%g\n", (double) y);
x = d * (y = ((int) 2.9 + 1.1)/d); printf("x=%g, y=%g\n", x,
    (double) y);
```

Soluzione 4:
Viene stampato:

```
x=2, y=1
x=1.6, y=3
y=2
x=0, y=0
```

Si noti che:

- Il risultato di un'espressione aritmetica tra un valore reale e uno intero è un valore reale (ad es., la valutazione di `3.2/2` produce `1.6`).
- La conversione tra tipo reale e tipo intero viene effettuata tramite troncamento della parte decimale (ad es., la valutazione di `(int) 2.6` produce `2`).
- L'operazione di cast ha precedenza maggiore rispetto agli operatori aritmentici (quindi la valutazione di `(int) 2.9 + 1.1` produce `3.1`).

* * *

Esercizio 5:

```
int x, y, z;

x = y = 0;
while (y < 10) ++y; x += y;
printf("x=%d, y=%d\n", x, y);

y = x = 0;
while (y < 10) x += ++y;
printf("x=%d, y=%d\n", x, y);

y = 1;
while (y < 10)
{
    x = y++;
    z = ++y;
}
printf("x=%d, y=%d, z=%d\n", x, y, z);

for (y = 1; y < 10; ++y) ;
printf("x=%d, y=%d\n", x, y);

for (y = 1; (x=y) < 10; y++) ;
printf("x=%d, y=%d\n", x, y);

for (x = 0, y = 1000; y > 1; ++x, y /= 10)
    printf("x=%d, y=%d\n", x, y);

y = 0;
do
{
    printf("y=%d\n", y);
    --y;
}
while (y > 0);
```

Soluzione 5:
Viene stampato:

```
x=10, y=10
x=55, y=10
x=9, y=11, z=11
x=9, y=10
x=10, y=10
x=0, y=1000
x=1, y=100
x=2, y=10
y=0
```

Si noti che:

- Per tutti i costrutti di ciclo, l'uscita dal ciclo avviene quando la condizione di ciclo diventa falsa.
- La differenza tra **`while`** `(...) {...}` e **`do`** `{ ... }` **`while`** `(...)` è che per quest'ultimo il corpo del ciclo viene eseguito almeno una volta.

* * *

Esercizio 6:

```
1  int reset();
2  int next(int);
3  int last(int);
4  int new(int);
5
6  int i = 1;
7
8  int main()
9  {
10     int i, j;
11
12     i = reset();
13     for (j = 1; j <= 3; ++j)
14     {
15         printf("i = %d, j = %d\n", i, j);
16         printf("next = %d\n", next(i));
17         printf("last = %d\n", last(i));
18         printf("new = %d\n", new(i+j));
19     }
20 }
21
22 int reset()
23 {
24     return i;
25 }
26
27 int next(int j)
28 {
29     return j = i++;
30 }
31
32 int last(int j)
33 {
34     static int i = 10;
35     return j = i--;
36 }
37
38 int new(int i)
39 {
```

```
40      int j = 10;
41      return i= j += i;
42  }
```

Soluzione 6:
Viene stampato:

```
1  i = 1, j = 1
2  next = 1
3  last = 10
4  new = 12
5  i = 1, j = 2
6  next = 2
7  last = 9
8  new = 13
9  i = 1, j = 3
10 next = 3
11 last = 8
12 new = 14
```

Si noti che:

- Le variabili dichiarate all'esterno di qualsiasi funzione sono dette *globali* e sono caratterizzate dall'avere una visibilità (*scope*) estesa a tutte le funzioni che seguono la loro dichiarazione e che non necessariamente appartengono allo stesso file.
- Le variabili globali dichiarate come **`static`** hanno una visibilità limitata alle funzioni dichiarate nello stesso file che contiene le loro dichiarazioni.
- Le variabili dichiarate all'interno di una funzione sono dette *locali* e sono caratterizzate dall'avere una visibilità limitata a quella funzione. Quando la dichiarazione di una variabile locale non contiene le parole chiavi **`static`** o **`extern`**, la variabile è anche detta *automatica* in quanto viene creata al momento della chiamata della funzione che la contiene e la si distrugge quando quest'ultima termina.
- Le variabili locali dichiarate come **`static`** si differenziano dalla variabili automatiche in quanto si crea un'unica copia condivisa da tutte le chiamate alla funzione che le contiene. La loro visibilità rimane sempre locale alla funzione in cui sono dichiarate (cioè, è possibile usarle solo all'interno della funzione che le contiene) ma il loro valore persiste anche fra una chiamata e l'altra della funzione che le contiene.

* * *

Esercizio 7:

```
1  int a[] = {0,1,2,3,4};
2  int i, *p;
3
```

```
for (i = 0; i <= 4; ++i) printf("a[%d]=%d\n", i, a[i]);
putchar('\n');
for (p = &a[0]; p <= &a[4]; ++p)
    printf("*p=%d\n", *p);
printf("\n\n");

for (p = &a[0], i = 1; i <= 5; ++i)
    printf("p[%d]=%d\n", i, p[i]);
putchar('\n');
for (p = a, i = 0; p+i <= a+4; ++p,++i)
    printf("*(p+%d)=%d\n", i, *(p+i));
printf("\n\n");

for (p = a+4; p >= a; --p) printf("*p=%d\n",*p);
putchar('\n');
for (p = a+4, i = 0; i <= 4; ++i) printf("p[%d]=%d\n", -i,
    p[-i]);
putchar('\n');
for (p = a+4; p >= a; --p ) printf("a[%ld]=%d\n", p-a,
    a[p-a]);
putchar('\n');
```

Soluzione 7:
Viene stampato:

```
a[0]=0
a[1]=1
a[2]=2
a[3]=3
a[4]=4

*p=0
*p=1
*p=2
*p=3
*p=4


p[1]=1
p[2]=2
p[3]=3
p[4]=4
p[5]=0

*(p+0)=0
*(p+1)=2
*(p+2)=4


```

```
*p=4
*p=3
*p=2
*p=1
*p=0

p[0]=4
p[-1]=3
p[-2]=2
p[-3]=1
p[-4]=0

a[4]=4
a[3]=3
a[2]=2
a[1]=1
a[0]=0
```

Si noti che:

- Il nome di un array è un alias per l'indirizzo della prima cella dell'array. Quindi, se `a` è un array e `p` è un puntatore allo stesso tipo di dati degli elementi di `a` (ad es., `int a[]` e `int *p`), le espressioni `p=a` e `p=&a[0]` sono equivalenti.
- Per far scorrere un puntatore lungo un array è sufficiente fare in modo che punti a una cella dell'array e poi usare l'aritmetica dei puntatori per incrementare o decrementare la posizione del puntatore. Per esempio, se `p=&a[2]` è un puntatore che punta alla terza cella dell'array `a`, allora l'espressione `p+1` punta alla quarta cella di `a`, mentre `p-2` punta alla prima cella di `a`.
- È del tutto legale usare numeri interi negativi per l'aritmetica dei puntatori e quindi anche con l'operatore di indicizzazione `[]`. Infatti, secondo lo standard *ISO C99* le espressioni `e1[e2]` e `*((e1)+(e2))` sono equivalenti. Quindi, se `p` è un puntatore, le espressioni `p[-1]` e `*(p-1)` sono equivalenti.

* * *

Esercizio 8:

```
int a[] = {0,1,2,3,4};
int *p[] = {a, a+1, a+2, a+3, a+4};
int **pp=p;

printf("*a=%d\n", *a);
printf("**p=%d\n", **p);
printf("**pp=%d\n", **pp);
putchar('\n');

pp++; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p, *pp-a,
    **pp);
```

```
11  *pp++; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p, *pp-a,
        **pp);
12  *++pp; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p, *pp-a,
        **pp);
13  ++*pp; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p, *pp-a,
        **pp);
14  putchar('\n');
15
16  pp = p;
17  **pp++; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p,
        *pp-a, **pp);
18  *++*pp; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p,
        *pp-a, **pp);
19  ++**pp; printf("pp-p=%ld, *pp-a=%ld, **pp=%d\n", pp-p,
        *pp-a, **pp);
20  putchar('\n');
```

Soluzione 8:

Viene stampato:

```
1  *a=0
2  **p=0
3  **pp=0
4
5  pp-p=1, *pp-a=1, **pp=1
6  pp-p=2, *pp-a=2, **pp=2
7  pp-p=3, *pp-a=3, **pp=3
8  pp-p=3, *pp-a=4, **pp=4
9
10 pp-p=1, *pp-a=1, **pp=1
11 pp-p=1, *pp-a=2, **pp=2
12 pp-p=1, *pp-a=2, **pp=3
```

Si noti che:

- Gli operatori di incremento prefissi e suffissi hanno la stessa precedenza e la stessa associatività dell'operatore di dereferenziamento (cioè, associatività a destra). Quindi, nell'espressione `*e++`, si valuta prima l'espressione `e++` (per l'associatività a destra) e poi si applica al risultato l'operatore di dereferenziamento.

* * *

Esercizio 9:

```
1  char *p = "abcdefgh";
2  int *p2 = (int*) p;
3
4  p2 +=1;
5  p = (char*) p2;
6
7  printf("%s\n", p);
```

Soluzione 9:
Viene stampato:

```
efgh
```

Si noti che:

- L'aritmetica dei puntatori tiene conto della dimensione del tipo a cui il puntatore punta. Per esempio, se il tipo `int` occupa 4 byte e `ptr` è un `int*`, allora `++ptr` va avanzare `ptr` di 4 byte.

* * *

Esercizio 10:

```
void incr1(int);
void incr2(int*);
void incr3(int**);

int main()
{
    int v = 1;
    int *p = &v;

    incr1(v); printf("v=%d, p-&v=%ld\n", v, p-&v);
    incr1(p); printf("v=%d, p-&v=%ld\n", v, p-&v);
    incr2(&v); printf("v=%d, p-&v=%ld\n", v, p-&v);
    incr2(p); printf("v=%d, p-&v=%ld\n", v, p-&v);
    incr3(&p); printf("v=%d, p-&v=%ld\n", v, p-&v);

    return 0;
}

void incr1(int x)
{
    ++x;
}

void incr2(int *x)
{
    ++*x;
}

void incr3(int **x)
{
    *++*x;
}
```

Soluzione 10:
Viene stampato:

```
v=1, p-&v=0
v=1, p-&v=0
v=2, p-&v=0
v=3, p-&v=0
v=3, p-&v=1
```

Si noti che:

- In C, il passaggio dei parametri a una funzione è sempre *per valore*. Per emulare il passaggio *per riferimento* per un certo parametro occorre passare un puntatore e quindi modificare il suo contenuto.
- Quando a una funzione si passa un puntatore, il puntatore è passato *per valore* e quindi non può essere modificato. Ciò che può essere invece modificato è il contenuto della cella di memoria a cui il puntatore punta.

2 Qual è il problema?

Per ciascuno dei seguenti frammenti di codice, individuare le problematiche e proporre una soluzione per risolvere.

* * *

Esercizio 11:
Verifica se due numeri sono uguali.

```
int x = 10, y = 5;

if (x = y)
    printf("x e y sono uguali\n");
else
    printf("x e y sono diversi\n");
```

Soluzione 11:
Nella condizione dell'istruzione **if** occorre sostituire = (operatore di assegnamento) con == (operatore di confronto) altrimenti la condizione risulta sempre verificata in quanto il valore di y è diverso da zero. Per risolvere il problema nell'esercizio occorre quindi effettuare la sostituzione suddetta:

```
int x = 10, y = 5;

if (x == y)
...
```

* * *

Esercizio 12:
Verifica un valore.

```
int x;
int y = x+1;

if (y == 1)
    printf("OK\n");
else
    printf("KO\n");
```

Soluzione 12:

L'esecuzione del programma non è ben definita: potrebbe stampare `OK` oppure `KO`. Il problema è l'utilizzo della variabile `x` senza che a essa venga prima esplicitamente assegnato un valore. Lo standard `C`, infatti, non definisce una regola per l'inizializzazione implicita della variabili, e lascia che sia l'implementazione specifica di un compilatore a decidere cosa fare. Per risolvere il problema nell'esercizio occorre quindi inizializzare in modo esplicito la variabile `x` (ad es., impostandola a zero) prima che questa venga utilizzata:

```
int x = 0;
int y = x+1;
...
```

* * *

Esercizio 13:

Limita a una soglia massima.

```
int x = 5;
int max = 10;

if (x > max);
    x = max;

printf("x: %d\n", x);
```

Soluzione 13:

L'esecuzione del programma stampa 10 a causa di un "punto e virgola" di troppo dopo l'istruzione **`if`**. Per risolvere il problema nell'esercizio occorre rimuovere quel carattere:

```
int x = 5;
int max = 10;

if (x > max)
    x = max;
...
```

* * *

Esercizio 14:
Stampa il colore giusto.

```
enum colors_t
{
    red,
    green,
    blue
};

enum colors_t color = green;

switch (color)
{
    case red:
        printf("Red\n");
    case green:
        printf("Green\n");
    case blue:
        printf("Blue\n");
    default:
        printf("Unknown\n");
}
```

Soluzione 14:
L'esecuzione del programma stampa `Green`, `Blue` e `Unknown` in quanto manca l'istruzione **`break`** alla fine di ogni **`case`**. Per risolvere il problema nell'esercizio occorre aggiungere un'istruzione **`break`** per ogni caso dello **`switch`** (per **`default`**, l'uso di **`break`** è opzionale):

```
enum colors_t
{
    red,
    green,
    blue
};

enum colors_t color = green;

switch (color)
{
    case red:
        printf("Red\n");
        break;
    case green:
        printf("Green\n");
        break;
    case blue:
        printf("Blue\n");
        break;
```

```
21      default:
22          printf("Unknown\n");
23  }
```

* * *

Esercizio 15:
Itera da n a 0 (estremi inclusi).

```
1   void itera(unsigned int n)
2   {
3       unsigned int i;
4
5       for (i = n; i >= 0; --i)
6           printf("i: %u\n", i);
7       }
8   }
```

Soluzione 15:
Il ciclo **for** cicla indefinitivamente a causa dell'overflow provocato dall'operatore di decremento quando la variabile di ciclo `i` vale `0`. Una possibile soluzione è utilizzare una nuova variabile di ciclo `j` che assume valori tra `1` e `n+1`, e quindi dichiarare la variabile `i` all'interno del ciclo, assegnandole come valore `j-1`:

```
1   void itera(unsigned int n)
2   {
3       unsigned int j;
4
5       for (j = n+1; j > 0; --j)
6       {
7           unsigned int i = j-1;
8   ...
```

In alternativa, si può utilizzare una nuova variabile di ciclo `j` che iteri in maniera crescente, e quindi chiarare la variabile `i` all'interno del ciclo, assegnandole come valore `n-j`:

```
1   void itera(unsigned int n)
2   {
3       unsigned int j;
4
5       for (j = 0; j <= n; ++j)
6       {
7           unsigned int i = n-j;
8   ...
```

* * *

Esercizio 16:

Itera da n a 0 (estremi esclusi).

```
1  void itera(unsigned int n)
2  {
3      unsigned int i;
4
5      for (i = n-1; i > 0; --i)
6          printf("i: %u\n", i);
7  }
```

Soluzione 16:

Quando `n` è uguale a `0`, il ciclo **`for`** cicla indefinitivamente a causa dell'overflow provocato dall'operatore di sottrazione. Una possibile soluzione è aggiungere un test che permetta di eseguire il ciclo solo per valori di `n` maggiori di `0`:

```
1  void itera(unsigned int n)
2  {
3      if (n > 0)
4      {
5          unsigned int i;
6
7          for (i = n-1; i > 0; --i)
8  ...
```

* * *

Esercizio 17:

Incrementa una variabile.

```
1  int v = 1;
2  int *ptr1 = NULL, ptr2 = NULL;
3
4  ptr1 = &v;
5  ptr2 = ptr1;
6  *ptr2 += 1; *@/printf("
```

Soluzione 17:

L'istruzione in linea 6 non è corretta in quanto si sta applicando l'operatore di dereferenziamento a una variabile (`ptr2`) di tipo intero, anziché a un puntatore. Per risolvere il problema occorre dichiarare `ptr2` come un puntatore a interi, cioè:

```
1  int v = 1;
2  int *ptr1 = NULL, *ptr2 = NULL;
3  ...
```

* * *

Esercizio 18:

Stringhe costanti.

```
1  char ary[] = "Hello";
2  char *ptr = "hello";
3
4  strcpy(ary, "World");
5  strcpy(ptr, "world");
6
7  printf("%p -> %s\n", ary, ary);
8  printf("%p -> %s\n", ptr, ptr);
```

Soluzione 18:

L'esecuzione del programma causa un accesso non valido alla memoria e genera l'errore "Segmentation fault (core dumped)". Ciò è dovuto al fatto che in C i puntatori e gli array non sono la stessa cosa. In questo caso, si può notare la differenza nel modo in cui avviene l'inizializzazione: l'inizializzazione di un array di caratteri con una stringa costante causa la copia di ogni carattere della stringa costante nell'array, mentre l'inizializzazione di un puntatore a caratteri con una stringa costante fà si che il puntatore punti alla prima cella di memoria in cui è salvata la stringa costante. Quindi, l'istruzione problematica è quella in linea 5 in quanto `ptr` (essendo un puntatore a una stringa costante) punta a un'area di memoria immutabile. Ciò non accade per l'istruzione in linea 4 in quanto `ary` è un array di caratteri il cui contenuto è una copia di un stringa costante. Per risolvere il problema, una possibilie soluzione è fare in modo che `ptr` punti a un'area di memoria modificabile, ad esempio utilizzando un array ausiliario:

```
1  char ary[] = "Hello";
2  char ary2[] = "hello";
3  char *ptr = ary2;
4
5  strcpy(ary, "World");
6  strcpy(ptr, "world");
7  ...
```

oppure, utilizzando l'allocazione dinamica della memoria tramite le funzioni `malloc()` e `free()` della libreria standard del C:

```
1  char ary[] = "Hello";
2  char *ptr = NULL;
3
4  ptr = malloc(6); // Sia "hello", sia "world" richiedono 6
       caratteri, incluso fine stringa
5  if (ptr == NULL)
6  {
7      perror("Unable to allocate memory");
8      abort();
9  }
10 strcpy(ptr, "hello");
11
12 strcpy(ary, "World");
13 strcpy(ptr, "world");
```

```
...
free(ptr);
```

In alternativa, si rinuncia all'uso della funzione `strcpy()` e si fa puntare `ptr` a una diversa stringa costante (e quindi a un'area di memoria differente):

```
char ary[] = "Hello";
char *ptr = "hello";

strcpy(ary, "World");
ptr = "world";
...
```

* * *

Esercizio 19:
Operatore **sizeof** su array e puntatori.

```
int ary[] = {0,1,2,3,4,5,6,7,8};
int *ptr = ary;

size_t len_ary = sizeof ary/sizeof ary[0];
size_t len_ptr = sizeof ptr/sizeof *ptr;

if (len_ary == len_ptr)
{
    printf("OK\n");
}
else
{
    printf("ary -> %lu\n", len_ary);
    printf("ptr -> %lu\n", len_ptr);
}
```

Soluzione 19:
Il problema consiste nel fatto che array e puntatori sono due oggetti differenti in C e quindi l'operatore **sizeof** restituisce valori differenti. L'operatore **sizeof** ritorna il numero di byte occupati in memoria dal suo operando. Ci sono due varianti di utilizzo di **sizeof**:

1. **sizeof**(T), dove T è il nome di un tipo di dati (ad es., **sizeof**(**int**)). In questo caso, **sizeof**(T) ritorna il numero di byte richiesti dal tipo di dati T (ad es., **sizeof**(**char**) ritorna 1).

2. **sizeof** var, dove var è il nome di una variabile. Per esempio, se var è una variabile di tipo **char**, **sizeof** var ritorna 1. Quando var è un array, **sizeof** var ritorna il numero totale di byte richiesti dall'intero array. Per esempio, se ary è un array di 9 **int** e se un **int** richiede 4 byte, allora **sizeof** ary ritorna $4 * 9 = 36$. Se var è un puntatore, **sizeof** var ritorna il numero di byte richiesti per memorizzare un indirizzo di memoria. Per esempio, se ptr è un puntatore

a un certo tipo, se gli indirizzi di memoria sono rappresentati mediante il tipo **unsigned long** e se il tipo **unsigned long** richiede 8 byte, allora **sizeof** ptr ritorna 8.

Quindi, mentre è possibile utilizzare l'operatore **sizeof** per conoscere il numero di elementi contenuti in un array (come correttamente fatto in linea 4), non è possibile usarlo con i puntatori (come erroneamente fatto in linea 5). Per risolvere il problema nell'esercizio occorre rimpiazzare l'utilizzo di **sizeof** sul puntatore con il valore corretto:

```
1  int ary[] = {0,1,2,3,4,5,6,7,8};
2  int *ptr = ary;
3
4  size_t len_ary = sizeof ary/sizeof ary[0];
5  size_t len_ptr = len_ary;
6  ...
```

* * *

Esercizio 20:

Lunghezza di una stringa e operatore **sizeof**.

```
1  char ary[] = "Hello, World!";
2  size_t szof = sizeof ary/sizeof ary[0];
3  size_t len = strlen(ary);
4
5  if (szof == len)
6  {
7      printf("OK\n");
8  }
9  else
10 {
11     printf("sizeof -> %lu\n", szof);
12     printf("strlen -> %lu\n", len);
13 }
```

Soluzione 20:

Il problema è che l'operatore **sizeof** considera anche il carattere di terminazione stringhe '\0', mentre la funzione strlen no. Per risolvere il problema nell'esercizio occorre aggiungere (o sottrarre) 1 al valore ritornato da strlen (o da **sizeof**), cioè:

```
1  char ary[] = "Hello, World!";
2  size_t szof = sizeof ary/sizeof ary[0];
3  size_t len = strlen(ary) + 1;
4  ...
```

oppure:

```
1  char ary[] = "Hello, World!";
2  size_t szof = sizeof ary/sizeof ary[0] - 1;
3  size_t len = strlen(ary);
4  ...
```

* * *

Esercizio 21:
Valutazione dell'operatore **sizeof**.

```
1  int x = 0;
2  size_t size = sizeof(++x);
3  printf("size: %lu, x: %d\n", size, x);
```

Soluzione 21:
Il problema è che l'operatore **sizeof** viene valutato a tempo di compilazione, considerando quindi solo il tipo del suo operando. Per risolvere il problema nell'esercizio occorre spezzare l'istruzione in due:

```
1  int x = 0;
2  size_t size = sizeof(x);
3  ++x;
4  ...
```

* * *

Esercizio 22:
Visibilità delle variabili.

```
1  int *get_handle()
2  {
3      int hnd = 10;
4
5      return &hnd;
6  }
7  int main()
8  {
9      int *hnd = get_handle();
10
11     printf("Handle: %d\n", *hnd);
12
13     return 0;
14 }
```

Soluzione 22:
L'esecuzione del programma provoca un accesso non valido alla memoria e genera l'errore "Segmentation fault (core dumped)". Infatti, la variabile automatica `hnd` è locale alla funzione `get_handle()`; per cui la sua visibilità è limitata alla durata del record di attivazione sullo stack delle chiamata relativo all'esecuzione di `get_handle()`. Quando l'esecuzione di `get_handle()` termina, il record di attivazione associato viene distrutto e quindi gli indirizzi di memoria associati ad esso (compresi quelli per le variabili automatiche) non sono più validi. Per risolvere il problema nell'esercizio occorre passare alla funzione un'area di memoria valida in cui memorizzare il valore:

```
void get_handle(int *hnd)
{
    if (hnd != NULL)
        *hnd = 10;
}
int main()
{
    int hnd = -1;

    get_handle(&hnd);

    printf("Handle: %d\n", hnd);
...
```

In alternativa, è possibile utilizzare all'interno di una funzione delle variabili "statiche", la cui visibilità persiste anche al termine della chiamata della funzione, dato che non sono allocate sullo stack di attivazione. Il problema di questo approccio è che ha un effetto collaterale (non necessariamente voluto dal programmatore): il valore della variabile statica diventa modificabile anche all'esterno della funzione dato che se ne restituisce l'indirizzo di memoria:

```
int *get_handle()
{
    static int hnd = 10;

    return &hnd;
}
int main()
{
    int *hnd = get_handle();

    printf("Handle: %d\n", hnd); // OK

    // Effetto collaterale: modifica del valore della
    variabile statica di get_handle()
    *hnd = 20;
    hnd = get_handle();
    printf("Handle: %d\n", hnd);
...
```

* * *

Esercizio 23:
Copia di una stringa.

```
void my_strcpy(char *dest, const char *src)
{
    if (!dest || !src)
        return;

```

```
6       while (*src)
7       {
8           dest = src;
9           ++src;
10          ++dest;
11      }
12  }
```

Soluzione 23:
Ci sono due problemi:

1. La funzione modifica il puntatore della stringa di destinazione anzichè il suo contenuto (si veda la linea 8).

2. Non viene copiato il carattere di fine stringa (si veda la linea 6).

Per risolvere il problema nell'esercizio occorre modificare il valore puntato da `dest` e inserire il carattere di fine stringa all'uscita del ciclo:

```
1   void my_strcpy(char *dest, const char *src)
2   {
3       if (!dest || !src)
4           return;
5
6       while (*src)
7       {
8           *dest = *src;
9           ++src;
10          ++dest;
11      }
12      *dest = '\0';
13  }
```

La stessa funzione può essere implementata in maniera più compatta nel seguente modo:

```
1   void my_strcpy(char *dest, const char *src)
2   {
3       if (!dest || !src)
4           return; // NULL pointer(s)
5
6       while ((*dest++ = *src++)) ;
7   }
```

* * *

Esercizio 24:
Allocazione dinamica della memoria.

```c
struct user_t
{
   char name[100];
};
typedef struct user_t user_t;

user_t *user = malloc(sizeof(user));

strcpy(user->name, "John Doe");
printf("Name: %s\n", user->name);

free(user);
```

Soluzione 24:

L'esecuzione di questo frammento di codice potrebbe generare l'errore "Segmentation fault (code dumped)". Il problema è l'errato numero di byte allocati con la funzione `malloc()`. Infatti, occorre allocare un numero di byte sufficiente a contenere una valore del tipo puntato da `user`, cioè del tipo `user_t`. Tuttavia, l'operando di **`sizeof`** è la variabile `user` che è di tipo `user_t*`. Per risolvere il problema nell'esercizio occorre modificare l'operando di **`sizeof`** nel seguente modo:

```c
struct user_t
{
   char name[100];
};
typedef struct user_t user_t;

user_t *user = malloc(sizeof(user_t));
...
```

3 Strutture dati di base

Implementare le seguenti strutture dati di base. Non è consentito utilizzare variabili globali.

* * *

Esercizio 25:

Implementare il tipo *lista concatenata*, in cui l'informazione memorizzata in ogni elemento è un valore intero, e le seguenti operazioni:

- inserimento di un elemento in testa alla lista (sono ammessi elementi duplicati),
- ricerca di un elemento nella lista (la funzione ritorna il puntatore al nodo della lista che contiene il valore cercato se il valore è presente nella lista, o `NULL` altrimenti),
- cancellazione di un elemento dalla lista (in caso di elementi duplicati, si rimuove la prima occorrenza che s'incontra),

- stampa il contenuto della lista su un file (passato come un parametro di tipo `FILE*`), utilizzando il seguente formato di output: `[valore1, valore2, ...]`.

Nella funzione `main()` effettuare le seguenti operazioni:

1. Inserire nella lista i valori della sequenza `[1,2,3,4,5,4,3,2,1]`.
2. Stampare il contenuto della lista.
3. Rimuovere dalla lista il primo, il secondo e l'ultimo valore della suddetta sequenza, e il valore `-10` (che non è memorizzato nella lista). Ogni volta che si cancella un elemento ricercare nella lista se è ancora presente.
4. Stampare il contenuto della lista.
5. Rimuovere tutti gli elementi.
6. Stampare il contenuto della lista.

L'output prodotto dovrebbe essere simile al seguente:

```
[1,2,3,4,5,4,3,2,1]
After removal -> Element 1 found
After removal -> Element 2 found
After removal -> Element 1 not found
After removal -> Element -10 not found
[3,4,5,4,3,2]
[]
```

Soluzione 25:

Di seguito si fornisce una possibile soluzione. Il tipo lista è realizzato dalla struttura `list_node_t` la quale rappresenta un nodo della lista. Le operazioni d'inserimento, cancellazione, ricerca e stampa sono implementate dalle funzioni `list_insert`, `list_remove`, `list_find` e `list_dump`, rispettivamente.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

struct list_node_t
{
    struct list_node_t *next;
    int data;
};

void list_insert(struct list_node_t **p_list, int v);
struct list_node_t* list_find(struct list_node_t *list, int
    v);
void list_remove(struct list_node_t **p_list, int v);
void list_dump(struct list_node_t *list, FILE *fp);

int main()
{
```

```
18      int a[] = {1,2,3,4,5,4,3,2,1}; // Elements to insert
19      size_t n = sizeof a/sizeof a[0];
20      int xa[] = {a[0],a[1],a[n-1],-10}; // Elements to remove
21      size_t xn = sizeof xa/sizeof xa[0];
22      struct list_node_t *list = NULL;
23
24      // Insert elements
25      for (size_t i = 0; i < n; ++i)
26      {
27          list_insert(&list, a[i]);
28      }
29      list_dump(list, stdout);
30      printf("\n");
31
32      // Remove selected elements and check
33      for (size_t i = 0; i < xn; ++i)
34      {
35          list_remove(&list, xa[i]);
36          if (list_find(list, xa[i]) != NULL)
37          {
38              printf("After removal -> Element %d found\n",
    xa[i]);
39          }
40          else
41          {
42              printf("After removal -> Element %d not
    found\n", xa[i]);
43          }
44      }
45      list_dump(list, stdout);
46      printf("\n");
47
48      // Remove all elements
49      while (list != NULL)
50      {
51          list_remove(&list, list->data);
52      }
53      list_dump(list, stdout);
54      printf("\n");
55
56      return 0;
57  }
58
59  void list_insert(struct list_node_t **p_list, int v)
60  {
61      if (p_list == NULL)
62      {
63          fprintf(stderr, "Invalid list pointer argument\n");
64          abort();
65      }
```

```
66
67      struct list_node_t *node = malloc(sizeof(struct
        list_node_t));
68      if (node == NULL)
69      {
70          perror("Unable to allocate memory for list node");
71          abort();
72      }
73      node->data = v;
74      node->next = *p_list;
75      *p_list = node;
76  }
77
78  struct list_node_t* list_find(struct list_node_t *list, int
        v)
79  {
80      if (list != NULL)
81      {
82          while (list != NULL && list->data != v)
83          {
84              list = list->next;
85          }
86      }
87      return list;
88  }
89
90  void list_remove(struct list_node_t **p_list, int v)
91  {
92      if (p_list == NULL)
93      {
94          fprintf(stderr, "Invalid list pointer argument\n");
95          abort();
96      }
97      struct list_node_t *prev_node = NULL;
98      struct list_node_t *node = *p_list;
99      while (node != NULL && node->data != v)
100     {
101         node = node->next;
102         prev_node = node;
103     }
104     if (node != NULL)
105     {
106         if (prev_node == NULL)
107         {
108             *p_list = (*p_list)->next;
109         }
110         else
111         {
112             prev_node->next = node->next;
113         }
```

```
114             free(node);
115         }
116     }
117
118     void list_dump(struct list_node_t *list, FILE *fp)
119     {
120         if (fp == NULL)
121         {
122             fprintf(stderr, "Invalid file pointer argument\n");
123             abort();
124         }
125
126         const struct list_node_t *head = list;
127         fprintf(fp, "[");
128         while (list != NULL)
129         {
130             if (list != head)
131             {
132                 fputc(',', fp);
133             }
134             fprintf(fp, "%d", list->data);
135
136             list = list->next;
137         }
138         fprintf(fp, "]");
139     }
```

* * *

Esercizio 26:

Implementare il tipo *lista concatenata ordinata*, in cui l'informazione memorizzata in ogni nodo è un valore intero e tale per cui i nodi sono ordinati rispetto al valore memorizzato (quindi un nodo non può contenere un valore inferiore a quelli memorizzati nei nodi precedenti), e le seguenti operazioni:

- inserimento di un elemento nella lista (sono ammessi elementi duplicati),
- ricerca di un elemento nella lista (la funzione ritorna il puntatore al nodo della lista che contiene il valore cercato se il valore è presente nella lista, o `NULL` altrimenti),
- cancellazione di un elemento dalla lista (in caso di elementi duplicati, si rimuove la prima occorrenza che s'incontra),
- stampa il contenuto della lista su un file (passato come un parametro di tipo `FILE*`), utilizzando il seguente formato di output: `[valore1, valore2, ...]`.

Nella funzione `main()` effettuare le seguenti operazioni:

1. Inserire nella lista i valori della sequenza `[1,2,3,4,5,4,3,2,1]`.
2. Stampare il contenuto della lista.

3. Rimuovere dalla lista il primo, il secondo e l'ultimo valore della suddetta sequenza, e il valore `-10` (che non è memorizzato nella lista). Ogni volta che si cancella un elemento ricercare nella lista se è ancora presente.

4. Stampare il contenuto della lista.

5. Rimuovere tutti gli elementi.

6. Stampare il contenuto della lista.

L'output prodotto dovrebbe essere simile al seguente:

```
[1,1,2,2,3,3,4,4,5]
After removal -> Element 1 found
After removal -> Element 2 found
After removal -> Element 1 not found
After removal -> Element -10 not found
[2,3,3,4,4,5]
[]
```

Soluzione 26:

In questo esercizio le varie operazioni della lista devono tenere in considerazione l'ordinamento. Per esempio, quando viene inserito un valore nella lista, occorre inserirlo fra due nodi i cui valori sono rispettivamente $\leq$ e $\geq$ al valore da inserire. Oppure, quando si ricerca un nodo, occorre arrestare la ricerca non appena si trova un valore $\geq$ a quello da cercare; infatti, visto che si garantisce l'ordinamento rispetto a valori dei nodi della lista, se il valore su cui si è arrestata la lista è maggiore di quello da cercare, sicuramente il valore da cercare non è contenuto nella lista.

Di seguito si fornisce una possibile soluzione. Il tipo lista concatenata ordinata è realizzato dalla struttura `olist_node_t` la quale rappresenta un nodo della lista. Le operazioni d'inserimento, cancellazione, ricerca e stampa sono implementate dalle funzioni `olist_insert`, `olist_remove`, `olist_find` e `olist_dump`, rispettivamente.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

struct olist_node_t
{
    struct olist_node_t *next;
    int data;
};

void olist_insert(struct olist_node_t **p_list, int v);
struct olist_node_t* olist_find(struct olist_node_t *list,
    int v);
void olist_remove(struct olist_node_t **p_list, int v);
void olist_dump(struct olist_node_t *list, FILE *fp);

int main()
{
```

```
18      int a[] = {1,2,3,4,5,4,3,2,1}; // Elements to insert
19      size_t n = sizeof a/sizeof a[0];
20      int xa[] = {a[0],a[1],a[n-1],-10}; // Elements to remove
21      size_t xn = sizeof xa/sizeof xa[0];
22      struct olist_node_t *list = NULL;
23
24      // Insert elements
25      for (size_t i = 0; i < n; ++i)
26      {
27          olist_insert(&list, a[i]);
28      }
29      olist_dump(list, stdout);
30      printf("\n");
31
32      // Remove selected elements and check
33      for (size_t i = 0; i < xn; ++i)
34      {
35          olist_remove(&list, xa[i]);
36          if (olist_find(list, xa[i]) != NULL)
37          {
38              printf("After removal -> Element %d found\n",
    xa[i]);
39          }
40          else
41          {
42              printf("After removal -> Element %d not
    found\n", xa[i]);
43          }
44      }
45      olist_dump(list, stdout);
46      printf("\n");
47
48      // Remove all elements
49      while (list != NULL)
50      {
51          olist_remove(&list, list->data);
52      }
53      olist_dump(list, stdout);
54      printf("\n");
55
56      return 0;
57  }
58
59  void olist_insert(struct olist_node_t **p_list, int v)
60  {
61      if (p_list == NULL)
62      {
63          fprintf(stderr, "Invalid list pointer argument\n");
64          abort();
65      }
```

```
66
67      struct olist_node_t *new_node = malloc(sizeof(struct
        olist_node_t));
68      if (new_node == NULL)
69      {
70          perror("Unable to allocate memory for list node");
71          abort();
72      }
73      new_node->data = v;
74      new_node->next = NULL;
75
76      struct olist_node_t *prev_node = NULL;
77      struct olist_node_t *node = *p_list;
78      while (node != NULL && node->data < v)
79      {
80          prev_node = node;
81          node = node->next;
82      }
83
84      if (prev_node == NULL)
85      {
86          new_node->next = *p_list;
87          *p_list = new_node;
88      }
89      else
90      {
91          new_node->next = node;
92          prev_node->next = new_node;
93      }
94  }
95
96  struct olist_node_t* olist_find(struct olist_node_t *list,
        int v)
97  {
98      if (list != NULL)
99      {
100         while (list != NULL && list->data < v)
101         {
102             list = list->next;
103         }
104     }
105     return (list != NULL && list->data == v) ? list : NULL;
106 }
107
108 void olist_remove(struct olist_node_t **p_list, int v)
109 {
110     if (p_list == NULL)
111     {
112         fprintf(stderr, "Invalid list pointer argument\n");
113         abort();
```

```
114     }
115     struct olist_node_t *prev_node = NULL;
116     struct olist_node_t *node = *p_list;
117     while (node != NULL && node->data < v)
118     {
119         prev_node = node;
120         node = node->next;
121     }
122     if (node != NULL)
123     {
124         if (prev_node == NULL)
125         {
126             *p_list = (*p_list)->next;
127         }
128         else
129         {
130             prev_node->next = node->next;
131         }
132         free(node);
133     }
134 }
135
136 void olist_dump(struct olist_node_t *list, FILE *fp)
137 {
138     if (fp == NULL)
139     {
140         fprintf(stderr, "Invalid file pointer argument\n");
141         abort();
142     }
143
144     const struct olist_node_t *head = list;
145     fprintf(fp, "[");
146     while (list != NULL)
147     {
148         if (list != head)
149         {
150             fputc(',', fp);
151         }
152         fprintf(fp, "%d", list->data);
153
154         list = list->next;
155     }
156     fprintf(fp, "]");
157 }
```

* * *

Esercizio 27:

Implementare il tipo *lista concatenata ordinata* di interi come l'esercizio precedente, in cui però non sono ammessi duplicati (cioè in cui non è possibile avere due o più nodi in cui è memorizzato lo stesso valore). In particolare, l'operazione d'inserimento non dovrà inserire nella lista un valore già presente in un suo nodo.

Nella funzione `main()` effettuare le seguenti operazioni:

1. Inserire nella lista i valori della sequenza `[1,2,3,4,5,4,3,2,1]`.
2. Stampare il contenuto della lista.
3. Rimuovere dalla lista il primo, il secondo e l'ultimo valore della suddetta sequenza, e il valore `-10` (che non è memorizzato nella lista). Ogni volta che si cancella un elemento ricercare nella lista se è ancora presente.
4. Stampare il contenuto della lista.
5. Rimuovere tutti gli elementi.
6. Stampare il contenuto della lista.

L'output prodotto dovrebbe essere simile al seguente:

```
[1,2,3,4,5]
After removal -> Element 1 not found
After removal -> Element 2 not found
After removal -> Element 1 not found
After removal -> Element -10 not found
[3,4,5]
[]
```

Soluzione 27:

Rispetto all'esercizio precedente, l'unica operazione che varia è l'inserimento. Infatti, se nella lista esiste già un nodo contenente il valore da inserire, la funzione non deve effettuare alcun inserimento.

Di seguito si fornisce una possibile soluzione.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

struct olist_node_t
{
    struct olist_node_t *next;
    int data;
};

void olist_insert(struct olist_node_t **p_list, int v);
struct olist_node_t* olist_find(struct olist_node_t *list,
    int v);
void olist_remove(struct olist_node_t **p_list, int v);
void olist_dump(struct olist_node_t *list, FILE *fp);

```

```
16 int main()
17 {
18     int a[] = {1,2,3,4,5,4,3,2,1}; // Elements to insert
19     size_t n = sizeof a/sizeof a[0];
20     int xa[] = {a[0],a[1],a[n-1],-10}; // Elements to remove
21     size_t xn = sizeof xa/sizeof xa[0];
22     struct olist_node_t *list = NULL;
23 
24     // Insert elements
25     for (size_t i = 0; i < n; ++i)
26     {
27         olist_insert(&list, a[i]);
28     }
29     olist_dump(list, stdout);
30     printf("\n");
31 
32     // Remove selected elements and check
33     for (size_t i = 0; i < xn; ++i)
34     {
35         olist_remove(&list, xa[i]);
36         if (olist_find(list, xa[i]) != NULL)
37         {
38             printf("After removal -> Element %d found\n",
    xa[i]);
39         }
40         else
41         {
42             printf("After removal -> Element %d not
    found\n", xa[i]);
43         }
44     }
45     olist_dump(list, stdout);
46     printf("\n");
47 
48     // Remove all elements
49     while (list != NULL)
50     {
51         olist_remove(&list, list->data);
52     }
53     olist_dump(list, stdout);
54     printf("\n");
55 
56     return 0;
57 }
58 
59 void olist_insert(struct olist_node_t **p_list, int v)
60 {
61     if (p_list == NULL)
62     {
63         fprintf(stderr, "Invalid list pointer argument\n");
```

```
64          abort();
65      }
66
67      struct olist_node_t *prev_node = NULL;
68      struct olist_node_t *node = *p_list;
69      while (node != NULL && node->data < v)
70      {
71          prev_node = node;
72          node = node->next;
73      }
74
75      if (node != NULL && node->data == v)
76      {
77          // Duplicates not allowed
78          return;
79      }
80
81      struct olist_node_t *new_node = malloc(sizeof(struct
        olist_node_t));
82      if (new_node == NULL)
83      {
84          perror("Unable to allocate memory for list node");
85          abort();
86      }
87      new_node->data = v;
88      new_node->next = NULL;
89
90      if (prev_node == NULL)
91      {
92          new_node->next = *p_list;
93          *p_list = new_node;
94      }
95      else
96      {
97          new_node->next = node;
98          prev_node->next = new_node;
99      }
100 }
101
102 struct olist_node_t* olist_find(struct olist_node_t *list,
        int v)
103 {
104     if (list != NULL)
105     {
106         while (list != NULL && list->data < v)
107         {
108             list = list->next;
109         }
110     }
111     return (list != NULL && list->data == v) ? list : NULL;
```

```
112 }
113
114 void olist_remove(struct olist_node_t **p_list, int v)
115 {
116     if (p_list == NULL)
117     {
118         fprintf(stderr, "Invalid list pointer argument\n");
119         abort();
120     }
121     struct olist_node_t *prev_node = NULL;
122     struct olist_node_t *node = *p_list;
123     while (node != NULL && node->data < v)
124     {
125         prev_node = node;
126         node = node->next;
127     }
128     if (node != NULL && node->data == v)
129     {
130         if (prev_node == NULL)
131         {
132             *p_list = (*p_list)->next;
133         }
134         else
135         {
136             prev_node->next = node->next;
137         }
138         free(node);
139     }
140 }
141
142 void olist_dump(struct olist_node_t *list, FILE *fp)
143 {
144     if (fp == NULL)
145     {
146         fprintf(stderr, "Invalid file pointer argument\n");
147         abort();
148     }
149
150     const struct olist_node_t *head = list;
151     fprintf(fp, "[");
152     while (list != NULL)
153     {
154         if (list != head)
155         {
156             fputc(',', fp);
157         }
158         fprintf(fp, "%d", list->data);
159
160         list = list->next;
161     }
```

```
162     fprintf(fp, "]");
163 }
```

* * *

Esercizio 28:

Si consideri il tipo *pila* in cui gli inserimenti e le cancellazioni seguono la politica *Last-In First-Out* (LIFO; cioè l'ultimo elemento inserito è anche il primo a essere rimosso), e l'unico elemento a cui è possibile accedere è quello in cima alla pila. Implementare il tipo *pila* tramite lista contenata, in cui l'informazione memorizzata in ogni nodo è un valore intero, e le seguenti operazioni:

- inserimento ("push") di un elemento in cima alla pila;
- cancellazione ("pop") di un elemento dalla cima della pila;
- stampa il contenuto della pila su un file (passato come un parametro di tipo `FILE*`), utilizzando il seguente formato di output in cui l'elemento più a sinistra è quello in cima alla pila: `[valore1, valore2, ...]`.

Nella funzione `main()` effettuare le seguenti operazioni:

1. Inserire nella pila i valori della sequenza `[1,2,3,4,5,6,7,8,9]`.
2. Stampare il contenuto della pila.
3. Rimuovere i primi 3 elementi dalla cima della pila.
4. Stampare il contenuto della pila.
5. Rimuovere tutti gli elementi.
6. Stampare il contenuto della pila.

L'output prodotto dovrebbe essere simile al seguente:

```
[9,8,7,6,5,4,3,2,1]
[6,5,4,3,2,1]
[]
```

Soluzione 28:

Di seguito si fornisce una possibile soluzione. Il tipo pila è realizzato dalla struttura `stack_node_t` la quale rappresenta un nodo della pila. Le operazioni push, pop e stampa sono implementate dalle funzioni `stack_push`, `stack_pop`, e `stack_dump`, rispettivamente.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

struct stack_node_t
{
    struct stack_node_t *next;
```

```
8       int data;
9   };
10
11  void stack_push(struct stack_node_t **p_stack, int v);
12  void stack_pop(struct stack_node_t **p_stack);
13  void stack_dump(struct stack_node_t *stack, FILE *fp);
14
15  int main()
16  {
17      int a[] = {1,2,3,4,5,6,7,8,9}; // Elements to insert
18      size_t n = sizeof a/sizeof a[0];
19      size_t xn = 3; // Remove the first xn elements from top
        of the stack
20      struct stack_node_t *stack = NULL;
21
22      // Insert elements
23      for (size_t i = 0; i < n; ++i)
24      {
25          stack_push(&stack, a[i]);
26      }
27      stack_dump(stack, stdout);
28      printf("\n");
29
30      // Remove the first xn elements
31      for (size_t i = 0; i < xn; ++i)
32      {
33          stack_pop(&stack);
34      }
35      stack_dump(stack, stdout);
36      printf("\n");
37
38      // Remove all elements
39      while (stack != NULL)
40      {
41          stack_pop(&stack);
42      }
43      stack_dump(stack, stdout);
44      printf("\n");
45
46      return 0;
47  }
48
49  void stack_push(struct stack_node_t **p_stack, int v)
50  {
51      if (p_stack == NULL)
52      {
53          fprintf(stderr, "Invalid stack pointer argument\n");
54          abort();
55      }
56
```

```
57      struct stack_node_t *node = malloc(sizeof(struct
        stack_node_t));
58      if (node == NULL)
59      {
60          perror("Unable to allocate memory for stack node");
61          abort();
62      }
63      node->data = v;
64      node->next = *p_stack;
65      *p_stack = node;
66  }
67
68  void stack_pop(struct stack_node_t **p_stack)
69  {
70      if (p_stack == NULL)
71      {
72          fprintf(stderr, "Invalid stack pointer argument\n");
73          abort();
74      }
75      struct stack_node_t *node = *p_stack;
76      if (node != NULL)
77      {
78          (*p_stack) = (*p_stack)->next;
79          free(node);
80      }
81  }
82
83  void stack_dump(struct stack_node_t *stack, FILE *fp)
84  {
85      if (fp == NULL)
86      {
87          fprintf(stderr, "Invalid file pointer argument\n");
88          abort();
89      }
90
91      const struct stack_node_t *head = stack;
92      fprintf(fp, "[");
93      while (stack != NULL)
94      {
95          if (stack != head)
96          {
97              fputc(',', fp);
98          }
99          fprintf(fp, "%d", stack->data);
100
101         stack = stack->next;
102     }
103     fprintf(fp, "]");
104 }
```

* * *

Esercizio 29:

Si consideri il tipo *coda* in cui gli inserimenti e le cancellazioni seguono la politica *Fist-In First-Out* (FIFO; cioè il primo elemento inserito è anche il primo a essere rimosso), e gli unici elementi a cui è possibile accedere sono quelli all'inizio ("front") e alla fine ("back") della coda. Implementare il tipo *coda* tramite lista contenata, in cui l'informazione memorizzata in ogni nodo è un valore intero, e le seguenti operazioni:

- inserimento ("enqueue") di un elemento nella coda;
- cancellazione ("dequeue") di un elemento dalla coda;
- stampa il contenuto della coda su un file (passato come un parametro di tipo `FILE*`), utilizzando il seguente formato di output in cui l'elemento più a sinistra è l'emento "front" e quello più a destra è l'elemento "back": `[valore1, valore2, ...]`.

Nella funzione `main()` effettuare le seguenti operazioni:

1. Inserire nella coda i valori della sequenza `[1,2,3,4,5,6,7,8,9]`.
2. Stampare il contenuto della coda.
3. Rimuovere i primi 3 elementi dalla coda.
4. Stampare il contenuto della coda.
5. Rimuovere tutti gli elementi.
6. Stampare il contenuto della coda.

L'output prodotto dovrebbe essere simile al seguente:

```
[1,2,3,4,5,6,7,8,9]
[4,5,6,7,8,9]
[]
```

Soluzione 29:

Di seguito si fornisce una possibile soluzione. Il tipo coda è realizzato dalla struttura `queue_node_t` la quale rappresenta un nodo della coda. Per comodità, gli inserimenti si effettuano in coda e le cancellazioni dalla testa. Occorre mantenere due informazioni: il puntatore all'elemento "front" e quello all'elemento "back". Le operazioni enqueue, dequeue e stampa sono implementate dalle funzioni `queue_enqueue`, `queue_dequeue`, e `queue_dump`, rispettivamente.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

struct queue_node_t
{
    struct queue_node_t *next;
```

```
8       int data;
9   };
10
11  void queue_enqueue(struct queue_node_t **p_front, struct
        queue_node_t **p_back, int v);
12  void queue_dequeue(struct queue_node_t **p_front, struct
        queue_node_t **p_back);
13  void queue_dump(struct queue_node_t *p_front, FILE *fp);
14
15  int main()
16  {
17      int a[] = {1,2,3,4,5,6,7,8,9}; // Elements to insert
18      size_t n = sizeof a/sizeof a[0];
19      size_t xn = 3; // Remove the first xn elements on top of
        the queue
20      struct queue_node_t *queue_front = NULL;
21      struct queue_node_t *queue_back = NULL;
22
23      // Insert elements
24      for (size_t i = 0; i < n; ++i)
25      {
26          queue_enqueue(&queue_front, &queue_back, a[i]);
27      }
28      queue_dump(queue_front, stdout);
29      printf("\n");
30
31      // Remove the first xn elements
32      for (size_t i = 0; i < xn; ++i)
33      {
34          queue_dequeue(&queue_front, &queue_back);
35      }
36      queue_dump(queue_front, stdout);
37      printf("\n");
38
39      // Remove all elements
40      while (queue_front != NULL)
41      {
42          queue_dequeue(&queue_front, &queue_back);
43      }
44      queue_dump(queue_front, stdout);
45      printf("\n");
46
47      return 0;
48  }
49
50  void queue_enqueue(struct queue_node_t **p_front, struct
        queue_node_t **p_back, int v)
51  {
52      if (p_front == NULL || p_back == NULL)
53      {
```

```
54          fprintf(stderr, "Invalid queue pointer(s)
        argument\n");
55          abort();
56      }
57
58      struct queue_node_t *node = malloc(sizeof(struct
        queue_node_t));
59      if (node == NULL)
60      {
61          perror("Unable to allocate memory for queue node");
62          abort();
63      }
64      node->data = v;
65      node->next = NULL;
66      if (*p_back != NULL)
67      {
68          (*p_back)->next = node;
69      }
70      *p_back = node;
71      if (*p_front == NULL)
72      {
73          *p_front = *p_back;
74      }
75  }
76
77  void queue_dequeue(struct queue_node_t **p_front, struct
        queue_node_t **p_back)
78  {
79      if (p_front == NULL || p_back == NULL)
80      {
81          fprintf(stderr, "Invalid queue pointer(s)
        argument\n");
82          abort();
83      }
84
85      struct queue_node_t *node = *p_front;
86      if (node != NULL)
87      {
88          (*p_front) = (*p_front)->next;
89          if (node == *p_back)
90          {
91              *p_back = *p_front;
92          }
93          free(node);
94      }
95  }
96
97  void queue_dump(struct queue_node_t *queue, FILE *fp)
98  {
99      if (fp == NULL)
```

```
100     {
101         fprintf(stderr, "Invalid file pointer argument\n");
102         abort();
103     }
104 
105     const struct queue_node_t *front = queue;
106     fprintf(fp, "[");
107     while (queue != NULL)
108     {
109         if (queue != front)
110         {
111             fputc(',', fp);
112         }
113         fprintf(fp, "%d", queue->data);
114 
115         queue = queue->next;
116     }
117     fprintf(fp, "]");
118 }
```